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

Maintaining the presentation of a continuous media data stream at a sufficient rate and quality for human perception represents a significant challenge for multimedia systems, and may impose significant resource requirements on the multimedia computing environment. Aside from this inherent constraint (sometimes referred to as the problem of intra-media synchronization) a further difficulty arises from the fact that multimedia applications often wish to use several instances of continuous media data at the same time: an animation sequence with some accompanying sound, a video sequence with textual annotations, etc. The difficulty here is that not only should the individual media data be presented with an acceptable quality, but well-defined portions of the various media content should appear, at least from a perceptual point of view, simultaneously: some parts of a sound track belong to a specific animation sequence, subtitles should appear with specified frames in a video sequence, etc. This problem is usually referred to as inter-media synchronization. The specific problems raised by intra-media synchronization will not be addressed in this paper; in what follows, the term synchronization is always used to refer to inter-media synchronization.

Synchronization has received significant attention in the multimedia literature, see, for example, the recent book by Gibbs and Tsichritzis[11] or the article of Koegel Buford[2] for further information and references on the topic. An efficient implementation of inter-media synchronization represents a major load on a multimedia system, and it is one of the major challenges in the field. What emerges from the experience of recent years is that, as is very often the case, one cannot pin down one specific place among all the computing layers (from hardware to the application) where the synchronization problem should be solved. Instead, the requirements of synchronization should be considered across all layers, i.e., in network technology, operating systems, software architectures, programming languages, etc. and user interfaces. This paper describes and formalises a model for inter-media synchronization which is contained in the PREMO specification [16], an ISO/IEC standard under development for multimedia programming. Being part of an upcoming ISO/IEC standard, the model represents a synthesis of the various synchronization techniques used in practice.

PREMO standardization is still at a development stage, hence a short overview of the main goals of this Standard are given below. The remainder of the paper concentrates on the formal specification of these objects. The formal notation used is Object-Z [3, 7, 8], an object-oriented extension to the Z notation [20]. Use of Object-Z to specify PREMO was one of the recommendations of a Special Rapporteur's Report on Formal Description Techniques in PREMO [19] prepared for ISO/IEC JTC1/SC24, and the specification of aspects of PREMO using the Z and Object-Z notations have appeared as [6] and [5]. One contribution of this paper is demonstrating use of a convention for describing exceptions and error handling, and details of the approach are to be found

in Appendix A. An overview of Z and Object-Z is beyond the scope of this paper, and the reader is directed to [12, 20] (Z) and [3, 7, 8] (Object-Z) for details.

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 [13] or [14].

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[15]). 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, a project was started in SC24 for a new standard called PREMO (Presentation Environment for Multimedia Objects) and is now a major ongoing activity in ISO/IEC JTC1 SC24 WG6. The subcommittee's goal is to reach the stage of a Draft International Standard in 1997.

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

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

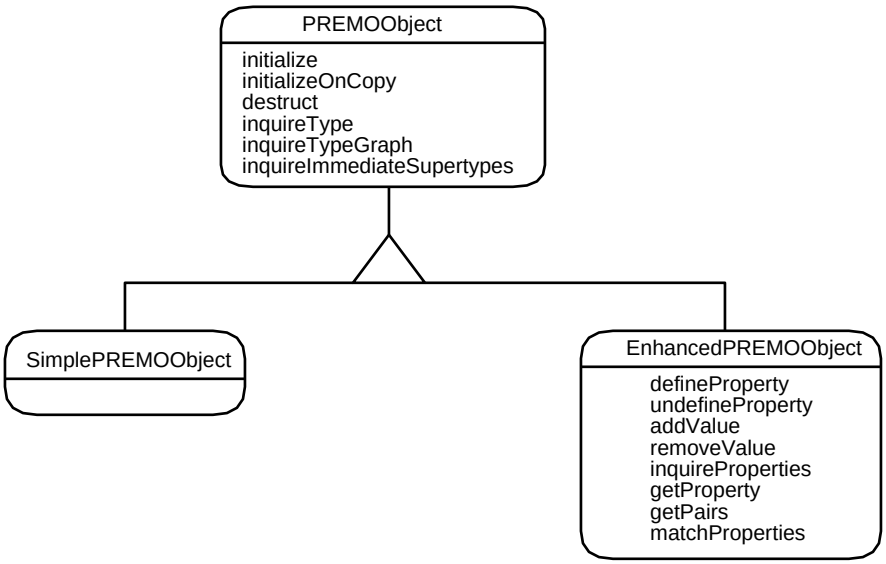

Figure 1: The top of the PREMO Object Type Hierarchy.

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 dependency.

The emphasis on the activity of objects stems primarily from the need for synchronization in multimedia environments and forms the basis of the synchronization model 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, [4] and PREMO, whose task is to provide a synthesis for standardization, has obviously been influenced by these models.

2 Synchronizable Objects

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 application. Examples of how the various synchronization objects may be used can be found in [14].

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 are described in somewhat more detail below.

2.1 Supporting Synchronization in PREMO

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 symbol "C" is used in this section to denote either an extended real, an extended integer, or extended 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.

Technical Note: 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[1]. 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. The use of infinity, for example, 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.

Reference points are points on the internal coordinate space of synchronizable objects where synchronization elements can be attached. Synchronization elements contain information on an event instance (which is, essentially, a structure containing the object reference of the sender, a unique event type identity, and some event-dependent data), a reference to a PREMO object, a reference to one of the operations of this object, and, finally, 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

[1] The same is true actually in many specification languages.

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 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 synchronizable object type defines operations for moving the machine between four different modes:

$$SyncMode == \mathbb{N}$$

$$STOPPED, PLAY, PAUSED, WAITING : SyncMode$$

$$\hline$$

$$STOPPED = 0 \land PLAY = 1 \land PAUSED = 2 \land WAITING = 3$$

Technical Note: The data type $SyncMode$ might better be represented in Z as a disjoint union; the use of explicit numerical tags is used to reflect the approach taken in the Standard. The use of natural numbers to represent modes allows later subtypes to extend the set of modes; the cost is that operations that accept as input a natural number representing a mode have to check that this input describes a mode that is valid for objects of that type.

In more precise terms, a Synchronizable object type is defined in PREMO as a supertype for all objects which may be subject to synchronization. This object is defined to be a finite state machine. The possible states, the state transitions, and the operations resulting in state transitions, are shown in Figure 2. The initial state is STOPPED. Note that no *observable* operation is defined for a transition into state WAITING; the only way a Synchronizable object can go into the WAITING state is through an operation internal to its processing cycle.

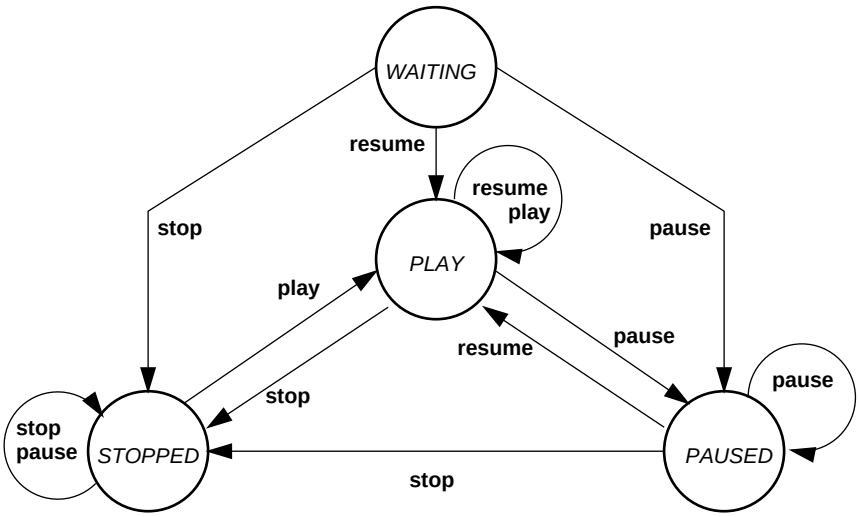

Figure 2: The States and Transitions of the *Synchronizable* Object Type.

The Synchronizable object type places no particular interpretation on these modes, except that certain operations can only be performed in certain modes. Apart from mode, the type also makes use of a notion of direction, which defines the meaning of progress through the coordinate space:

$$Direction ::= Forward \mid Backward$$

The positions within a synchronizable object at which some action needs to take place is marked by a synchronization element. Synchronization is defined in terms of the PREMO event system. The basic unit of this is the event structure, defined below.

Event

SimplePREMOObject

$eventName : String$
$eventData : \text{seq}(Key \times Value)$
$eventSource : EnhancedPREMOObject$

Technical Note: The object type *Event* inherits only from *SimplePREMOObject*; it is used as a structure type to encapsulate information about events. It uses the following data types:

- *String* is a given type representing character strings;
- *Key* is a given type representing names that can be used as the first component of key-value pairs, where the *Value* component is a disjoint union over the non-object data types of PREMO;
- *EnhancedPREMOObject* is the root of the active object hierarchy, and thus the source of an event can be any active PREMO object.

In the ISO documents, the type of *eventSource* is *Ref EnhancedPREMOObject*, since the standard is explicit about the use of object references. 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.

A synchronization element contains a reference to an event handler, plus the event that should be signalled to that handler when the element is activated; the *EventHandler* object type is described in Section 5. The third component of the element is a boolean flag, *wait*. When a synchronization object encounters an element with this flag set to true, it enters pause mode immediately upon signalling the event to the handler. This is necessary to enable synchronization between multiple media objects. Here it may be important to ensure that the position within the media does not advance once the event has been signalled, and because of message delays, network latency, or contention between objects, this control cannot be guaranteed through the use of normal messages.

SyncElement

SimplePREMOObject

$eventHandler : Callback$
$syncEvent : Event$
$waitFlag : \mathbb{B}$

Technical Note: *Callback* is the name of an object type that defines an operation called *callback* that accepts an input *callbackValue?* of type *Event*. By inheriting from *Callback*, the interface of an object type will contain the *callback* operation, and instances of the object type can then be used as targets for event notification. The object type, *CallbackByName*, a subtype of *Callback*, is similar, except that the *eventName* field of the event passed as input to *callback* is taken to be the name of an operation, and in processing the event the receiver invokes that operation on itself.

An action element, like a synchronization element, contains a reference to a *Callback* object, but in this case only the name of an event is provided. The event data will be defined by the context in which the action element is used.

ActionElement

SimplePREMOObject

$eventHandler : Callback$
$eventName : String$

2.2 The 'Synchronizable' Object Type

A specification of the synchronizable object type appears below. The generic parameter C represents the coordinate system restricted to be an extended integer, extended real, or extended time.

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

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

The following declarations are read-only attributes, i.e. each comes with an implicit operation for getting its value, but the value can only be changed by the action of specific operations in the interface of the type.

$currentDirection : Direction$
$loopCounter : \mathbb{N}$ [number of loops completed]
$currentState : SyncMode$ [playing, paused etc]
$currentPosition : C$
$maximumPosition : C$
$minimumPosition : C$ [fixed bounds of the span]

Technical Note: The match between Object-Z access concepts and PREMO access concepts, for example read-only, is not trivial. 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 we do utilise the Z convention of naming an operation schema ΦS if S is not a final operation but a 'frame' that captures a common pattern of processing that will be used as the basis for defining subsequent operations.

The declarations in this fragment of the specification are readable and writable attributes, i.e. each comes with implicit operations for setting and getting its value. The first pair of attributes identify the user-selectable subset of the coordinate space that is to be processed or presented.

$startPosition : C$
$endPosition : C$ [user-definable boundary]

$minimumPosition \leq startPosition$
$startPosition \leq stopPosition$
$stopPosition \leq maximumPosition$

Technical Note: Within the standard, setting the values of these attribute may cause exceptions. This is difficult to document formally in Object-Z, as the set and get methods are not an explicit part of the text. 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.

Two further read/write attributes are provided:

$$repeatFlag : \mathbb{B} \qquad \text{[should the presentation cycle?]}$$
$$nloop : \mathbb{N} \qquad \text{[total number of loops required]}$$

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

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

and we define a total order over this type.

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

No invariant is given at this point to link the coordinates visited during traversal with the parameters that determine traversal behaviour. Operations defined later in this object type can update these parameters, and it simplifies the specification if the relationship between these variables is captured as part of a 'framing' schema that is then used to define the effect of such operations. Here $refpoints$ defines the synchronization elements that have been associated with specific reference points; $loopStart$ is the coordinate that progression will start from initially. The locations that will be passed during traversal define the $span$, while the relation $\prec$ defines the order in which locations will be traversed.

$$refpoints : C \pfun SyncElement \qquad \text{[the sync. points]}$$
$$loopStart : C \qquad \text{[the starting coord for loops]}$$
$$span : \mathbb{P}\, Location \qquad \text{[locations to be traversed]}$$
$$\_ \prec \_ : Location \leftrightarrow Location \qquad \text{[order of traversal]}$$
$$\mathrm{dom}\, refpoints \subseteq minimumPosition \mathinner{..} maximumPosition$$
$$currentDirection = forward \Rightarrow loopStart = startPosition \land (\prec) \subseteq prec$$
$$currentDirection = backward \Rightarrow loopStart = endPosition \land (\prec) \subseteq prec^{-1}$$

Three further variables are used to define how progress is made during play, while moving from processing one item of data in the coordination space to the next.

$$stepping : \mathbb{B} \qquad \text{[true while moving from current to new]}$$
$$requiredPosition : Location \qquad \text{[determined by progressPosition]}$$
$$point : Location \qquad \text{[location in span while moving to requiredPosition]}$$

The final state variable represents actions that may be associated with specific pairs of states, with the semantics that the event handler component of the action will have its 'callback' method invoked with an event containing the name stored in action whenever the synchronizable object makes a transition from the first state to the second.

$$actions : SyncMode \times SyncMode \pfun ActionElement$$
$$\forall\, s, t : SyncMode \bullet (s, t) \in \mathrm{dom}\, actions \Rightarrow s \neq t$$

In the initialisation section below, the initial position is set to the 'front' of the media, and the direction to forward. There are no refpoints, and the state of the object is 'STOPPED'.

INIT

$$
\begin{aligned}
&startPosition = minimumPosition \\
&endPosition = maximumPosition \\
&currentPosition = loopStart \\
&loopCounter = 0 \\
&repeatFlag = False \\
&nloop = 1 \\
&refpoints = \varnothing \\
&currentState = STOPPED \\
&currentDirection = Forward \\
&point = (loopStart, loopCounter)
\end{aligned}
$$

Technical Note: Object-Z officially does not accept parameters to the init description, since init is a state, rather than an operation. This means that some initialisations in PREMO may need to be modelled by explicit operations. More problematic from a specification viewpoint, PREMO objects may also have an $initializeOnCopy$ operation. Copying is part of the operational machinery involved in executing a program on a machine, and is not a concept that can be described (easily) within a specification, without making details of the underlying machine (for example, locations) explicit in the model.

From quite early on in the specification, we will be describing operations that can change the state of a synchronizable object and therefore may cause the invocation of an action associated with the corresponding state change. Rather than repeat the relevant fragment of specification text in every such operation, it is more convenient for us first to define a *framing schema* that captures this common situation, and to then include this as part of the description of those operations that might result in a change to $currentState$.

$\Phi DoAction$

$$
\begin{aligned}
&\Delta(currentState) \\
&\hline
&(currentState, currentState') \in \operatorname{dom} actions \\
&\Rightarrow \\
&\exists\, callbackValue? : Event \bullet \\
&\quad callbackValue?.eventName = actions(currentState, currentState').eventName \\
&\quad callbackValue?.eventSource = self \\
&\quad actions(currentState, currentState').eventHandler.callback
\end{aligned}
$$

The callback operation will be invoked with the event data given in the event structure.

The next part of the specification addresses the meaning of progression through the coordinate space. If the object's state is PLAY, the object carries out its internal processing in a loop of processing stages. Each stage consists of the following steps:

1. The value of the current position is advanced using a (protected) operation $progressPosition$ which returns the required next position. Here 'protected' means 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 C). 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 WAIT 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.

The details of these steps will be explained through a series of operation definitions. Note however that 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.

$progressPosition$

$$\Delta(requiredPosition, stepping)$$
$$newPosition! : C$$

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

Technical Note: Once a new location has been calculated, the object is placed into a 'stepping' mode. This is not a mode behaviour described in the standard, but rather is an artefact of the specification introduced to model the sequence of operations that are assumed to take place internally. 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.

The technical point above raises an important issue about PREMO, which has implications for other standards and systems, for example VRML [17]. 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 be 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 [18].

We return to the semantics of a synchronizable object in *stepping* mode. This is captured through four conceptual operations which represent the aspects of the processing cycle, and which are later composed into a description of stepping behaviour.

The $\Phi step$ operation represent the selection of the next location in the span for which the related data will be presented. Some of the locations between the current point and the new point may be skipped. However, no location for which the underlying coordinate has a reference point can be skipped.
In the formal text, the functions *fst* and *snd* select the first and second components of a tuple respectively.

$\Phi step$

$$\Delta(point, span, loopCounter)$$

$$stepping \land point \neq requiredPosition$$
$$point \prec point'$$
$$\neg\ requiredPosition \prec point'$$
$$\textbf{let}\ skipped == \{loc : span \mid loc \prec point'\} \bullet$$
$$fst(\!|\ skipped\ |\!) \cap \mathrm{dom}\, refpoints = \varnothing$$
$$loopCounter' = loopCounter + |snd(point') - snd(point)|$$
$$span' = span \setminus skipped$$

Technical Note: PREMO itself describes an ideal situation where the progression space may be continuous. The stepping mechanism introduced here has a discrete flavour. There is a correspondence with the sampling and quantization processes that occur when continuous media are processed by digital systems.

The $\Phi signal$ schema is a framing schema that describes the relationship between the location of the current point in the coordination space and the synchronisation point, if any, that is located at that coordinate. If a synchronisation point has been set, then an event is signalled to the appropriate event handler, and further, if the wait flag has been set, the object enters WAITING mode. If no synchronisation element is present, the state of the object is unchanged.

$\Phi signal$

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

$$\textbf{let}\ coord == fst(point) \bullet$$
$$coord \in \mathrm{dom}\, refpoints$$
$$\Rightarrow \left(\begin{array}{l} \exists\ callbackValue? : Event \bullet \\ \quad callbackValue = refpoints(coord).syncEvent \\ \quad refpoints(coord).eventHandler.callback \\ refpoints(coord).wait \Rightarrow currentState' = WAITING \\ \neg\ refpoints(coord).wait \Rightarrow currentState' = currentState \end{array} \right)$$
$$coord \notin \mathrm{dom}\, refpoints$$
$$\Rightarrow currentState' = currentState$$

The callback operation will be invoked with the event data given in the event structure.

The "$eventHandler$" object in the synchronization element (i.e., the object which has to be notified that the synchronizable object has reached a reference point) can be any PREMO object that inherits from the $Callback$ object type. PREMO offers several different types of objects which fulfil this criterion, for example

- so-called "controller" objects, which are essentially finite state machines;
- event handler objects, which can dispatch events among several registered targets;
- other synchronizable objects.

The next two parts of the behaviour description deal with termination of stepping mode. This will happen when:

- the value of $point$ reaches $requiredPosition$ as established by $progressPosition$, or
- while in $WAITING$ mode, the object undergoes a $STOP$ or $PAUSE$ operation, or
- the object completes playing.

The first two cases are described in $\Phi doneStepping$, the latter in $\Phi donePlay$.

$\Phi doneStepping$

$$\Delta(stepping, currentPosition)$$

$$point = requiredPosition \land \neg\ stepping' \land currentPosition' = fst(point)$$

$\Phi donePlay$

$$\Delta(stepping)$$

$$stop$$

[Defined on next page]

$$span = \varnothing \land currentState' = STOPPED \land \neg\ stepping'$$

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.

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

A number of operations are provided to move a synchronizable object from one current state to another, and to modify various attributes that define the behaviour in PLAY mode. As a result of these operations the set of points and the successor relation between points can change. Rather than include the relationships between these parameters as a state invariant, a framing schema called $\Phi UpdateSpan$ defines the relationships as a postcondition. By using this schema as an initial framework for operations that can affect playback parameters, the postcondition of these operations will ensure that the state of the object is left in a state consistent with the most recent settings.

The following framing schema defines the value of the specification variable $span$ that is used to describe how progress occurs through the coordinate space of an object when the current state of the object is 'PLAY'. The schema captures a general invariant that will be required to hold after certain other operations, defined below.
In general the play operation can involve making several iterations through the coordinate space, and thus any coordinate may be visited a number of times. This is captured by the use of the $Location$ type rather than the generic coordinate space parameter C to represent the span.
An auxiliary function is defined to determine the target number of iterations based on the value of the 'repeatFlag' and 'nloop' attributes. Note that, as the value of the 'loopCounter' attribute represents the number of completed loops, its target value in the case that the repeat flag is false is one less than the number of loops required.

$$target : \mathbb{B} \times \mathbb{N} \to \mathbb{N}_\infty$$

$$\begin{aligned}
&\forall\, loops : \mathbb{N} \bullet \\
&\quad target(true, loops) = \infty \\
&\quad target(false, loops) = loops - 1
\end{aligned}$$

The span is updated to the set of all locations where the coordinate lies between the start and end positions and the loop number is between zero and the target computed by the function above, **minus** those locations that have already been visited. This restriction is necessary as a span may need to be recomputed when a synchronizable object has been paused.

ΦUpdateSpan

$$\Delta(span)$$

$$span' = (startPosition \,..\, endPosition) \times (0..target(repeatFlag, nloop)) \setminus \{loc : Location \mid loc \prec (currentPosition, loopCounter)\}$$

The operations that modify the main parameters of a synchronizable object are defined next. Some of these are only applicable in certain 'modes', and therefore come with preconditions.

Technical Note: The notation used in this part of the specification for describing exceptions and error handling is described in Appendix A.

The first pair of operations allow the object to be placed into the state 'PLAY' or 'STOPPED'. The former can only be achieved when the media object is stopped; if this condition is not met, an error is raised. A media object can however be stopped when it is in any state. Stopping an object causes its position and loop counter to be reset to their initial values, and therefore requires 'internal' variables to be updated.

play

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

$$currentState \in \{PLAY, STOPPED\} \longrightarrow \boxed{\text{exc}}\ \text{WrongState}$$

$$currentState' = PLAY$$
$$\text{WrongState} \longrightarrow currentState' = currentState$$

stop

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

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

If the object is PAUSED or WAITING, then it can only react to a very restricted set of operation requests: the attributes of the object may be retrieved (but not set) and the resume, pause or stop operations may be invoked, which may result in a change in state. The difference between PAUSED and WAITING is that, in the latter case, the object returns to the place where it had been suspended by a *Wait* flag, whereas, in the former case,

a complete new processing stage begins. The differentiation between these two states, i.e., the usage of the *Wait* flag, is essential; this mechanism ensures an instantaneous control over the behaviour of the object at a synchronization point. If the object could only be stopped by another object via a pause call, an unwanted race condition could occur.

pause

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

$currentState \neq STOPPED \land currentState' = PAUSED \land \neg\ stepping'$
$\lor$
$currentState = STOPPED \land currentState' = currentState$

resume

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

$currentState \in \{PLAY, PAUSED, WAITING\} \quad \longrightarrow$ exc WrongState

---

$currentState' = PLAY$
WrongState $\longrightarrow\ currentState' = currentState$

The remaining operations are relatively straightforward, and involve setting or retrieving the values of state components. Again, some of these involve preconditions.

The 'current state' of an object can be inquired.

getState

$currentState! : SyncMode$

$currentState! = currentState$

Both the start and end positions within an object can be set to new coordinates, provided that the start position remains strictly less than the end position.

setStartPosition

$\Delta(startPosition)$
$\Phi UpdateSpan$
$startPosition? : C$

$startPosition? < endPosition \quad \longrightarrow$ exc WrongValue
$currentState = STOPPED \quad \longrightarrow$ exc WrongState

---

$startPosition' = startPosition?$
WrongValue $\longrightarrow\ startPosition' = startPosition$

setEndPosition

$\Delta(endPosition)$
$\Phi UpdateSpan$
$endPosition? : C$

$endPosition? > startPosition \quad \longrightarrow$ exc WrongValue
$currentState = STOPPED \quad \longrightarrow$ exc WrongState

---

$endPosition' = endPosition?$
WrongValue $\longrightarrow\ endPosition' = endPosition$

The current value of both the start and end positions can be inquired.

getStartPosition

$startPosition! : C$

$startPosition! = startPosition$

getEndPosition

$endPosition! : C$

$endPosition! = endPosition$

In addition to the progression caused in 'PLAY' state, the position within the coordinate space can also be changed explicitly, using the *jump* operation. The Standard requires that the object is in 'PLAY' or 'WAITING' state, and that the new position must be within the bounds of the object.

jump

$\Delta(currentPosition)$
$\Theta UpdateSpan$
$newPosition? : C$

$currentState \in \{PAUSED, STOPPED\} \longrightarrow$ exc WrongState
$newPosition? \in startPosition \mathinner{..} endPosition \longrightarrow$ exc WrongValue

---

$currentPosition' = newPosition?$
WrongState $\vee$ WrongValue $\longrightarrow currentPosition' = currentPosition$

At any time, the current position can be inquired.

getPosition

$currentPosition! : C$

$currentPosition! = currentPosition$

The *repeat* flag determines what will happen when the position reaches the end of the extent defined by the start and end locators. If true, the position will be reset and playback continues. The value of this flag is set and queried by the pair of operations given below.

setRepeatFlag

$\Delta(repeatFlag)$
$\Theta UpdateSpan$
$r? : \mathbb{B}$

$currentState = STOPPED$
$\longrightarrow$ exc WrongState

$repeatFlag' = r?$

getRepeatFlag

$repeatFlag! : \mathbb{B}$

$repeatFlag! = repeatFlag$

A bounded number of repetitions can be requested by setting the 'number of loops' variable through the operation defined below left; the value of this variable can be enquired through the operation on the right.

setNumberOfLoops

$\Theta UpdateSpan$
$\Delta(nloop)$
$numberOfLoops? : \mathbb{N}$

$currentState = STOPPED$
$\longrightarrow$ exc WrongState

$nloop' = numberOfLoops?$

getNumberOfLoops

$numberOfLoops! : \mathbb{N}$
$loopCounter! : \mathbb{N}$

$numberOfLoops! = nloop$
$loopCounter! = loopCounter$

While in play mode, a loop counter keeps track of the number of loops that have been completed through the media object. This count can be reset to the number of loops specified by $nloop$.

resetLoopCounter

$\Delta(loopCounter)$

$currentState = STOPPED \quad \longrightarrow \boxed{\text{exc}}\ \text{WrongState}$

$loopCounter' = nloop$

The extreme bounds of a given media object can be discovered through the $getBounds$ operation given below.

getBounds

$minimumPosition! : C$
$maximumPosition! : C$

$minimumPosition! = minimumPosition$
$maximumPosition! = maximumPosition$

The remaining operations are those that manipulate the synchronization elements, for example setting a new one at a reference point or deleting an existing one. The first group of these set and delete individual points, and allow enquiry about the set of points between two given coordinates.

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.

setSyncElement

$\Delta(refpoints)$
$refpoint? : C$
$syncData? : SyncElement$

$currentState \in \{PAUSED, STOPPED\} \quad \longrightarrow \boxed{\text{exc}}\ \text{WrongState}$
$refpoint? \in minimumPosition \ldots maximumPosition \quad \longrightarrow \boxed{\text{exc}}\ \text{WrongValue}$

$refpoints' = refpoints \oplus \{refpoint? \mapsto syncData?\}$
$\text{WrongState} \lor \text{WrongValue} \longrightarrow refpoints' = refpoints$

A synchronization element at a given coordinate can be deleted, by passing the coordinate to the $deleteSyncElement$ operation. As before, this cannot be done while the object is in 'PLAY' state, and the specified coordinate must be valid for that object.

deleteSyncElement

$\Delta(refpoints)$
$refpoint? : C$

$currentState \in \{PAUSED, STOPPED\} \quad \longrightarrow \boxed{\text{exc}}\ \text{WrongState}$
$refpoint? \notin \text{dom}\, refpoints \quad \longrightarrow \boxed{\text{exc}}\ \text{WrongValue}$

$refpoints' = \{refpoint?\} \lhd refpoints$
$\text{WrongState} \lor \text{WrongValue} \longrightarrow refpoints' = refpoints$

The synchronization elements occurring between two specified coordinates can be obtained through the operation $getSyncElements$ given below. The sequence of values returned does not necessarily preserve the order of the points; instead, each value in the sequence is a pair consisting of the synchronization element, plus the coordinate at which it occurs.

getSyncElements

$$refpoint_1? : C$$
$$refpoint_2? : C$$
$$syncData! : \text{seq}(SyncElement \times C)$$

$$minimumPosition \leq refpoint_1? \leq refpoint_2? \leq maximumPosition$$
$$\longrightarrow \boxed{\text{exc}}\ \text{WrongValue}$$

$$\textbf{let}\ inrange == \{s : SyncElement;\ c : C \mid refpoint_1? \leq c \leq refpoint_2? \wedge s = refpoints(c)\} \bullet$$
$$\text{ran}\ syncData! = inrange \wedge \#syncData! = \#inrange$$
$$\text{WrongValue} \longrightarrow syncData! = \langle\rangle$$

It is also possible to set and delete *periodic* synchronization elements, in terms of a base coordinate plus offsets from that base.

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

$$\Delta(refpoints)$$
$$startRefPoint? : C$$
$$periodicity? : C$$
$$syncData? : SyncElement$$

$$currentState \in \{PAUSED, STOPPED\} \longrightarrow \boxed{\text{exc}}\ \text{WrongState}$$
$$startRefPoint? \in minimumPosition \ldots maximumPosition \longrightarrow \boxed{\text{exc}}\ \text{WrongValue}$$

$$\textbf{let}\ points == \{n : \mathbb{N} \bullet (startRefPoint? + n \times periodicity?)\} \bullet$$
$$refpoints' = refpoints \oplus \left(\begin{array}{l} (minimumPosition \ldots maximumPosition) \\ \lhd \\ \{p : points \bullet p \mapsto syncData?\} \end{array} \right)$$
$$\text{WrongState} \vee \text{WrongValue} \longrightarrow refpoints' = refpoints$$

Technical comment: 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 sychronization 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.

deletePeriodicSyncElement

$\Delta(refpoints)$
$startRefPoint? : C$
$periodicity? : C$

$currentState \in \{PAUSED, STOPPED\} \longrightarrow$ exc WrongState
$startRefPoint? \in minimumPosition \ldots maximumPosition \longrightarrow$ exc WrongValue

---

let $points == \{n : \mathbb{N} \bullet (startRefPoint? + n \times periodicity?)\} \bullet$
$refpoints' = points \lhd refpoints$
WrongState $\lor$ WrongValue $\longrightarrow refpoints' = refpoints$

The next two operations set and remove actions that are to be invoked on transitions between states.

Technical Note: The handling of actions invoked by state changes is orthogonal to other aspects of the synchronizable object type, and probably would be better defined in a separate super type of synchronizable, both in the specification presented here, and in the Standard.

setActionOnPair

$\Delta(actions)$
$stateOld? : SyncMode$
$stateNew? : SyncMode$
$action? : ActionElement$

$stateOld? \in SyncMode \land stateNew? \in SyncMode \longrightarrow$ exc WrongState

---

$actions' = actions \oplus \{(stateOld?, stateNew?) \mapsto action?\}$
WrongState $\longrightarrow actions' = actions$

removeActionOnPair

$\Delta(actions)$
$stateOld? : SyncMode$
$stateNew? : SyncMode$

$stateOld? \in SyncMode \land stateNew? \in SyncMode \longrightarrow$ exc WrongState

---

$actions' = \{(stateOld?, stateNew?)\} \lhd actions$
WrongState $\longrightarrow actions' = actions$

The final operation clears all of the synchronization elements.

clearSyncElements

$\Delta(refpoints)$

$currentState \in \{PAUSED, STOPPED\} \longrightarrow$ exc WrongState

---

$refpoints' = \varnothing$
WrongState $\lor$ WrongValue $\longrightarrow currentPosition' = currentPosition$

Before concluding this section, it is worth mentioning one further weakness in the specification which is a consequence of the use of the 'internal' $stepping$ mode to define the meaning of progression. From a client's view, the progression of an object consists of atomic steps, and in particular an operation request to an object should only be received and processed outside of 'stepping' activities. In an implementation, progression through the coordinate space would be achieved through the internal invocation of the $progressPosition$ operation, and other operation requests would either be blocked or buffered until that processing was accomplished. In other words, the existence of the $stepping$ mode should be invisible to the environment of an object. However, in the formal specification, operations such as changing the current mode or setting various attributes should have $\neg\ stepping$ as a precondition. Including this would result in a more cluttered specification, and may be misleading to an implementor. In a sense, the approach taken to the specification has forced us to posit a mechanism for explaining the concept of progression. However, it is the behaviour that emerges from this mechanism, rather than its components, that is important. Finding a way of describing such behaviour without introducing implementation details or bias into the specification appears to be a challenge. A more abstract model perhaps based on traces or behaviours might be suitable, but it is unclear how well such an approach would cope with the structural complexity of the $Synchronizable$ object type, or how directly it would map to the text of the Standard.

3 The 'Clock', and Related Object Types

The clock object type provides PREMO with an interface to whatever notion of time is supported by its environment. Specifically, the clock object type supports an operation, $inquireTick$, that returns the number of ticks elapsed. This start of era is defined for all PREMO systems in the Standard. However, the accuracy and units with which a particular PREMO implementation can describe the elasped duration since the start of era will vary, and for this reason the clock object type provides two inquiry functions for determining the performance of the local object. The clock object type assumes the existence of the following two non-object types, one to measure elapsed ticks (realised for example as a 64-bit integer), and the other, an enumerated type, to define the unit represented by each clock tick, for example a year or micro-second.

$[Time]$
$[TimeUnit]$

Clock

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

$ticks : Time$

$\Phi\, Tick$

$\Delta(ticks)$

$ticks' > ticks$

$inquireTick$

$ticks! : Time$

$ticks! = ticks$

Suppose that the output of $inquireTick$ is T, and of $inquireAccuracy$ is A. Suppose the start of era is E. Then mathematically, the actual time in the world when $inquireTick$ is called will be between $E + (T - f(A)/2)$ and $E + (T + f(A)/2)$, i.e $E + T \pm f(A)/2$, where $f(A)$ is a function which converts the accuracy value from its own units to the units of T.

Technical Note: The operation $\Phi\, Tick$ is intended to represent the progression of time by requiring that the number of ticks after the operation occurs is larger than the number of ticks before. How occurrences of the operation should be related to the notion of time in the environment of PREMO is an open problem. Note that this operation is only defined in the specification, and does not appear in the Standard, where the semantics of time are conveyed informally.

A system clock is a clock for which the $inquireTick$ operation returns the number of ticks elapsed since the start of the PREMO era, which in the Standard is defined to be 00:00am, 1st January 1995, UTC.

SysClock

$Clock$ **redef** $(inquireTick)$

inquireTick

$ticks! : TIME$

$ticks! = ticks$

PREMO also provides an object type called a $Timer$, which intuitively behaves like a combination of a clock and a state machine similar to that governing the modes of a synchronizable object. There are three timer states, identified by distinct natural numbers:

$TRUNNING, TSTOPPED, TPAUSED : \mathbb{N}$

$\#\{TRUNNING, TSTOPPED, TPAUSED\} = 3$

The specification of the timer object type appears below; in many ways it is quite similar to the Synchronizable object type, raising a question of whether these two types should be derived from a common supertype in the Standard.

Timer

$Clock$ **redef** $(inquireTick, \Phi Tick)$

$timerCurrentState : \mathbb{N}$

inquireTick

$ticks! : Time$

$ticks! = ticks$

Technical Note: The standard indicates that the $inquireTick$ operation of $Timer$ redefines that of $Clock$, since the implementation of the former may be substantially different. However, in terms of the abstract model, the operations are the same, and so the use of '**redef**' is strictly unnecessary.

run

$\Delta(timerCurrentState)$

$timerCurrentState \in \{TRUNNING, TSTOPPED\}$
$timerCurrentState' = TRUNNING$

pause

$\Delta(timerCurrentState)$

$timerCurrentState' = TPAUSED$

stop

$\Delta(timerCurrentState)$

$timerCurrentState' = TSTOPPED$

resume

$\Delta(timerCurrentState)$

$timerCurrentState \in \{TRUNNING, TPAUSED\}$
$timerCurrentState' = TRUNNING$

reset

$\Delta(ticks)$

$ticks' = 0$

$\Phi\, Tick$

$\Delta(ticks)$

$timerCurrentState = TRUNNING$
$ticks' > ticks$

Note that the internal $\Phi\, Tick$ operation has been redefined to indicate that ticks should only be recorded when an instance of the object type is in $TRUNNING$ mode. The technical comment following the *Clock* object type still applies, however.

4 The 'TimeSynchronizable', and Related Object Types

A *TimeSynchronizable* object type is a synchronizable object type enriched with a 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.

TimeSynchronizable $[T]$

Synchronizable $[T]$ **redef** $(play, stop, resume, pause)$

Timer **redef** $(play, stop, resume, pause)$

$speed : C$

Speed represents the number coordinate space units in C that are processed in PLAY mode per unit tick.

The operations for moving a *TimeSynchronizable* object instance from one mode to another redefine the like-named operations from the two inherited classes. In the new operations, the two machines state machines are slaved, so that a change in mode at the outer level is realised by changing the mode of both component machines.

$$
\begin{aligned}
run &\mathrel{\widehat=} Timer :: run \fatsemi Synchronizable :: run \\
play &\mathrel{\widehat=} Timer :: play \fatsemi Synchronizable :: play \\
stop &\mathrel{\widehat=} Timer :: stop \fatsemi Synchronizable :: stop \\
resume &\mathrel{\widehat=} Timer :: resume \fatsemi Synchronizable :: resume \\
pause &\mathrel{\widehat=} Timer :: pause \fatsemi Synchronizable :: pause
\end{aligned}
$$

The *reset* operation inherited from *Timer* is extended to place a marker on the corresponding point of the coordinate space. Some aspects of the marker concept are unclear, and therefore this part of the specification is tentative. Since the value of the *marker* is persistent, it is introduced as a component of the internal state. From the text, it seems likely that the initial value of *marker* is *startPosition*.

$marker : C$

Init

$marker = startPosition$

reset

$\Delta(marker)$

$marker' = currentPosition$

Two operations are defined for converting between values of time relative to the marker, and positions within the coordinate space.

timeToSpace

$positionTime? : Time$
$positionSpace! : C$

$positionSpace! = marker + positionTime? \times speed$

spaceToTime

$positionSpace? : C$
$positionTime! : Time$

$positionTime! = (positionSpace? - marker)/speed$

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.

Technical Note: Object-Z, like many specification languages, does not allow such overloading, so the following block of specification text would be rejected by a type checker.

$jump \mathrel{\widehat=} timeToSpace \fatsemi Synchronizable :: jump\ [positionSpace!/refPoint?]$

$setSyncElement \mathrel{\widehat=} timeToSpace \fatsemi Synchronizable :: setSyncElement\ [positionSpace!/refPoint?]$

$deleteSyncElement \mathrel{\widehat=} timeToSpace \fatsemi Synchronizable :: deleteSyncElement\ [positionSpace!/refPoint?]$

$setPeriodicSyncElement \mathrel{\widehat=} timeToSpace \fatsemi Synchronizable :: setPeriodicSyncElement\ [positionSpace!/refPoint?]$

$deletePeriodicSyncElement \mathrel{\widehat=} timeToSpace \fatsemi Synchronizable :: setPeriodicSyncElement\ [positionSpace!/refPoint?]$

The final part of this object type is a promoted variant of $getSyncElements$ which uses values of type $Time$ rather than C to define the boundaries for obtaining the synchronization points.

Technical Note: The specification of this operation uses a framing schema to convert inputs of type $Time$ into values of type C. This can then be composed with the $getSyncElements$ operation from the $Synchronizable$ class, though the previous comment concerning overloading still applies.

$\Phi convert$

$refPoint1? : Time$
$refPoint2? : Time$
$rp1! : C$
$rp2! : C$

$self.timeToSpace[rp1!/positionSpace!]$
$self.timeToSpace[rp2!/positionSpace!]$

$$getSyncElements \mathrel{\widehat=} \Phi convert \gg Synchronizable :: getSyncElements\ [rp1?/refPoint1?, rp2?/refPoint2?]$$

The $TimeLine$ object type extends the $TimeSynchronizable$ object type with a constraint that each coordinate represents one unit of time; in other words, the speed is fixed at '1'.

TimeLine

$TimeSynchronizable\ [Time]$

$$speed = 1$$

The remaining time-related object type is called $TimeSlave$, and allows progression of an instance through its coordinate space to be related to that of a 'master' object of type $TimeSynchronizable$.

TimeSlave $[C]$

$TimeSynchronizable\ [C]$

$master : TimeSynchronizable$
$speed : C$

The value of $master$ is either a reference to another $TimeSynchronizable$ object or the $NULLObject$. In the latter case, no synchronization with an external master is done. In the former case, the value of $speed$ is measured in terms of the $master$ object's ticks.

A master-slave relationship allows the calculation of time discrepancies between the clocks of the two objects involved. This requires the slave to have access to a function, called $masterToSlave$ in the specification, that maps ticks of the master clock into ticks of the slave clock. In the specification, this function is defined as a state variable. Also defined in the state are a set of alignment thresholds, consisting of a mapping from $Time$ to $Callback$ object references. The meaning of these are, that when the difference between the internal time of the slave, and the time of the master converted via $masterToSlave$ exceeds a particular threshold, the $callback$ operation of the corresponding object is invoked.

$masterToSlave : Time_\infty \to Time_\infty$
$thresholds : \mathrm{seq}(Time \times Callback)$

In the Standard, the $reset$ operation is redefined to allow data for time discrepencies to be requested from the master. This is not needed in the specification, since the definition of discrepancy assumes the existence of a suitable conversion function, and operations in the specification are timeless.

The current alignment between the slave and master can be inquired using an operation called $inquireAlignment$. It is not clear how this operation should behave in the event that no master has been set. Callbacks for time discrepencies can be set via $setSyncEventHandlers$, and are cleared by providing an empty sequence as input. From this, it is assumed that it is not possible to update the set of callbacks, other than by removing all and asserting a new set.

inquireAlignment

$positionSpace! : Time$

$$positionSpace! = |ticks - masterToSlave(master.ticks)|$$

setSyncEventHandlers

$\Delta(thresholds)$
$syncEventHandlers? : \mathrm{seq}(Time \times Callback)$

$$thresholds' = syncEventHandlers?$$

The remaining part of the specification is an extension to the $\Phi Tick$ operation that was introduced to model the behaviour of time-related objects. It is unclear from the Standard exactly when time discrepencies are checked; here it is assumed that the check takes place after each tick. It is also unclear whether, once callback objects have been notified of a discrepancy, continued deviation should result in further notifications.

$\Phi Tick$

$$
\begin{array}{l}
\textbf{let}\ dev == |ticks - masterToSlave(master.ticks)| \bullet \\
\forall t : Time;\ c : Callback \mid (t, c) \in \operatorname{ran} thresholds \land dev > t \bullet \\
\quad \neg\ \exists u : Time \mid u \in \operatorname{dom} \operatorname{ran} thresholds \land dev > u > t \\
\quad \Rightarrow \quad \exists callbackValue? : Event \mid \\
\qquad callbackValue?.eventName = \text{`OutofSync'} \\
\qquad callbackValue?.eventData = \langle \text{`Discrepency'} \mapsto dev \rangle \\
\qquad callbackValue?.eventSource = self \bullet \\
\qquad c.callback
\end{array}
$$

5 The 'EventHandler' Object Type

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.

$[EventId]$ - Identifiers for event registration.

$[Constraint]$ - The constraint description language.

$AndOr ::= And \mid Or$ - Constraint matching mode

$\_ sat \_ : \operatorname{seq}(Key \times Value) \leftrightarrow Constraint$

The $EventHandler$ object type is an enhanced PREMO object that also inherits from $Callback$ to enable instances of $EventHandler$ to be themselves the targets of event notification.

$EventHandler$

$EnhancedPREMOObject$

$Callback$ **redef** $(callback)$

$$
\begin{array}{l}
registered : \mathbb{F}\ EventId \\
notify : EventId \pfun Callback \\
constraints : EventId \pfun \operatorname{seq} Constraint \\
matchMode : EventId \pfun AndOr \\
eventType : EventId \pfun String \\
\hline
registered = \operatorname{dom} notify = \operatorname{dom} constraints = \operatorname{dom} matchMode = \operatorname{dom} eventType
\end{array}
$$

The (internal) state of an event handler consists of the set of event id's representing registration, together with the information associated with each registration. This consists (in order) of the object that registered (and should therefore be notified), the constraints on the event data required to trigger notification, whether all or any constraints must be met, and the type of event on which notification is to occur.

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

register

$\Delta(registered, notify, constraints, matchMode, eventType)$
$eventType? : String$
$constraints? : \text{seq } Constraint$
$fullConstraintMatchMode? : AndOr$
$objectRef? : Callback$
$id! : EventId$

$id! \notin registered$
$registered' = registered \cup \{id!\}$
$eventType' = eventType \oplus \{id! \mapsto eventType?\}$
$constraints' = constraints \oplus \{id! \mapsto constraints?\}$
$matchMode' = matchMode \oplus \{id! \mapsto fullConstraintMatchMode?\}$
$notify' = notify \oplus \{id! \mapsto objectRef?\}$

unregister

$\Delta(registered, notify, constraints, matchMode, eventType)$
$id? : EventId$

$id? \in registered \quad \longrightarrow$ exc InvalidEventId

---

$registered' = registered \setminus \{id?\}$
$eventType' = \{id?\} \lhd eventType$
$constraints' = \{id?\} \lhd constraints$
$matchMode' = \{id?\} \lhd matchMode$
$notify' = \{id?\} \lhd notify$

InvalidEventId $\longrightarrow$ $registered' = registered \land eventType' = eventType$
$constraints' = constraints \land matchMode' = matchMode$
$notify' = notify$

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.

dispatchEvent

$newEvent? : Event$

$\forall e : registered \bullet$
$eventName(e) = newEvent?.eventName$
$matchMode(e) = AND \Rightarrow \forall c : \text{ran } constraints(e) \bullet newEvent?.eventData \ sat \ c$
$matchMode(e) = OR \Rightarrow \exists c : \text{ran } constraints(e) \bullet newEvent?.eventData \ sat \ c$
$\Rightarrow$
$notify(e).callback \ [newEvent?/callbackValue?]$

Technical Note: 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.

$callback \mathrel{\widehat=} dispatchEvent \ [callbackValue?/newEvent?]$

6 The 'SynchronizationPoint' Object Type

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. One such specialization is defined later in this section. For brevity, we introduce a type name to represent the key-value pairs that are used as the data component of events:

$$EventData == \text{seq}(Key \times Value)$$

The base type for synchronization points is introduced below.

SynchronizationPoint

A Synchronization point is a kind of event handler, but we redefine the dispatchEvent operation as the default behaviour - invoking the operation associated with the event - is no longer legitimate.

$EventHandler$ **redef** $(dispatchEvent)$

The state of an object contains a register of source objects that are synchronizing via an event. This variable relates 'signals', defined as the combination of an event name and event data, with references to the objects that have registered as sources for this event. For example, if $en : String, ed : EventData$ and $o : objref$, then $(en, ed) \mapsto o \in sources$ means that the object o has registered an interest on synchronizing on those en events that carry ed data.

$$events : \mathbb{F}\, Event$$
$$sources : (String \times EventData) \leftrightarrow EnhancedPREMOObject$$
$$\text{dom}\, sources = \{e : events \bullet e.eventName \times e.eventData$$

Initially, the registries of sources and events are empty.

initialise

$$events = \varnothing$$

A new synchronization event can be defined by passing an event containing the event name and data of interest to a synchronization point. The name and data components of the event are used to identify the kind of event being registered, while the $eventSource$ field of the event represents an object that can notify the handler that the event has occurred; notifications from unregistered sources are ignored. In the Standard, attempting to add an event twice results in an exception being generated.

addSyncEvent

$$\Delta(sources, events)$$
$$syncEvent? : Event$$
$$syncEvent? \notin events \quad \longrightarrow \boxed{\text{exc}}\ \text{RepeatedEvent}$$
$$\textbf{let} \left[\begin{array}{l} signal == (syncEvent?.eventName, syncEvent?.eventData) \\ source == syncEvent?.eventSource \end{array} \right] \bullet$$
$$events' = events \cup \{syncEvent?\}$$
$$sources' = sources \cup \{signal \mapsto source\}$$
$$\text{RepeatedEvent} \longrightarrow sources' = sources \land events' = events$$

An event can be removed from the set of registered events; attempting to delete a non-existent event raises an exception.

deleteSyncPoint

$\Delta(sources, received, events)$
$syncEvent? : Event$

$syncEvent? \in events \quad \longrightarrow$ exc UnknownEvent

---

let $\left[\begin{array}{l} signal == (syncEvent?.eventName, syncEvent?.eventData) \\ source == syncEvent?.eventSource \end{array} \right]$
•
$events' = events \setminus \{syncEvent?\}$
$sources' = sources \setminus \{signal \mapsto source\}$
UnknownEvent $\longrightarrow sources' = sources \land received' = received$

The dispatch operation, at this point in the object type hierarchy, extends the corresponding operation inherited from *EventHandler* by checking the validity of the input event, and raising an exception if this event has not previously been registered.

dispachEvent

$newEvent? : Event$

$newEvent? \in events \quad \longrightarrow$ exc UnknownEvent

---

UnknownEvent $\longrightarrow received' = received$

7 The 'ANDSynchronizationPoint' Object Type

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 description of this behaviour in the Standard is somewhat unclear, for example on the relationship between the registration facilities defined in *EventHandler* and the *addSyncEvent* operation inherited from *SynchronizationPoint*. To model the behaviour of this object type, the state is extended with a variable that records event notifications.

ANDSynchronizationPoint

SynchronizationPoint **redef** (*dispatchEvent*)

The Standard indicates all of the operations from *SynchronizationPoint* are redefined in this object type. However, from a specification viewpoint, only the behaviour of the *dispatchEvent* operation need be redefined explicitly; other operations can either be extended, or in the case of *addSyncElement*, inherited without any change.

$received : (String \times EventData) \leftrightarrow EnhancedPREMOObject$

$received \subseteq sources$

The invariant indicates that any notification of an event must correspond to an event that has been registered.

initialise

$received = \varnothing$

The *deleteSyncEvent* operation can remove an event from the set of registered events, and in this case any partial notifications (i.e. cases where some, but not all, of the sources have notified the object of this event) must be

cleared. Note that this operation extends the description of $deleteSyncEvent$ from the $SynchronizationPoint$ object type.

$$
\begin{array}{|l}
\hline
deleteSyncEvent \\
\Delta(received) \\
syncEvent? : Event \\
\hline
\textbf{let}\ signal == (syncEvent?.eventName, syncEvent?.eventData) \bullet \\
received' = \{signal\} \lhd received \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\hline
dispatchEvent \\
\Delta(received) \\
newEvent? : Event \\
\hline
newEvent? \in events \quad \longrightarrow\ \boxed{\text{exc}}\ \text{UnknownEvent} \\
\hline
\textbf{let} \left[\begin{array}{l} signal == (newEvent?.eventName, newEvent?.eventData) \\ source == newEvent?.eventSource \end{array} \right] \\
\bullet \\
received(\!| \{signal\} |\!) \cup \{source\} = sources(\!| \{signal\} |\!) \Rightarrow \\
\quad received' = \{signal\} \lhd received \\
\quad EventHandler :: dispatchEvent \\
\quad \forall e : registered \bullet \\
\qquad eventName(e) = newEvent?.eventName \\
\qquad matchMode(e) = AND \Rightarrow \\
\qquad\quad \forall c : \text{ran}\ constraints(e) \bullet newEvent?.eventData\ sat\ c \\
\qquad matchMode(e) = OR \Rightarrow \\
\qquad\quad \exists c : \text{ran}\ constraints(e) \bullet newEvent?.eventData\ sat\ c \\
\qquad \Rightarrow \\
\qquad notify(e).callback\ [newEvent?/callbackValue?] \\
received(\!| \{signal\} |\!) \cup source \neq sources(\!| \{signal\} |\!) \Rightarrow \\
\quad received' = received \cup \{signal \mapsto source\} \\
\text{UnknownEvent} \longrightarrow received' = received \\
\hline
\end{array}
$$

8 Conclusion

A substantial part of this document was written at the same time as the material that was entered into the Working Draft of the PREMO document. 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 exchange 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 will need 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 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 [5] 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, synchronisation, 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] J.F. Koegel Buford. Architecture and issues for distributed multimedia systems. In *Multimedia Systems*. ACM Press/Addison Wesley, 1994.

[3] 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.

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

[5] 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.

[6] 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), 1994. Conference Issue: Proc. Eurographics'94, Oslo, Norway.

[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 OME-116, ISO/IEC JTC1 SC24/WG6, 1995. 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] S.J. Gibbs and D.C. Tsichritzis. *Multimedia Programming*. ACM Press/Addison-Wesley, 1995.

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

[13] 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, G.J. Reynolds, 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.

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

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

[16] Geneva ISO Central Secretariat. Information processing systems, computer graphics, presentation environment for multimedia objects (PREMO). ISO/IEC 14478, 1996. http://www.cwi.nl/Premo/docs.html.

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

[18] 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.

[19] 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, 1994.

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

A Exception Handling in a Specification

The formal specifications of PREMO Object types published to date have not included explicit descriptions of exceptions. 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.

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

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 $setSyncElement$ operation from the class $Synchronizable$ appears below. What is of concern here is the amount of material needed to describe the two exceptions.

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

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

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

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.