INTERNATIONAL STANDARD

ISO/IEC 14478-2

First edition
1998-12-15

Corrected and reprinted
2000-09-15

Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) —

Part 2: Foundation Component

Technologies de l'information — Infographie et traitement d'images — Environnement de présentation d'objects multimédia (PREMO) —

Partie 2: Composant fondamental

Reference number
ISO/IEC 14478-2:1998(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 14478 may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 14478-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 24, *Computer graphics and image processing*.

ISO/IEC 14478 consists of the following parts, under the general title *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO)*:

— *Part 1: Fundamentals of PREMO*

— *Part 2: Foundation Component*

— *Part 3: Multimedia Systems Services*

— *Part 4: Modelling, rendering and interaction component*

Annexes A and B form a normative part of this part of ISO/IEC 14478. Annex C is for information only.

Introduction

This part of ISO/IEC 14478 defines those object types and non–object types which belong to the Foundation Component. Any conforming PREMO implementation shall support these object types. The description of object types categories are given first and then the foundation object types in each category are described.

Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 2: Foundation Component

1 Scope

This part of ISO/IEC 14478 lists an initial set of object types and non–object types useful for the construction of, presentation of, and interaction with multimedia information. This part is dependent on the PREMO object model defined in clause 8 of ISO/IEC 14478-1. The foundation component does not depend on any other components.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14478. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 14478 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 14478-1:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 1: Fundamentals of PREMO.*

ISO/IEC 11172 (all parts), *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s.*

3 Definitions

3.1 PREMO Part 1 definitions

This part of ISO/IEC 14478 makes use of all terms defined in ISO/IEC 14478-1 (Fundamentals of PREMO).

3.2 Additional definitions

For the purposes of this part of ISO/IEC 14478, the following definitions apply.

3.2.1 basic data type: Non-object data type which cannot be expressed via other data types. Examples are integers, floating point numbers.

3.2.2 constructed data type: As opposed to basic data type; non-object data type which is constructed with the help of permitted type constructors using basic data types.

3.2.3 time: A non–object data type which is appropriate for the representation of real time in the execution environment. It is typically realized through either float numbers or large (64 bit) integers.

3.2.4 extended coordinates: An extension of real, integer, or time coordinates with the symbols $-\infty$ and ∞, and the natural comparison operators. It gives a succinct way of describing unlimited intervals on these coordinate systems.

3.2.5 key–value pair: A constructed data type, consisting of a key (described as a string) and a corresponding value.

3.2.6 foundation object type: Object types defined in the foundation component of PREMO.

3.2.7 structure: A category of object types in PREMO; these objects are characterized through attributes only.

3.2.7.1 structure tag: A synonym for an attribute for a structure.

3.2.8 property: Key with associated value or sequence of values, which can be attached to any PREMO object, and which can be inquired, possibly created and deleted through operations defined on the object.

3.2.8.1 read–only property: A property whose value or values cannot be set by operations of the object.

3.2.9 fundamental object behaviour: Operations defined on the *PREMOObject* type; this type is the supertype of all PREMO object types.

3.2.10 finite state machine: Implementation of an abstract finite state automaton.

3.2.11 constraint: A constructed data type, consisting of a key-value pair and an associated constraint operation; this latter is used to compare the values, in case the keys are identical.

3.2.12 event: A constructed data type, serving as a basic building block for the PREMO Event Model.

3.2.12.1 event source: Object (instance) which creates events. This is a structure tag of an event.

3.2.12.2 event client: Object (instance) which consumes events.

3.2.12.3 event name: A means to denote and/or to refer to a specific event. This name is also referred to as *event type*. This is a structure tag of an event.

3.2.12.4 event data: List of non-object types in the form of key–value pairs attached to an event. This is a structure tag of an event.

3.2.13 event handler: An object which provides event processing services to other objects.

3.2.14 era: The base date for all PREMO systems to measure the amount of elapsed time. This value is set to 00:00am, 1st January 1970, UTC.

3.2.15 reference point: A point in the internal coordinate system of a synchronizable objects, to which a synchronization element is attached.

3.2.16 synchronization element: Synchronization information for a synchronizable object; it contains information on another object and its operation which shall be invoked if synchronization is set up.

3.3.17 capability: Description of the property values an object type can take for a specific key.

3.3.18 native property value: Description of the property value an object instance can take for a specific key.

3.3.19 private properties: Properties of the object which are not defined as part of the functional specification of the object.

The following alphabetical list gives the sub-clause of each definition.

basic data type	3.2.1
capability	3.3.17
constraint	3.2.11
constructed data type	3.2.2
event	3.2.12
event client	3.2.12.2
event data	3.2.12.4
event handler	3.2.13

event name	3.2.12.3
event source	3.2.12.1
event type	3.2.12.3
era	3.2.14
extended coordinates	3.2.4
finite state machine	3.2.10
foundation object type	3.2.6
fundamental object behaviour	3.2.9
key–value pair	3.2.5
native property value	3.3.18
private properties	3.3.19
property	3.2.8
read–only property	3.2.8.1
reference point	3.2.15
structure	3.2.7
structure tag	3.2.7.1
synchronization element	3.2.16
time	3.2.3

4 Symbols and abbreviations

AIFF:	Audio Interchange File Format.
FSM:	Finite State Machine.
IEC:	International Electrotechnical Commission.
IS:	International Standard.
ISO:	International Organization for Standardization.
MPEG:	Moving Picture Experts Group.
PREMO:	Presentation Environments for Multimedia Objects.
2D:	Two-dimensional.
3D:	Three-dimensional.

5 Conformance

A conforming implementation of the PREMO Foundation Component shall comply with the general conformance rules defined in clause 5 of ISO/IEC 14478-1 and the component specification in clause 10.

6 Foundation non-object types

The foundation non–object types in PREMO are defined in two categories: basic data types, and data types directly defined from these basic data types in terms of the notations described in clause A.2 of ISO/IEC 14478-1.

The basic data types are (with their type names):

a) **N**: non-negative integer.

b) **Z**: integer.

c) **R**: real number.

d) *ObjectType*: a data type uniquely identifying an object type.

e) *EventId*: a data type uniquely identifying an event registration for a PREMO event handler.

f) *Time*: a data type to measure progression of real world time. This type is either a real number or a (possibly large) integer. The choice among these is implementation dependent.

g) As described in 8.5 of ISO/IEC 14478-1, for each object of type *T* an object reference type, which is a non–object type, referring to object instances of type *T*, automatically exists in PREMO. As a notational convention, *RefT* denotes the non–object type of object reference referring to object instances of type *T*.

The environment shall provide comparison facilities for each basic data type which unambiguously decide whether two data values are identical or not. In the case of object references the environment shall also include a facility to test whether two references refer to the same object instance or not, or whether the value of the object reference is *NULLObject*. How these facilities are realized depends on the programming language and the execution environment in which PREMO is implemented.

Coordinate spaces can be “extended” to include positive and negative “infinity”. Although the underlying implementation may not have a direct representation of these types, the obvious extension of the notion of “greater than”, “smaller than”, etc., on these types allows the behaviour of objects to be defined more succinctly. The following extended coordinate space definitions are used:

h) Extended real numbers:

$R_{\infty} == \mathbf{R} \cup \{-\infty, \infty\}$

i) Extended integers:

$Z_{\infty} == \mathbf{Z} \cup \{-\infty, \infty\}$

j) Extended time:

$Time_{\infty} == Time \cup \{-\infty, \infty\}$

The foundation object types, described in this part, make also use of a number of (constructed) non–object types, defined formally in 9.2 (page 27). Some of these non–object types play a key role in the behavioural description of several object types; they are therefore also listed here, to make the semantic description in clause 7 easier to follow.

— Boolean:

$Boolean ::= TRUE \mid FALSE$

— Character String:

$String ::= \text{seq } Char$

— Constraint specification for key–value pairs (used, for example, by property management, event handlers, and aggregate object types):

ConstraintOp ::= *Equal* | *NotEqual*
| *GreaterThan* | *GreaterThanOrEqual* | *LessThan* | *LessThanOrEqual*
| *Prefix* | *Suffix* | *NotPrefix* | *NotSuffix*
| *Includes* | *Excludes*

Values in an operation request are constrained to values which satisfy these type constraints and the constructions defined in clause A.2 of ISO/IEC 14478-1 (see also 8.6 of ISO/IEC 14478-1). No particular representation for these values is mandated by the PREMO functional specification, although bindings of PREMO to programming languages or to distributed programming paradigms may specify such formats.

7 Foundation object types

7.1 Introduction

Foundation objects types are those which support a fundamental set of services suitable for use by a wide variety of higher level components. PREMO conformance rules require that, whenever a PREMO implementation includes these objects, they be included in the manner specified in this clause. This is the basis for interoperability. The following criteria are used to identify foundation objects:

a) they are used by a majority of higher level components;

b) together they provide an adequate minimal functional set;

c) they are needed to support output on widely available presentation resources;

d) algorithms exist for decomposing more complex functionality into the foundation object types.

In this clause, foundation object types are identified. By means of subtyping the application developer or component supplier may create objects and object types for their own specific needs. Clause 9 of this part gives the detailed definitions of each of these object types; clause A gives an pictorial overview of all object types defined in this clause.

7.2 PREMO objects and fundamental object behaviour

All PREMO objects are assumed to be subtyped from a type called *PREMOObject*. *PREMOObject* is an abstract type, i.e., it is not instantiable.

Operations on *PREMOObject* type fall into two categories described below.

7.2.1 Creation and destruction of objects

These operations are used by the object and object reference life cycle facilities when object instances are created and destroyed (see 8.11 of ISO/IEC 14478-1 for a detailed description of these facilities). The *initialize*, *initializeOnCopy*, and *destruct* operations are defined to be protected, i.e., no other PREMO object can re–initialize a PREMO object or directly call the *destruct* operation, only through the facilities provided by the environment.

7.2.2 Inquiries on types

These operations return information on the object type, the sequence of supertypes, or the complete type graph of the object. Using the information returned by these operations, complex negotiations are possible to optimize the behaviour of various other PREMO objects.

7.3 Simple PREMO objects

SimplePREMOObject is an abstract subtype of *PREMOObject*. *SimplePREMOObject* does not extend the behaviour of *PREMOObject*, but serves as a common supertype for a family of PREMO objects, called structures. Using such a supertype allows operation specifications to impose type constraints on their arguments.

7.3.1 Structures

The term “structure” does not denote a specific object type in PREMO but, instead, a category of types. These object types are characterized by:

a) they are the subtypes of *SimplePREMOObject* but are *not* subtypes of *EnhancedPREMOObject;*

b) they are not abstract types, although they may be generic types;

c) their behaviour in PREMO is expressed in terms of attributes rather than explicit operations (apart from the operations inherited from the supertype *PREMOObject*).

The attributes of a structure are also referred to as “structure tags”.

NOTE — Implementations, or further components, may define subtypes of structures by adding operations to the type specification. Item c above does *not* preclude this. However, as a use of terminology, such types are not labelled as “structures” any more.

Structures can be used as tools to encapsulate various non–object data into the object hierarchy. As an example, the following object type is used to describe constraints on key–value pairs:

(This structure, formally defined in 9.5, plays an important role in the behavioural description of various objects in PREMO.)

NOTE — To increase the efficiency of the implementations, some programming languages may choose to implement structures as special data types and not as objects.

One of the most important structures used in PREMO is the event structure. Events structures consist of the following structure tags (see 9.5 for the precise specifications): an *event name* that provides a means to denote or refer to the event, also referred to as *event type*, an *event data* which is a sequence of key–value pairs, and the *event source*, which is the reference to the object instance which has created this event.

7.4 Callback objects

Very often object instances have to be notified by other objects on some status change, event occurrences, etc. This is done by ‘registering interest’ in some events. PREMO defines an abstract type, called *Callback*, to facilitate such mechanisms.

The *Callback* object type defines one single asynchronous operation, called *callback*. The signature of this operation consists of one input argument, which is a reference to an *Event* structure (see 9.5.2 for a detailed specification of this structure). Various PREMO objects, which may have to be notified under various circumstances, are defined to be subtypes of *Callback*, defining a type–specific behaviour to the *callback* operation.

NOTE — A typical example for the usage of the callback mechanism is the PREMO Event Model, described in detail in 7.7.1.

Whereas, in simple cases, the semantics of the *callback* operation may be defined to affect the state of the object directly, it is very often the case that this operation acts only as an entry point to call other operations on the object. To facilitate this second case, PREMO also defines a subtype of *Callback*, called *CallbackByName*. The (inherited) asynchronous *callback* operation of *CallbackByName* has the following behaviour: the *eventName* structure tag of the *Event* structure (appearing as the input argument of *callback*) is interpreted to be the name of a local operation which is then internally invoked by the *callback* operation. By default, all other structure tags of the *Event* structure are disregarded by the *callback* operation; subtypes of *CallbackByName* may add an additional behaviour to the operation which also takes these tags into consideration.

7.5 Enhanced PREMO Objects

EnhancedPREMOObject is an abstract type, i.e., is not instantiable. This type describes a set of behaviour, referred to as the *enhanced object behaviour*. The operations on *EnhancedPREMOObject* are related to *object properties*.

Enhanced PREMOObject represents the common, abstract supertype for PREMO objects with a more complex behaviour than, for example, structures. An important restriction in PREMO, which also reflects this characterization, is that only subtypes of *EnhancedPREMOObject* can appear in the ***provides service*** sub–schemas of profile specification (see clause 9 of ISO/IEC 14478-1).

7.5.1 Object properties

Properties are used to store values with an object that may be dynamically defined and are outside of the type system. Properties are pairs of keys and a sequence of values[1)] which are conceptually stored within a PREMO object. Operations are introduced to define, undefine, and inquire properties on PREMO object instances. Because, in general, the same key refers to a sequence of possible values, operations are also defined to add and to delete values from a sequence associated with a key. Properties can be used to implement various naming mechanisms, store information on the location of the object in a network, create annotations on object instances, etc.

Properties may be defined as read only. This means that they cannot be defined through an operation on the object, nor can they, or their associated values, be changed or deleted. Read only properties are typically set by the object when being initialized, and are used to describe the various capabilities of the object.

Properties of an object can also be matched against another list of key–value pairs using the *matchProperties* operation. This operation accepts a sequence of constraints, each defining a sequence of possible values for a specific property key, and returns the sequence of satisfied and unsatisfied constraints. Satisfaction is based on the boolean operation defined by the non–object data type *ConstraintOp*, see 9.2 (page 27), where the left operand of the operation is the value stored in the object, and the right operand of the operation is the value appearing in the argument of *matchProperties* (if the operation does not make sense, the result of the comparison is *FALSE*, i.e., it is the client's responsibility to ensure that the arguments are comparable). This mechanism may be used as part of complex negotiations.

NOTE — An example of using property matching is identifying the possible file formats of an audio service. The object providing the service may define a (read–only) sequence of values for the key "*AudioFormatK*", e.g., <"*AIFF*", "*IRCAM*">, describing the file formats it can use. The *matchProperties* operation may be invoked with a pair consisting of a key and a value, e.g.,

["*AudioFormatK*", "*AIFF*"]

using the comparison operator "Equal". The result will be:

satisfied: ["*AudioFormatK*", <"*AIFF*">]
unsatisfied: ["*AudioFormatK*", <"*IRCAM*">]

Another call, using:

["*AudioFormatK*", "*IRCAM*"]

1) A sequence may have only one element

will result in:

satisfied: [“*AudioFormatK*”, <>]
unsatisfied: [“*AudioFormatK*”, <“*AIFF*”, “*IRCAM*”>]

Based on this information the client may choose the *AIFF* file format which can be managed both by itself and the audio service. By using more than one key in the invocation of the *matchProperties* operation (e.g., also include sampling size), powerful negotiations may be implemented.

As a notational convenience if, in the case of a type hierarchy, it is necessary to stress that a certain property key is defined on a specific type, the notation *Type::key* will be used.

NOTE — For example, the notation *PREMOObject::InternetLocationK* may refer to a property key defined on the type *PREMOObject*; on the other hand, *AudioDevice::InputEncoding* refers to a property which is defined on the type *AudioDevice*, but not (necessarily) on its supertypes.

By default, if a property value is defined for a key which already exists for the object instance, the old value is overwritten. However, the client has the possibility to add the reference of a *Callback* object, with a corresponding event name, to a property key. The callback is activated whenever a new value is defined for the key; the event structure instance sent to the *Callback* object contains the key–value pair corresponding to the new setting.

The detailed functional specification of PREMO objects may contain property specifications, too. This means that the corresponding property keys are automatically defined for these objects at object creation time, together with the values the functional specification may also contain. These keys are therefore always available for an object instance. This property initialization mechanism is conceptually part of the object’s behaviour, and is inherited by all its subtypes.

7.6 Controller objects

A PREMO object may want to communicate with other objects by operation requests. Some of these request may be related to events generated by event sources such as input devices, synchronization requests, etc. An event may require actions by several different objects. The *Controller* object type provides facilities to coordinate this cooperation among the different objects.

An instance of the *Controller* object type is a programmable finite state machine (FSM), i.e., an abstract automaton with a finite number of internal states and a set of state transition rules. Transitions among states are produced when other objects invoke the *handleEvent* operation, which receives a reference to an *Event* structure as input argument. Subtypes of *Controller* define the exact set of states and the corresponding state transition rules; implementations may also choose to defer the definition of states and transition rules for a specific subtype of *Controller* to the final application and offer means to the end–user to program these states and rules. (The set of all possible states for a specific *Controller* instance is defined as a retrieve only attribute of the object, i.e., a client can always find out what states the object may take.) Typically, a controller object registers itself with an event handler or another controller. This event handler can cause a transition of states within the controller object. Actions taken by the FSM may include invoking operations of other objects, including other controller objects. Thus, a hierarchy of controller objects can be built.

NOTE — *Controller* objects can be used for different purposes. An example for their use is to implement complex interactions such as dragging graphical objects. In this interaction, a mouse-down event puts the interaction in a different mode that causes an object to be dragged on the screen using the mouse, until a mouse-up event is received. This can be viewed as an example of an FSM going through different states, where the user events trigger the transitions.

State transitions in controllers may be subject to constraints and are monitorable. Constraining state transitions mean dynamic control over whether a state transition should really occur or not, and this decision may also depend on the data associated to the event appearing as the input argument of *handleEvent*. Monitoring state transitions means that callbacks and local operations can be associated to the various steps of a state transition, i.e., the targets of the callback operations can be notified if a state transition occurs.

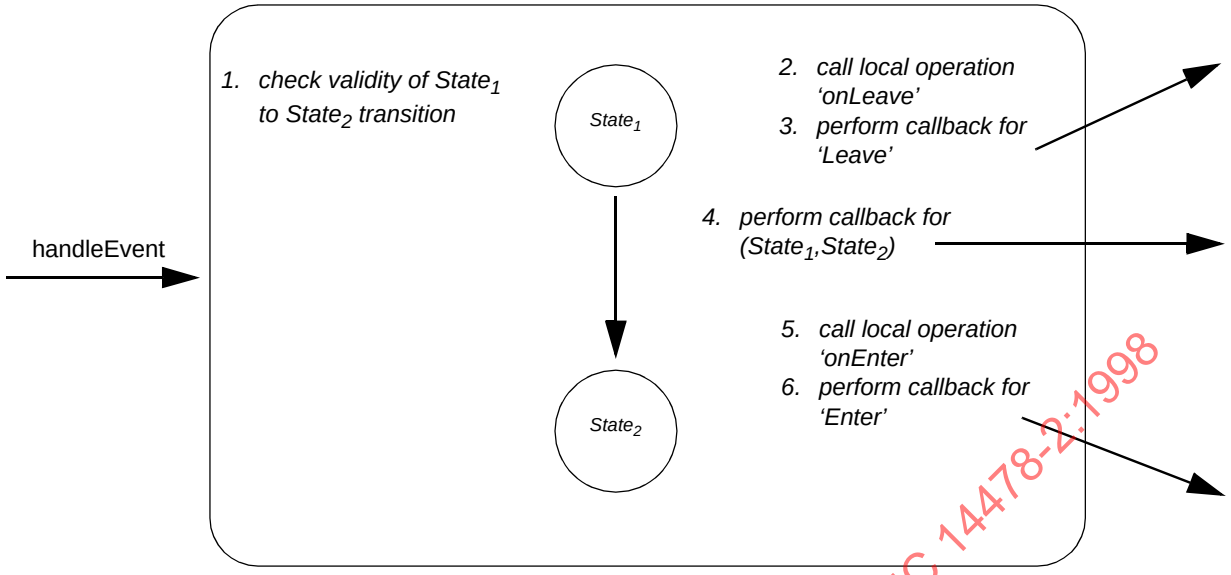


Figure 1 — *Controller* objects

The detailed specification of a controller is as follows (see also Figure 1):

— Each state in a *Controller* is identified by a symbolic name, i.e., a string.

— A special structure type, called *ActionElement*, is defined (see also 9.5.4), containing an event name and a reference to a callback object. Operations are defined on the *Controller* object type to add or remove such *ActionElement* structures to each state. Each state may have two such associated structures, labelled respectively '*Enter*' and '*Leave*'. Also, and independently of the structures associated to the states themselves, such structures may be associated to pairs of states.

— The following protected operations are defined on *Controller*: *checkTransition*, *handleUnknownEvent*, *onLeave*, and *onEnter*. Although a default behaviour is defined for each of these operations, the intention is that subtypes of *Controller* redefine these operations to suit the subtype's behaviour. A retrieve only attribute, denoting the current state, is also defined for *Controller*.

— With these definitions, the basic steps for a state transition of a *Controller* are as follows:

1) A state transition is requested by a client through the invocation of the operation *handleEvent*. The event name is interpreted to be the state name to which the *Controller* object should transit. For the sake of this discussion, $State_1$ is the name of the current state of the object, and $State_2$ is the name of the requested state.

2) The controller object invokes the local, protected operation *checkTransition*, forwarding the argument of *handleEvent*. This operation returns a boolean value indicating whether the transition is allowed.

3) If the transition is not allowed, the local, protected operation *handleUnknownEvent* is invoked, forwarding the argument of *handleEvent*, and *handleEvent* finishes.

4) If the transition is allowed, the following steps are executed:

i) The local, protected operation *onLeave* is invoked. This operation receives as arguments the event structure appearing as the argument of *handleEvent*, as well as (the strings) $State_1$ and $State_2$. The operation returns data suitable for an event data tag in an event structure.

ii) If there is no *ActionElement* structure associated to $State_1$ labelled as '*Leave*', this step is ignored. Otherwise, an event instance is created, using the event name in the *ActionElement* structure associated to $State_1$ labelled as '*Leave*', and the event data returned by *onLeave*. A callback is executed with the newly created event instance as argument.

iii) The value of the attribute denoting the current state is set to $State_2$.

iv) If an *ActionElement* is associated to the tuple $State_1 \times State_2$, a new event instance is created using the event name in the *ActionElement* and the tuple $State_1 \times State_2$ as event data with the key "*Transition*". The *callback* operation on the *Callback* object referenced by the *ActionElement* is then invoked using this new event instance.

v) The local, protected operation *onEnter* is invoked. This operation receives as arguments the event structure appearing as the argument of *handleEvent*, as well as (the strings) $State_1$ and $State_2$. The operation returns data suitable as an event data tag in an event structure.

vi) If there is no *ActionElement* structure associated to $State_2$ labelled as '*Enter*', this step is ignored. Otherwise, an event instance is created, using the event name in the *ActionElement* structure associated to $State_2$ labelled as '*Enter*', and the event data returned by *onEnter*. A callback is executed with the newly created event instance as argument.

NOTE — The *onEnter* and *onLeave* operation may be used, for example, to control the prompt and/or the echo of an elementary step in an interaction.

Controller objects are themselves defined to be subtypes of *Callback*, where the *callback* operation, inherited from *Callback*, is identified with the operation *handleEvent*. Consequently, controller objects may also be chained to form more complex interaction patterns.

7.7 Event handler objects

7.7.1 Basic Event Handler objects

Events form a special category of PREMO structure types and are the basic building block for the PREMO Event Model. This model is based on a small number of basic concepts: events, event registration, and event handling. An *event* can model any action that occurs at a definite time. Events are created by *event sources*, and are consumed by *event clients*, which are both object instances. A basic characteristic of an event is its name, which is one of the features that an event client uses to identify the events in which it is interested.

When using normal operation requests among objects the caller specifies the recipient of each request. When using event handler objects, as shown in Figure 2, events are not addressed to specific recipients. It is the recipient that determines which event types it wishes to receive. Event recipients are operations defined on the recipient object.

EventHandler objects provide the necessary event management services to other objects. This object type provides the following operations (see also Figure 2):

a) register interest in events (the *register* operation),

b) unregister interest in events (the *unregister* operation), and

c) dispatch an event (the asynchronous *dispatchEvent* operation).

The operation to dispatch an event is usually invoked on the *EventHandler* object by the event source. The *EventHandler* will then forward the event to all recipients which have expressed their interest in this specific event.

The choice of the recipients for the event to be handled is determined by the way the recipients have registered their interest in specific event types. This choice is based primarily on the name of the event (i.e., its type), but may also be associated with a constraint list, i.e., a sequence of key–value pairs with constraint operations. This constraint list gives a finer control on whether the event client is notified of the arrival of an event or not. The semantics of the constraint matching is as follows: the key–value pairs of the constraint list are compared to the event data of the incoming event instance. If there is a match in the keys, the comparison operation of the constraint is applied to compare the values. Comparison is based on the boolean operation defined within the structure *ConstraintOp*, (see 9.5), where the left operand of the operation is the value stored in the event handler, and the right operand of the operation is the value appearing in the incoming event instance (if the operation is undefined on these operands, e.g., the types are different, the result of the comparison is *FALSE*). During registration, the prospective event client can control whether the result of the full constraint matching is the logical conjunction or the disjunction of the individual constraint matches. If this result is *TRUE*, the event client is notified; if it is *FALSE*, it is not.

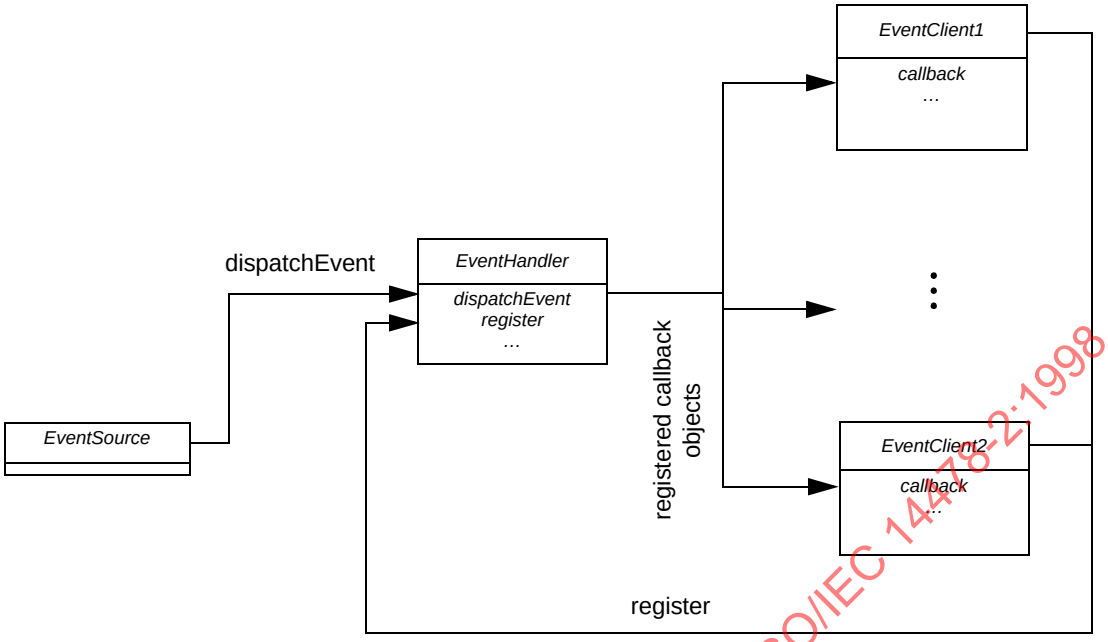


Figure 2 — PREMO event model

Objects, which can be registered within an event handler, shall be subtypes of *Callback* (see 7.4). The event handler calls the operation *callback* on these objects to achieve dispatching.

The registration of an event recipient within an event handler is identified through a non–object data type *EventId*; a value of this type is returned when an event is registered. This value shall be used when, subsequently, the object is no longer interested in this event, and the event is unregistered. Event registration using the *EventId* type is unique across different event handler instances, i.e., two distinct event handler objects shall not share a common *EventId* value. Event registration is also unique for one event handler instance, i.e., an event handler instance shall not reuse an *EventId* value, even if the corresponding event registration has been unregistered.

The *EventHandler* object type is defined as a subtype of *Callback*, where the *callback* operation, inherited from *Callback*, is identified with the operation *dispatchEvent*. Consequently, event handler objects may also be chained to form more complex event handling and filtering.

7.7.2 Synchronization Points

A synchronization point is a subtype of *EventHandler*. Its instances are particularly useful in conjunction with synchronizable objects (see 7.9.1 for more details on the synchronization model in PREMO). Whereas general event handler objects do not impose any general constraint on the dispatched events, synchronization points also maintain an internal set of registered events; events are dispatched if and only if they have been previously registered in this set. Because the event structure also contains a reference to the event source (see 9.2) this restriction also means that only registered object instances may dispatch events through a synchronization point.

A further specialization is offered by the *ANDSynchronizationPoint* object type (defined as a subtype of *SynchronizationPoint*), which redefines the behaviour of event dispatching. In an *ANDSynchronizationPoint* object events are not automatically forwarded to event recipients; instead, the arrival of the event is recorded using a boolean flag associated with each element of the set of registered events. If all events, having the *same* `eventName` and `eventData` values, have this flag set to TRUE, the original behaviour of the *dispatchEvent* operation applies, i.e., the event recipients are notified, and the corresponding flags are set back to *FALSE*.

7.8 Time objects

7.8.1 Clock object

The *Clock* object type is an abstract type which provides PREMO with an interface to any notion of time supported by its environment. The clock object type assumes the existence of two non–object types: *Time*, to measure elapsed ticks (realized, for example, as a 64 bit integer), and *TimeUnit*, which (as an enumerated type) defines the unit represented by each clock tick, for example an hour or a micro–second. Specifically, the clock object type supports an operation, *inquireTick*, to measure the time elapsed since a specific moment. Subtypes of the clock object shall attach a more precise semantic to what kind of value this operation returns.

The accuracy in various units with which particular PREMO implementations can describe the elapsed duration will vary, and for this reason the clock object type defines a retrieve–only attribute to determine the performance of the clock object. This accuracy can be measured in a different unit than the elapsed time. Suppose that the output of *inquireTick* is *T*, and the value of the attribute *accuracy* is *A* (both values are of type *Time*). If the moment used by *inquireTick* as a starting point in time is *E* then, mathematically, the actual time T_r, when *inquireTick* is called, follows the relation:

$$E + T - \frac{f(A)}{2} \leq T_r \leq E + T + \frac{f(A)}{2}$$

where $f(A)$ is the function which converts the accuracy value from its own unit to the units of *T*. I.e., an *accuracy* value with $f(A) = 0$ represents the most accurate timing possible, and increasing values represent a loss in precision.

7.8.2 System clock object

SysClock is a subtype of *Clock*, and provides real–time information (modulo the accuracy of the clock) to PREMO systems. *SysClock* does not add any new operation to *Clock*, but attaches a final semantics to the operation *inquireTick*. *SysClock.inquireTick* is defined to return the number of ticks that have occurred since the start of *era*. This start of era is defined for all PREMO systems to be 00:00am, 1st January 1970, UTC.

7.8.3 Timer object

Timer is a subtype of *Clock*, and provides facilities modelled after a stop–watch. *Timer* is defined as a finite state machine, with the state transition diagram in Figure 3 (see 7.8.3 for a more detailed specification of all possible state transitions).

The *Timer* object contains an internal time register, set to zero either when leaving the *TSTOPPED* state or by an explicit *reset* operation. *Timer.inquireTick* is defined to return the elapsed time the object spent in *TSTARTED* state since the register has been reset to zero (i.e., the time spent in *TPAUSED* is not counted).

7.9 Synchronization

7.9.1 Event Synchronizable objects

Synchronization in PREMO is based on the use of events and possibly event handlers to achieve complex synchronization patterns. Synchronization events are generated by special PREMO objects, called *Synchronizable* objects.

Synchronizable objects are autonomous objects which have an internal progression along an internal one dimensional coordinate space. The specification of the objects makes use of the notion of generic types; the formal type symbol *C* is used to denote this internal coordinate space. This space can be:

a) extended real (R_∞),

b) extended integer (Z_∞), or

c) extended time ($Time_\infty$).

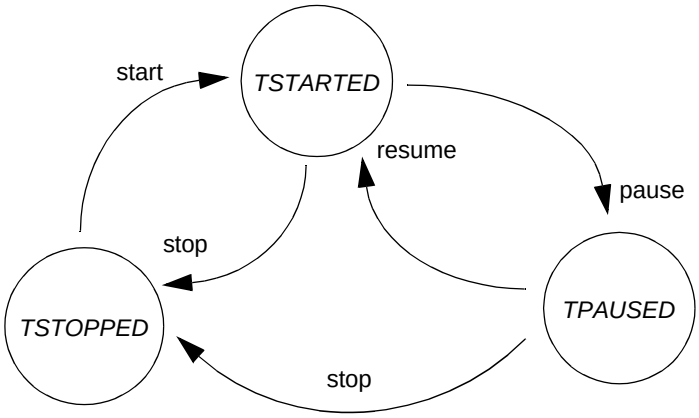


Figure 3 — State transitions in a *Timer* object

See clause 6 (page 3) for the specification of these extended spaces. Subtypes of synchronizable objects may add a semantic meaning to this coordinate space, e.g., media objects (audio, video, etc.) may represent time, or video frame numbers. Attributes of this progression (e.g., the span, i.e., the relevant interval on this coordinate space) can be set through appropriate operations. This coordinate space will also be referred to as the (native) progression space of the *Synchronizable* object.

Reference points are points on the progression space of synchronizable objects where *synchronization elements* can be attached. Synchronization elements contain information on an event instance, a reference to a *Callback* object (this object is typically an event handler, a controller, or another synchronizable object), and a boolean "wait" flag. When a reference point is reached, the synchronizable object uses the *Callback* object reference in the synchronization element to dispatch the event by calling the *callback* operation, and possibly suspends itself if the "wait" flag is set to *TRUE*. Through this mechanism the synchronizable object can stop other objects, restart them, suspend them, etc. The use of the subtypes of event handlers oriented toward synchronization (see 7.7.2, 9.9.3, and 9.9.3 of this part) gives the possibility to create more complex synchronization patterns.

Operations are defined to set and retrieve synchronization elements, either individually, or in a sequence, using a base reference point and a sequence of offsets from that base. Similar operations are defined to delete a reference point; there is also an operation to delete all reference points in one step.

In more precise terms, a *Synchronizable* object is defined to be a finite state machine. The different states are described in more detail in 7.9.1.1 below. The possible states, important state transitions, and the operations resulting in state transitions, are in Figure 4 on page 14. The figure does not include all state transitions; some of the trivial ones (i.e., transition from *STOPPED* state to *STOPPED* state) are not depicted. See 9.11 for a detailed specification for all possible state transition operations. Note that no operation is defined to transit into state *WAITING*; the only way a *Synchronizable* object can go into *WAITING* state is through its internal processing cycle (see 7.9.1.1.1 below). The initial state is *STOPPED*.

The behaviour of the object is described using the following non–object values, stored as part of the object's state:

currentDirection:	*Direction* (can be either *Forward* or *Backward*)
startPosition:	*C*
endPosition:	*C*
currentPosition:	*C*
repeatFlag:	*Boolean*
nloop:	**N**
loopCounter:	**N**

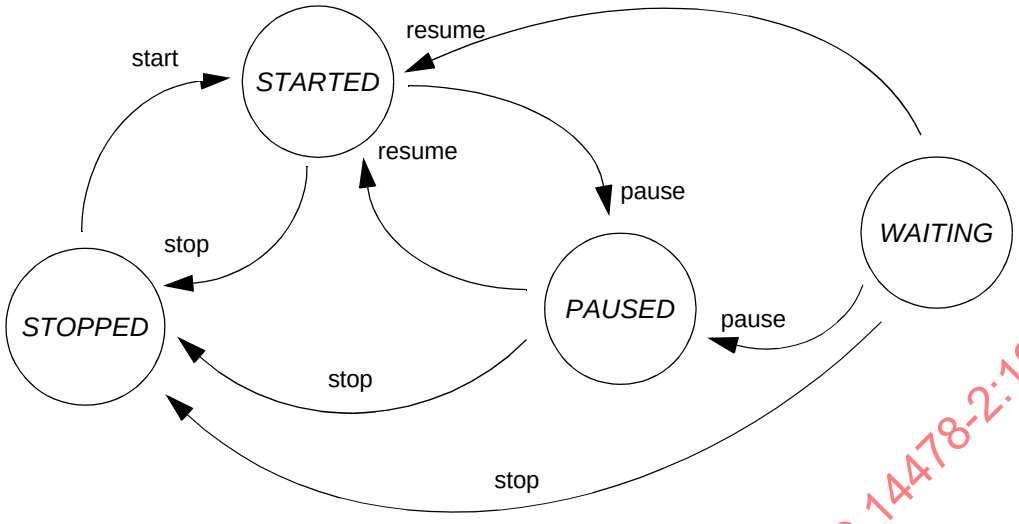


Figure 4 — State transition diagram for a *Synchronizable* object

7.9.1.1 Semantic specification of states

7.9.1.1.1 State *STARTED*

If the object's state is *STARTED*, the object carries on its internal processing in a loop of processing stages. Each stage consists of the following steps:

a) The value of the current position is advanced using the (protected) operation *progressPosition* (defined as part of the object's specification), which returns the required next position.

b) This required position is compared with the current position and the end position, and the following actions are performed if the value of *currentDirection* is *Forward*:

1) If there are reference points lying between the current position and the required position (including a possible reference point on the required position itself), then all associated synchronization actions are performed (in the order in which they are defined on *C*) in a loop. This means:

- Call the (protected) *processData* operation (defined as part of the object's specification) to perform data presentation. Arguments for this call are the current position or the previous reference point and the next reference point or the end point; data represented by this interval will be presented by *processData*.

- Invoke the *callback* operation on the *Callback* object, whose description is stored in the reference point, using the stored event as an argument (if the object reference is *NULLObject*, no operation invocation takes place at this point).

- If the `wait` flag stored in the synchronization element belonging to the reference point is set to *TRUE*, the current position is set to the reference point, and the object's state is changed to *WAITING*. If the state of the object is set back, eventually, to *STARTED*, the processing stage continues at this point.

2) If there are no reference points between the current position and the required position, perform data presentation for any data identified by the points on the progression space between the current position and the required position.

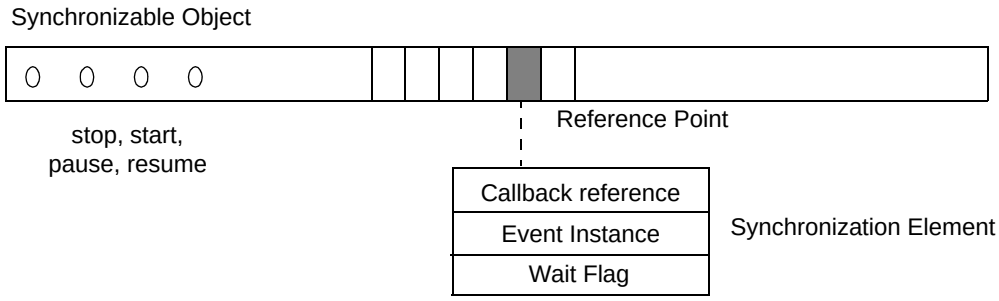


Figure 5 — *Synchronizable* object

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

4) If the required position is greater or equal to the end position, the current position is set to the start position, and the following occur:

- if the *repeatFlag* is *TRUE* the processing stage is finished (i.e., the processing continues at step “a” above);

- if the *repeatFlag* is *FALSE*, but the loop counter value is greater than 1, the loop counter value is decremented, and the processing stage is finished (i.e., the loop counter value represents the number of times the object has to play the defined span);

- otherwise, the complete loop of processing stages is finished, and the object’s state is changed to *STOPPED*.

If the value of *currentDirection* is *Backward*, the difference in behaviour is that, in 4, the role of the start and end position is reversed and, if the start position is reached, the current position is automatically set to the end position instead. Also, in 1, the direction of synchronization actions and data presentation is reversed. The specification of *progressPosition* is such that decrementing values are generated if the value of *currentDirection* is *Backward*.

Each processing stage, as described above, is “atomic”, meaning that no operation requests are accepted by the object while in *STARTED* state and performing any of the steps of the stage. In other words, servicing all other requests will be delayed (and the issuer of the request suspended), and serviced only when a full stage is finished, and before the next is started, i.e., before the next invocation of *progressPosition*.

Note that two aspects of this specification are left unspecified in the definition of *Synchronizable*:

— what “data presentation” exactly means (i.e., the detailed semantics of *processData*), and

— what “progression” exactly means, (i.e., the detailed semantics of *progressPosition*).

Both these aspects shall be specified in the appropriate subtypes of *Synchronizable*.

7.9.1.1.2 States *PAUSED* and *WAITING*

If the object state is in *PAUSED* or *WAITING*, only a limited subset of operation requests are accepted by the object. These include:

— *retrieve* operation for the attributes defined for the *Synchronizable* object; see 9.11 (page 47) for the detailed specification of these attributes;

— *resume* and *stop*, resulting in state transitions.

Some attributes as well as the synchronization elements may also be set; see 9.11 (page 47) for the detailed specification of these operations.

If the state of the object is changed from *PAUSED* to *STARTED*, a new processing stage (described in 7.9.1.1.1) is started. If the state of the object is changed from *WAITING* to *STARTED*, the processing stage continues where it was interrupted (see item 1 in 7.9.1.1.1 above), i.e., no new invocation of the operation *progressPosition* occurs in this case.

7.9.1.1.3 State *STOPPED*

If the object state is in *STOPPED*, only a limited subset of operation requests are accepted by the object. These are:

— *retrieve* operation for the attributes defined for the *Synchronizable* object; see 9.11 (page 47) for the detailed specification of these attributes;

— *start* and *pause*, resulting in state transitions.

Some attributes as well as synchronization elements may also be set when in *STOPPED* state; see 9.11 (page 47) for the detailed specification of the operations.

Any transition to *STOPPED* means resetting all the attributes to their default values (i.e., start and end positions are set to their default values, as defined in the subtypes of *Synchronizable*, the current position is set to the start position, the repeat flag is set to *FALSE*, and both the values of *nloop* and of *loopCounter* are set to 1. If the state of the object is changed from *STOPPED* to *STARTED*, a new infinite loop of processing stages (described in 7.9.1.1.1) is started.

7.9.1.2 Monitoring state transitions

External clients of a synchronizable object may also require to be notified when a state transition occurs. To achieve this, action element structures (see 9.5.4) can be associated with each pair of valid states. Whenever a state transition of the *Synchronizable* object occurs, and an action element is associated with the relevant pair of states, a new event instance is constructed by the *Synchronizable* object using the event name tag of the action element, and the pair of states as event data with the key "*Transition*". The callback, appearing within the action element, is invoked with this newly created event instance as input argument before the state transition effectively occurs.

7.9.2 Time synchronizable objects

A *TimeSynchronizable* object type is a *Synchronizable* object type enriched with a *Timer* interface (see 7.8.3) through multiple subtyping (see also Figure 6 on page 17).

Multiple subtyping means that the behaviour of *Timer* and *Synchronizable* objects are merged. This merge has several aspects, and introduces some new attributes and operations on *TimeSynchronizable*. These aspects are as follows.

a) Both the *Timer* and the *Synchronizable* object type are defined in terms of finite state machines. In *TimeSynchronizable*, these finite state machines are merged, using the following state identifications:

- *TSTARTED* is merged with *STARTED*;
- *TSTOPPED* is merged with *STOPPED*;
- *TPAUSED* is merged with *PAUSED*.

Merging means that the finite state machines governing *TimeSynchronizable* has the same states as *Synchronizable*, but the semantics of each of these states includes the semantics of both the *Timer* and the *Synchronizable*. The state transition operations defined both in *Timer* and in *Synchronizable* are inherited by the *TimeSynchronizable* object and result in the appropriate state transitions.

b) An attribute is defined, called *speed*, which relates progress through the progression space, inherited from *Synchronizable*, with time as measured by the *Timer*. The value of *speed* defines the number of units (e.g., number of frames) that the

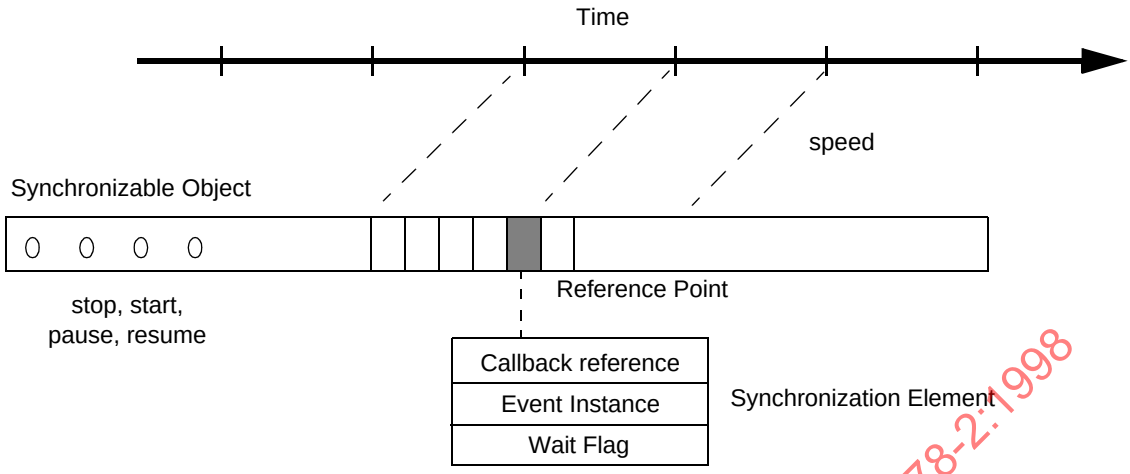


Figure 6 — *TimeSynchronizable* object

object will progress through in one tick. By default, this value may be set by the application, which can therefore control, e.g., the playback speed. Subtypes of *TimeSynchronizable* objects may restrict the behaviour so that the speed becomes retrieve–only.

c) *Synchronizable* has a number of attributes and operations to set/retrieve reference points, set/retrieve minimum and maximum positions, etc. which are expressed in terms of the native progression space. When using *TimeSynchronizable* the client may want to use the abstraction offered by the notion of relative time, i.e., the time returned by the *inquireTick* operation. For that purpose, the *reset* operation (inherited from *Timer*) is redefined in *TimeSynchronizable* to, conceptually, put a marker against the current position on the native progression space as well as to reset the time register. This marked position on the progression space will serve as zero point for relative positioning expressed with the time values. This marker, together with *speed*, defines a linear transformation between the progression space and time. *TimeSynchronizable* includes two operations, *timeToSpace* and *spaceToTime*, which transform the two coordinate spaces.

d) The operations and attributes defined for *Synchronizable* to set, retrieve, inspect, etc., reference points have also an alternative version in *TimeSynchronizable* which take relative time as input argument rather than values on the progression space. Using the current linear transformation between the two spaces, reference points defined in time are also considered by the main processing loop of *TimeSynchronizable* (see page 14). Note, however, that reference points defined in terms of time are, conceptually, also stored in terms of time. This means that if the linear transformation changes (through the invocation of the *reset* operation or by changing the speed), all reference points defined in terms of time are re–transformed. This may result in some of these reference points being used again or, conversely, never being used.

Different *TimeSynchronizable* objects may have different *accuracy* attribute values; this value may become larger (i.e., the accuracy of timing may become worse) than for other *TimeSynchronizable* or *Timer* objects if the implementation of the *TimeSynchronizable* object is not able to perform in its execution environment at a higher precision.

7.9.3 Time slave objects

A *TimeSlave* object is a subtype of *TimeSynchronizable* which permits synchronization over multiple *TimeSynchronizable* object instances. A *master* can be attached to a *TimeSlave* object, and the latter will attempt to synchronize its progression with its master. This means the following (see also Figure 7):

a) The *speed* value of the *TimeSlave* object (relating the progress through progression space with time ticks) is measured in terms of the ticks as returned by the master. This also means that if the client changes the way the master *Timer* operates (i.e, changing the ticks) this will influence all *TimeSlave* objects attached to the same master.

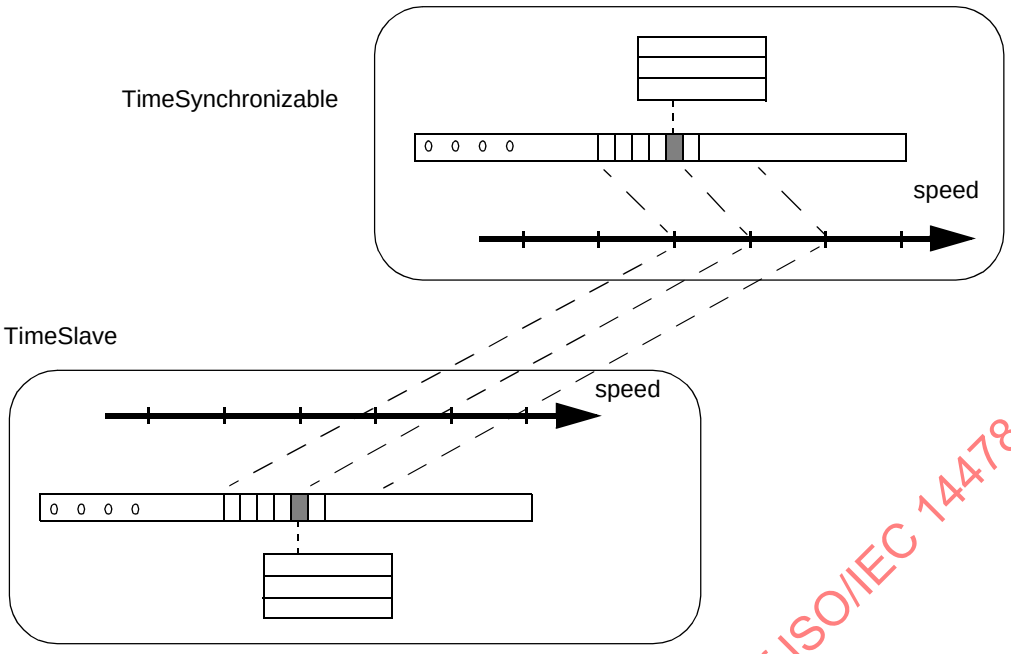


Figure 7 — *TimeSlave* object

b) The *TimeSlave* object measures the alignment between its own *Timer* values and the one of the master. A client of the *TimeSlave* object may inquire the alignment, and may attach *Callback*-s to various thresholds values. The *TimeSlave* object will raise specific events if the alignment between the master and the slave time values exceeds the threshold.

In order to calculate the possible alignment between the master and the slave time values, the *reset* operation of *TimeSlave* also stores all necessary information on the master clock (current value of tick, accuracy, units of measurement). Alignment values are always referred to in the units of *TimeSlave*. Using these terms, the alignment value is:

$$\left| Tick_{slave} - g(Tick_{master}) \right|$$

where *g()* is a function which transforms the ticks of the master into the units of the slave, and takes into account the tick value of the master when the *reset* operation has been invoked on *TimeSlave*.

The events, raised by *TimeSlave* objects when thresholds are exceeded, have the following structure tags: event name is "*OutOf-Sync*", event data contains one key–value pair, using "*Discrepancy*" as key and the actual alignment as a (float) value. By default, no events are raised, i.e., the client has to set the threshold values and the corresponding callback references through an explicit *setSyncEventHandlers* operation request.

7.9.4 Time line objects

The *TimeLine* object is a subtype of *TimeSynchronizable*, where the progression space is defined to be $Time_{\infty}$ and the value of *speed* is set to be of constant value 1. This object can be used to send events at predefined moments in time, or periodically, to dedicated PREMO objects, and may thereby serve as a basic tool for time–based synchronization patterns.

8 Enhanced property management and factories

This clause contains the specification of various object types which form an extension of the fundamental objects as described in clause 7. Because not all components or PREMO applications may need the additional complexity of these extensions, objects in this clause form an extended profile of the foundation component of PREMO (see clause 10 for the detailed component and profile specification of the Foundation Component of PREMO). Objects in this clause fall into two categories:

a) enhanced property management;

b) object factories and factory finders, which give a finer control over the life cycle of PREMO objects.

8.1 Enhanced Property management

8.1.1 Motivation

All objects in this clause are subtypes of *EnhancedPREMOObject*. As such, they also inherit the property operations defined in 7.5.1 (see also 9.7). These *properties* refine the definition of the objects and their behaviour beyond that defined by their type (i.e., the operations in their interfaces). Some properties are common to many PREMO objects, and others are particular to the object type to which the object belongs.

Properties, and the various property management operations described in this clause, are the basic building blocks for various configuration and negotiation mechanisms in PREMO. Such negotiations may be necessary to have, e.g., an optimal control over media flow, to control the quality of service of various multimedia devices, to ensure proper coding and decoding of media data when necessary, etc. As a general principle, the parameters governing the behaviour of objects are described in terms of properties, rather than attributes, if they may be subject to dynamic negotiations.

As a notational convenience, most property names defined as part of the functional specification of objects in this clause end with the character "*K*". Also as a notational convenience, the informal term "property space" will also be used to refer to the various properties available for an object (property keys playing the role of naming the "coordinate axes" in this "space", and values playing the role of points on these axes).

Various other PREMO components rely on a further refinement of the usage of properties, embodied by the type *PropertyInquiry* and its subtype *PropertyConstraint*. These types will be defined in details in 8.1.2 and 8.1.3 below; this clause gives only an overview and a motivation for the further refinement of property management.

Figure 8 on page 20 gives a schematic view of the notions involved. The figure represents the range of values belonging to *one* property key. The *capability* associated with this key describes the possible range of values which may belong to this key. This is a read–only information which belongs to a specific *type*. An instance of this type may have a *native property value* for this key, which describes the possible range of values this *instance* can associate to this key. Obviously, the native property value represents a subset of the capability. Capabilities and native property values give a dynamically accessible information on the possible behaviour of an object instance, which can be used in negotiations procedures. A *PropertyInquiry* type stipulates that it is always possible to retrieve the native property values for a property key for all properties explicitly defined as part of the functional specification of the object within PREMO. In other words, although the actual values of the property may be changed through the invocation of the various property management operations, it is always possible to access the native property values, too.

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

NOTE — For example, an audio object type may be defined in terms of the sample rate it can process. The possible values for the audio sample rates are characterized by its capability, e.g., *<8KHz, 11.3KHz, 22.05KHZ, 44.1KHz>*. When instantiating such an object, the object may find out that on the hardware environment it is running it cannot honour, say, the 44.1KHz range. Consequently, the corresponding native property value for this instance and this key will be *<8KHz, 11.3KHz, 22.05KHz>*. As a next step, a client may constrain the acceptable range

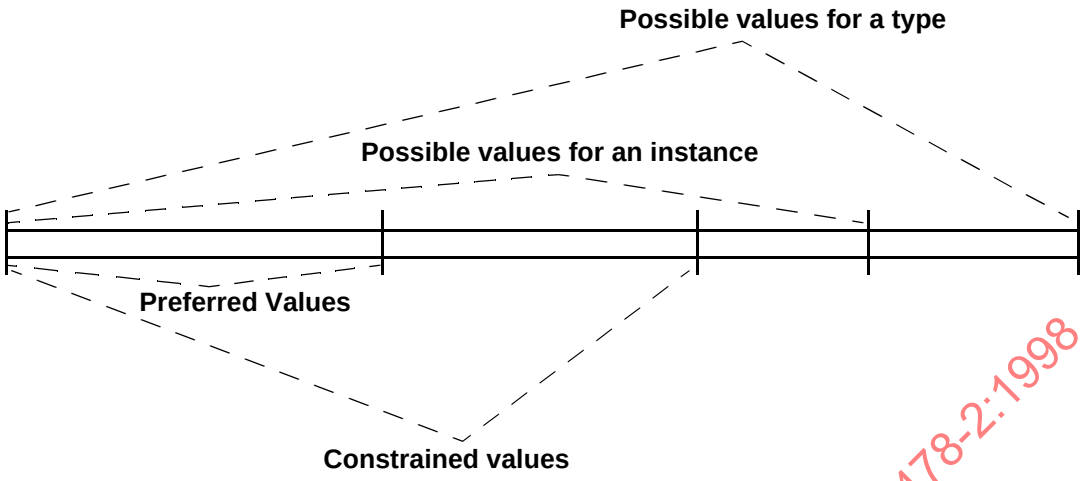


Figure 8 — Type properties, capabilities, constraining properties

of sample rates for this instance by requesting that the sample rate should be either 11.3KHz or 22.05KHz; i.e., the current property value for the key becomes *<11.3KHz, 22.05KHz>*. Finally, as a result of a subsequent *select* operation invocation, the audio object may decide to choose 22.05KHz as a sample rate to increase the quality of the output. Note that the *<8KHz, 11.3KHz, 22.05KHz>* range (i.e., the native property value) can always be inquired by a client.

8.1.2 Capabilities and native property values: the *PropertyInquiry* type

A *capability* of an object is a special type of read only property that describes the *value* or *values* another property of that object *type* may take on. Capabilities are defined as part of the type specification of the object, i.e., the information they provide is the same for all instances of that type. Capabilities, like other properties, are specified as key/value pairs, where the key identifies the characteristic of interest, and the value is of the general data type *Value*, as defined in 9.2 (note that a *Value* may also refer to a sequence of other values). As a notational convenience, the symbolic name for a type property ends with a "*CK*" and the remainder of the name is the same as that of the corresponding property.

NOTE — For example, the *InputEncodingCK* property has a set of values that represent the range of possible values that the *InputEncodingK* property can assume for the specific type.

A *PropertyInquiry* is an object type for which a number of capabilities are also defined. Furthermore, such an object also stores its *native property values* for all keys which are explicitly defined as part of the object specification; the native property value is defined to be the range of values (associated to a key) that object *instance* may take on. An operation, called *inquireNativePropertyValue*, is defined for the *PropertyInquiry* object type, which always returns the native property value, regardless of the current values associated to the key. This operation returns a sequence of values (e.g., if the type of the corresponding property is defined as a *String*), or a minimum–maximum range (if the value is a numerical type). The specification of the property shall define the return type of the result of operation invocation if this is not the case.

The type of the value that can be associated with a particular property is specified along with the key definition, as part of the object's specification. When querying the capabilities of a property of an object, the value associated with a key may be a single datum, a sequence, or a range.

NOTE — For example, an audio capture device might report a variety of properties (e.g., through the *getAllProperties* operation defined for *EnhancedPREMOObject* type), some of which are limited to only a single value, and some of which might take a range or sequence of values. After initialization, the values associated to the various keys may be as follows:

Key	Value
AudioDevice::InternetLocationCK	*<"mymachine.com","yourmachine.com","theirmachine.edu">*
AudioDevice::InternetLocationK	*"mymachine.com"*
AudioDevice::InputEncodingCK	*<"alaw", "ulaw", "linear">*
AudioDevice::InputEncodingK	*<"alaw", "ulaw", "linear">*

Here the location of the device instance is a single value (where it is wanted to be), although the object might have been instantiated on a range of possible machines, whereas the encoding that can be used at the input can take on several values.

The client can discover an object's type properties and its capabilities, as well as the current settings of its properties, using the operation

getPairs

$pairs_{out}$: seq (*Key* × seq *Value*)

If the client is interested only in the value or values a particular property takes, the following operation can be used:

getProperty

key_{in}: *Key*
$value_{out}$: seq *Value*

Both of these operations are defined for all *EnhancedPREMOObject* types (see 7.5.1).

The detailed functional specification of PREMO objects may contain capability specifications, too. This means that the corresponding property keys are automatically defined for these objects at object creation time, together with the values the functional specification contains. Subtypes may extend the possible values for a capability.

8.1.2.1 Required properties

PREMO requires all objects of type *PropertyInquiry* to provide three property values (i.e., these keys are defined for the object type *PropertyInquiry*):

a) Network location (key "*LocationK*")

b) Vendor tag (key "*VendorTagK*")

c) Release (key "*ReleaseK*")

All these properties have strings as possible values; the content and the interpretation of these values are implementation dependent.

8.1.3 Property constraint and selection: the *PropertyConstraint* type

PropertyConstraint is a subtype of *PropertyInquiry* which offers two more operations to manipulate properties: *constrain* and *select*.

The *constrain* operation receives a sequence of key–value pairs (each value may refer to a sequence of other values) as a parameter and tries to set the values for each of the keys involved. The new value for a specific key is as follows:

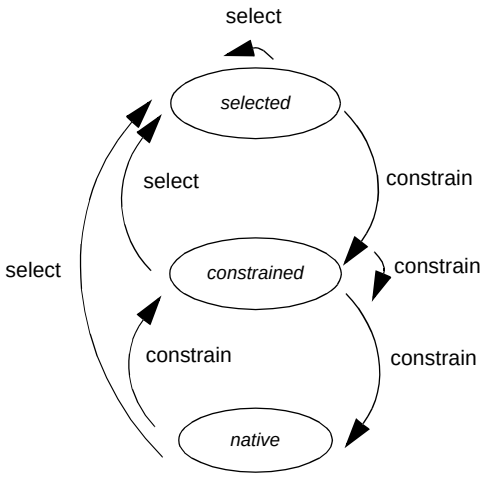


Figure 9 — *PropertyConstraint* state transition diagram

— if no native property value is defined for the key, then:

- if no value is associated to the key yet, the new value is the value appearing in the input parameter; else
- the new value is the intersection of the values appearing in the argument with the current values associated to the key.

— if a native property value is defined for the key, then:

- if no value is associated to the key yet, the new value is the sequence of values appearing in the input parameter, intersected with the native property values; else
- the new value is the intersection of the sequence of values appearing in the argument, the current values associated to the key, and the native property values.

The operation returns the new set of values set for each key.

The client can always return to the native property values. Indeed, the client can inquire the native property values, clear the associated values with an empty sequence, and use the native property values to restore the current value for the property key.

The *select* operation is used to determine the best–fit values based upon the intersection of a sequence of key–value pairs appearing as input parameter and the current values assigned to the key. To the extent that multiple values are feasible, the *PropertyConstraint* object selects values for the client. The selected key–value pairs are returned as the operation's output.

Figure 9 gives an overview of the state transitions related to the properties associated to one key. The initial state is the native property values known to the object (if defined). The values can be inquired by the various inquiry and the *matchProperties* operation (see 7.5.1) but these operations do not change the property values. Note that the *constrain* operation can be issued at all states.

8.1.3.1 Properties and object behaviour

The behaviour of the *PropertyConstraint* object may depend on the properties associated with the object. This sub–clause defines these relationships.

8.1.3.1.1 Private properties

The *PropertyConstraint* object is a subtype of *EnhancedPREMOObject*, hence it also allows clients to add properties beyond those which are defined as part of the object's functional specification, i.e., the client can extend the property space with private properties (e.g., for annotations). The client can annotate the space with private properties.

The semantics of the *constraint* and *select* operations are such that they ignore these private properties.

8.1.3.1.2 Interactions among properties

Considering the properties of an object one at a time, an object may appear to have the possibility to assume all possible combinations of properties. When the properties are considered in combination, only certain combinations may be possible.

NOTE — To illustrate this, consider a fictitious audio value space. Two properties — one with the key *SampleSize* and the other with the key *SampleRate* — describe the object. The sample size can be 8bit or 16bit, while sample rate can be 8KHz or 40KHz. If all combinations of properties are possible, then the possible options are shown in Table 1:

	Sr=8KHz	**Sr=40KHz**
Ss=8bit	Sz=8bit,Sr=8KHz	Sz=8bit,Sr=40KHz
Ss=16bit	Ss=16bit,Sr=8KHz	Ss=16bit,Sr=40KHz

Table 1 — All combinations of sample rate and sample size

The complication is that, in practice, media objects abstract real media devices. These media devices often allow only restricted combinations of property values. The audio device, for example, could support the specific values in Table 1 only.

	Sr=8KHz	**Sr=40KHz**
Ss=8bit	Sz=8bit,Sr=8KHz	
Ss=16bit		Ss=16bit,Sr=40KHz

Table 2 — Restricted combinations of sample rate and sample size

For this purpose, an additional property is defined that allows the precise values of such combinations to be expressed. The *PropertyConstraint* type provides a *ValueSpaceNameK* property for this purpose. If the value of the property is empty, then all combinations are allowed. If the value is not empty, then it is a sequence of key–value sequences which describe the allowable combinations.

NOTE — Following the example above, the value for the *ValueSpaceNameK* for the fictitious audio object may then include the following two sequences:

<<"SampleSize",8>,<"SampleRate",8>>,<<"SampleSize",16>,<"SampleRate",40>>

The operation *bind* (see also 8.1.3.1.3 below) checks the consistency of the properties against the values stored in *ValueSpaceNameK*. If the value in *ValueSpaceNameK* for a specific key is a sequence and the corresponding property value is not, then the value is checked against the elements of the sequence (i.e., the *ValueSpaceNameK* property describes all allowable values in a combination); otherwise equality is used. Also, if a key does not appear in a specific sequence, than all values are permissible. An exception is raised if the values are incorrectly set.

NOTE — For example, if the value for the *ValueSpaceNameK* is:

<<“Key1”,<“A”,”B”,”C”>>,<“Key2”,”P”>>, <<“Key1”,<“X”,”Y”,”Z”>>>

then, for example, the following key–value combination is accepted by *bind*:

<“Key1”,“A”>,<“Key2”,”P”>

because, by virtue of the first rule cited above, the first constraint pair of *ValueSpaceNameK* is used to check a subset; the combination

<“Key1”,“X”>,<“Key2”,”W”>

is also accepted because, by virtue of the second rule, the second constraint pair of *ValueSpaceNameK* allows for an unconstrained setting of the value for *Key2*, provided that the value for *Key1* is one of *X*, *Y*, or *Z*.

8.1.3.1.3 Dynamic changes to properties

During the lifetime of a *PropertyConstraint*, some of the properties may become static or changeable. For an example where the properties become static, consider the example in the previous sub–clause: the sample size and the sample rate should not change while the media stream flows, i.e., neither the client nor the object itself should change these values in this period. Changeable properties may again fall into two categories: mutable or dynamic. Mutable properties are such that no client should be able to change these values when the media stream flows, although the object itself may change them. Such values might, for example, change as a result of decoding a protocol found within the media stream which defines the mutable a property value (e.g., the quantization matrix of an MPEG flow). Finally, there may be properties which may be changed at any time; these are referred to as dynamic properties.

The *PropertyConstraint* type anticipates these situations. On the one hand, two operations are defined on the *PropertyConstraint* object type, namely *bind* and *unbind*, which determine the interval in the life cycle of the object when the values associated to certain keys cannot be changed. On the other hand, the type reserves some keys and capabilities which characterize the static or changeable nature of other properties. These are as follows:

— The key *MutablePropertyListK* identifies the public properties which can be changed by the object itself at any time, but the client may not change these values between a *bind* and an *unbind* call. The value for this property is a sequence of property keys.

— The capability *MutablePropertyListCK* is defined for various subtypes to describe the values the property *MutablePropertyListK* can take. Various instances may have a more restrictive native property value for *MutablePropertyListK* (see 8.1.2).

— The key *DynamicPropertyListK* identifies the public properties which can be changed at any time. The value for this property is a sequence of property keys.

— The capability *DynamicPropertyListCK* is defined for various subtypes to describe the values the property *DynamicPropertyListK* can take. Various instances may have a more restrictive native property value for *DynamicPropertyListK* (see 8.1.2).

All properties, whose keys are not listed in either *MutablePropertyListK* or *DynamicPropertyListK*, are defined to be static, i.e., their values can be changed neither by a client nor by the object itself between a *bind* and an *unbind* call.

8.2 Creating PREMO objects

The general object and object reference life cycle facilities of a PREMO environment are defined in 8.11 of ISO/IEC 14478-1. These facilities allow clients to create and destroy the objects required to perform the media functions and services required. This sub–clause defines some other objects which allow clients to create objects subject to certain constraints, expressed in terms of key–value pairs.

8.2.1 Generic Factory objects

The purpose of the generic factory object is to provide a wrapper around the object creation facilities, but taking a list of properties into consideration, too, when creating an object. These properties describe the required characteristics of the object to be created.

The factory object has only one operation, *createObject*, which takes an object type name and a list of key–value pairs as input arguments, and returns an object reference if an object can be created or found, or raises an exception, if the requirements cannot be met. The constraint imposed by the factory object is that the *native property value of the returned object, corresponding to a key in the list, should be a superset of the values in the argument list.* The signature of the operation is:

createObject

$type_{in}$: *ObjectType*
$constraints_{in}$: seq (*Key* × seq *Value*)
$initArg_{in}$: *Value*
$objectRef_{out}$: *RefPropertyInquiry*
exceptions: {*InvalidCapabilities*, *CannotMeetCapabilities*, *InvalidType*}

Note that the return value is of type *RefPropertyInquiry*; this reference has to be cast to a reference to the desired type, using the `cast` facility (see 8.11 of ISO/IEC 14478-1). Also, the properties of the returned object are not set by the factory; this is to be done by the client (using, for example, a subsequent invocation of the *constrain* operation). Because the factory uses the notion of native property values, the type of the object to be created shall be a subtype of *PropertyInquiry*. The factory object is itself defined to be a subtype of *PropertyInquiry*, too. This means that factory objects may also have properties and these properties can be inquired and/or set (e.g., using the operations described in 8.1.3 of this part), and that factories may also create other factories.

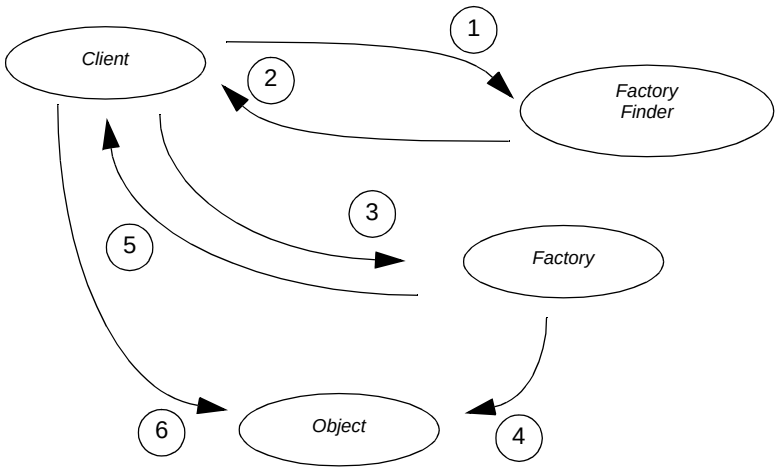
The specification of the generic factory does *not* require that a new instance of the object, corresponding to the type and the constraints, shall be created. In some cases, the factory may just return a new object reference to an already existing object (although subtypes of the factory object may impose additional restrictions in this respect). In the case a new object is created, the value of $initArg_{in}$ is used as an argument of the new object's initialize operation (see also 7.2)

8.2.1.1 Finding factories

The purpose of the factory finder object is to locate factory objects, using an object type and a set of capabilities to restrict the set of possible choices. The object provides one operation, whose signature is:

findFactories

$type_{in}$: *ObjectType*
$objectConstraints_{in}$: seq (*Key* × seq *Value*)
$factoryConstraints_{in}$: seq (*Key* × seq *Value*)
$factories_{out}$: seq *RefGenericFactory*
exceptions: {*NoFactory*, *InvalidCapabilities*, *CannotMeetCapabilities*}



1. *Client requests a reference to a factory capable of satisfying a capability list passed as parameter.*
2. *Factory Finder returns a Factory reference*
3. *Client requests the creation of an object from the Factory, with a constraint list on the object to be created.*
4. *Factory possibly creates the object…*
5. *…and returns the reference to its client.*
6. *Later, the client destroys the reference, which eventually leads to the destruction of the object.*

Figure 10 — PREMO Object life cycle with factories

This operation returns a sequence of references to *GenericFactory* objects, all capable of creating an object of type $type_{in}$, and matching the constraints described in $objectConstraints_{in}$. The $objectConstraints_{in}$ input argument is a sequence of key–value pairs, describing what sort of factory is sought in terms of the properties of the objects it is capable of creating. An additional sequence of constraints, $factoryConstraints_{in}$, refers to the generic factory objects themselves (as opposed to the constraints referring to the object which can be created).

NOTE — For example, the factory finder object may be used to locate factory objects which can create video objects capable of processing MHEG files (the $objectConstraints_{in}$ can be used to describe this fact) and which are on a restricted internet location (a constraint on the location of the factories themselves can be expressed by the $factoryConstraints_{in}$).

Figure 10 gives an overview of the life cycle of a PREMO object when using factories and factory finders.

8.2.1.2 Persistency of factories and factory finders

Because factories are used to create objects in which the caller is interested, some of them may be persistent objects. Also, at least one factory finder object shall be persistent. This means that PREMO makes the assumption that some factory finder objects, as well as some factory objects, are automatically created by the PREMO environment when a PREMO system is started and their object references are known. Details of how this is done is implementation and environment dependent.

8.2.1.3 Implementation of Factories

PREMO does not dictate how factories are implemented. A number of different implementations are possible. These include:

a) *A single factory for all of PREMO:* A single executable could provide all the client services including the object and object reference life cycle facilities, the factory, all the virtual devices and connections, etc.

b) *A single factory per node in a network:* Each node in a network could have a single factory that is responsible for the management of the virtual devices and connections on that machine. The factory could implement the virtual devices and connections in its own address space or spawn processes for that purpose.

c) *A factory per object type:* There could be a factory for each type of device and connection that can be created. These could be network wide, or per machine.

There are other possibilities and combinations. All of the implementation choices are compliant if they adhere to the type definitions.

9 Functional specification

9.1 Introduction

This clause provides the detailed functional specification of the non–object types, structures, and other PREMO object types that together define the PREMO Foundation Component. The notation used in this clause follows the rules detailed in annex A of ISO/IEC 14478-1.

Additionally to the object type definition, each PREMO type may have predefined set of properties and/or capabilities. These are defined in separate tables following the type specification schema. These tables include the name of the key, the type of the value, a flag whether the property is read only (R.O.) or not (R/W), and possibly a short description of the property. Capabilities are defined in a separate table.

9.2 Common non–object data types

This sub–clause defines all data types used by structure tags and operations defined on PREMO foundation object types.

Boolean values:

Boolean ::= *TRUE* | *FALSE*

Used to describe type graphs and/or sequence of immediate supertypes:

TypeGraph == seq *ObjectType*

Keys for key–value pairs, and key–value sequence pairs:

Key == *String*

Action types in controllers:

ActionType ::= *Enter* | *Leave*

Types for event identifications:

EventName == *String*

Enumerations controlling comparisons in key-value pairs, used, e.g., by the property operations or the event handler:

AndOr ::= *And* | *Or*

ConstraintOp ::= *Equal* | *NotEqual*
| *GreaterThan* | *GreaterThanOrEqual* | *LessThan* | *LessThanOrEqual*
| *Prefix* | *Suffix* | *NotPrefix* | *NotSuffix*
| *Includes* | *Excludes*

Units of time, used by clock objects.

TimeUnit ::= *Picoseconds* | *Nanoseconds* | *Microseconds* | *Milliseconds*
| *Second* | *Minute* | *Hour* | *Day* | *Month* | *Year*

Type synonym for the description of states:

$State == \mathbf{N}$

Constants identifying the states of a *Timer* object:

$$TSTOPPED : State \mid TSTOPPED = 0$$
$$TSTARTED : State \mid TSTARTED = 1$$
$$TPAUSED : State \mid TPAUSED = 2$$

Data types and state constants used for synchronizable objects:

$$Direction ::= Forward \mid Backward$$
$$STOPPED : State \mid STOPPED = 0$$
$$STARTED : State \mid STARTED = 1$$
$$PAUSED : State \mid PAUSED = 2$$
$$WAITING : State \mid WAITING = 3$$

The following discriminated union is used, e.g., when the argument type of an operation is not fully specified (the full specification being part of the semantics of the type), or as part of properties and key–value pairs:

$$\begin{aligned} Value ::= & \texttt{uInt}\langle\!\langle \mathbf{N} \rangle\!\rangle \\ & \mid \texttt{int}\langle\!\langle \mathbf{Z} \rangle\!\rangle \\ & \mid \texttt{real}\langle\!\langle \mathbf{R} \rangle\!\rangle \\ & \mid \texttt{objectType}\langle\!\langle ObjectType \rangle\!\rangle \\ & \mid \texttt{actionType}\langle\!\langle ActionType \rangle\!\rangle \\ & \mid \texttt{eventId}\langle\!\langle EventId \rangle\!\rangle \\ & \mid \texttt{time}\langle\!\langle Time \rangle\!\rangle \\ & \mid \texttt{premoObject}\langle\!\langle RefPREMOObject \rangle\!\rangle \\ & \mid \texttt{simplePremoObject}\langle\!\langle RefSimplePREMOObject \rangle\!\rangle \\ & \mid \texttt{callbackPremoObject}\langle\!\langle RefCallback \rangle\!\rangle \\ & \mid \texttt{enhancedPremoObject}\langle\!\langle RefEnhancedPREMOObject \rangle\!\rangle \\ & \mid \texttt{extendedReal}\langle\!\langle \mathbf{R}_{\infty} \rangle\!\rangle \\ & \mid \texttt{extendedInteger}\langle\!\langle \mathbf{Z}_{\infty} \rangle\!\rangle \\ & \mid \texttt{extendedTime}\langle\!\langle Time_{\infty} \rangle\!\rangle \\ & \mid \texttt{boolean}\langle\!\langle Boolean \rangle\!\rangle \\ & \mid \texttt{stringValue}\langle\!\langle String \rangle\!\rangle \\ & \mid \texttt{andOr}\langle\!\langle AndOr \rangle\!\rangle \\ & \mid \texttt{constraintOp}\langle\!\langle ConstraintOp \rangle\!\rangle \\ & \mid \texttt{timeUnit}\langle\!\langle TimeUnit \rangle\!\rangle \\ & \mid \texttt{direction}\langle\!\langle Direction \rangle\!\rangle \\ & \mid \texttt{valueSequence}\langle\!\langle \text{seq } Value \rangle\!\rangle \\ & \mid \texttt{valuesTuple}\langle\!\langle Value \times Value \rangle\!\rangle \end{aligned}$$

Note that

— The definition of *Value* is recursive in the sense that it also contains tags to a sequence or a tuple of *Value;*

— Three object references are part of *Value*, which allow for operations to refer either to any PREMO object, or to restrict to either simple or enhanced PREMO objects.

9.3 Exceptions

Exceptions are defined in PREMO as a data tuple:

$$Exception == String \times \text{seq } Value$$

As a convention, the first tag of the exception is referred to as the name of the exception. Operations may or may not assign data to the second tag of the exception when raising it. Details of how exceptions are raised is not defined in PREMO (see also 8.12 of ISO/IEC 14478-1).

The PREMO foundation component defines a number of exceptions, and further components may add their own exceptions. As a convention, exceptions in PREMO are defined to have their data type name used as the exception name. For example, the exception *IncorrectInit*, defined below, has the first tag set to the string "*IncorrectInit*".

The list of the exceptions raised by operations defined on PREMO foundation object types are as follows.

CannotMeetCapabilities == *Exception*
IncorrectInit == *Exception*
InvalidCapabilities == *Exception*
InvalidElementId == *Exception*
InvalidKey == *Exception*
InvalidType == *Exception*
InvalidValue == *Exception*
InvalidReference == *Exception*
NoKey == *Exception*
NotInTypeGraph == *Exception*
OperationNotDefined == *Exception*
ReadOnlyProperty == *Exception*
RepeatedEvent == *Exception*
UnknownEvent == *Exception*
UnknownType == *Exception*
WrongState == *Exception*
WrongValue == *Exception*

9.4 *PREMOObject* and fundamental object behaviour

PREMOObject is an abstract type. It is a supertype for all object types defined in PREMO.

PREMOObject$_{abstract}$

Ξ*initialize*

initValue$_{in}$: *Value*
exceptions: {*IncorrectInit*}

This operation is invoked by the `create` facility of the PREMO environment when an object instance is created (8.11 of ISO/IEC 14478-1). This operation is usually overloaded by specific implementations on various subtypes. In subtypes, this initialization operation may also invoke the *initialize* operations of its supertypes.

Exceptions raised:

IncorrectInit	The parameters for initialization are incorrect.

Ξ*initializeOnCopy*

This operation is invoked by the `copy` facility of the PREMO environment when an object instance is copied (8.11 of ISO/IEC 14478-1). This operation is usually overloaded by specific implementations on various subtypes. In subtypes, this initialization operation may also invoke the *initializeOnCopy* operations of its supertypes.

Exceptions raised: None.

Ξ*destruct*

This operation is invoked by the PREMO environment when an object instance is destroyed. This operation is usually overloaded by specific implementations on various subtypes. In subtypes, this destruction operation may also invoke the *destruct* operations of its supertypes.

Exceptions raised: None.

inquireType

type$_{out}$: *ObjectType*

This operation returns the immediate type of the object.

Exceptions raised: None.

inquireTypeGraph

typeGraph$_{out}$: *TypeGraph*

This operation returns the type graph of the immediate type of the object.

Exceptions raised: None.

inquireImmediateSupertypes

immediateSupertypes$_{out}$*: TypeGraph*

This operation returns the set of the immediate supertypes of the object.

Exceptions raised: None.

PREMOObject

9.5 Simple PREMO object and structures

9.5.1 SimplePREMOObject

This object is a common, abstract supertype for all other object types appearing in this clause.

SimplePREMOObject$_{abstract}$

PREMOObject

SimplePREMOObject

9.5.2 *Event* structure

Event

SimplePREMOObject

eventName: String
eventData: seq (*Key* × *Value*)
eventSource: RefEnhancedPREMOObject

The value of *eventSource* is usually set to the reference of the object which has created the structure instance.

Event

9.5.3 *Constraint* structure

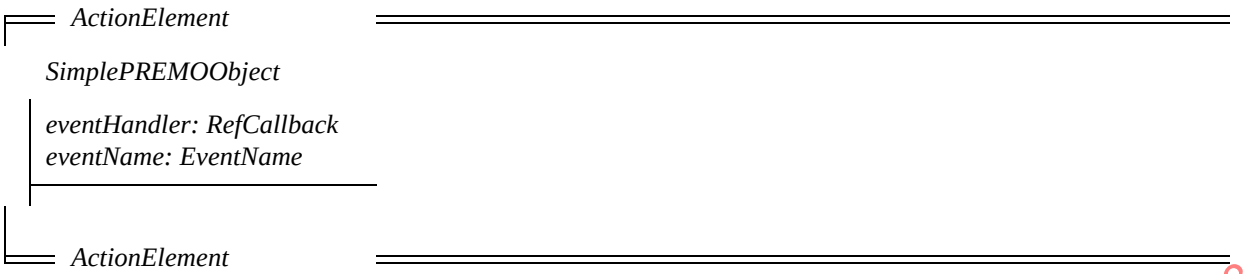
Constraint

SimplePREMOObject

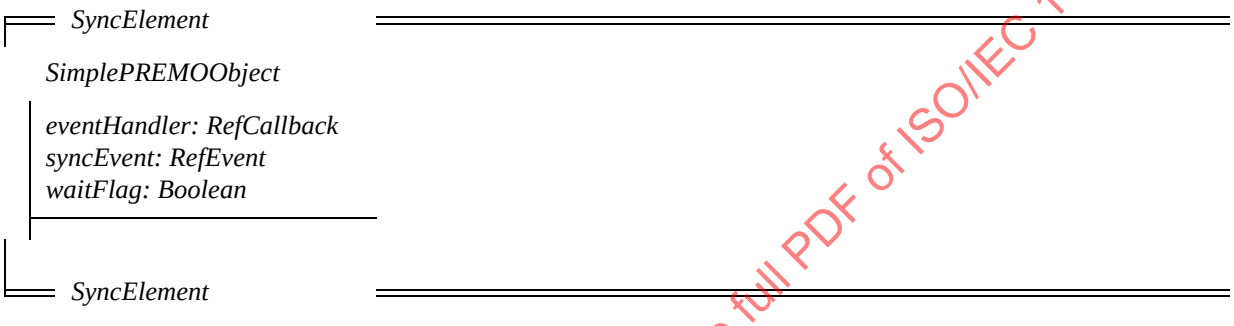
constraintOp: ConstraintOp
keyValue: Key × *Value*

Constraint

9.5.4 Action Element



9.5.5 Synchronization Element



9.6 Callback objects

Callback$_{abstract}$

PREMOObject

callback$_a$

callbackValue$_{in}$*: RefEvent* [Shallow Copy]

Subtypes of *Callback* should redefine this operation to give a more precise behaviour. Note that this operation is asynchronous.

Exceptions raised: None

Callback

CallbackByName$_{abstract}$

Callback redef *(callback)*

callback$_a$

callbackValue$_{in}$*: RefEvent* [Shallow Copy]
exceptions: {*OperationNotDefined*}

The local operation, referred to by the *eventName* tag in *callbackValue*$_{in}$, is invoked. All other structure tags are disregarded. Note that this operation is asynchronous.

Exceptions raised:

OperationNotDefined	The operation, referred to by *eventName*, is not defined on the object.

CallbackByName

9.7 Enhanced PREMO object

This object type adds the property management operations to *PREMOObject.*

EnhancedPREMOObject$_{abstract}$

PREMOObject

defineProperty

key$_{in}$*: Key*
value$_{in}$*:* seq *Value*
exceptions: {*ReadOnlyProperty*}

This operation adds a new property to the object. If the key identifies a property already defined for the object, the new value is assigned to the property replacing the previous value(s), unless *key*$_{in}$ refers to a read only property. Otherwise, a new property is created with *key*$_{in}$ and *value*$_{in}$.

Exceptions raised:

ReadOnlyProperty	*key*$_{in}$ refers to a read only property of the object.

undefineProperty

key$_{in}$*: Key*
exceptions: {*ReadOnlyProperty*, *NoKey*}

This operation removes the property identified by *key*$_{in}$, unless the property is read only.

Exceptions raised:

ReadOnlyProperty	*key*$_{in}$ refers to a read only property of the object.
NoKey	No property has been defined with *key*$_{in}$.

addValue

key$_{in}$*: Key*
value$_{in}$*: Value*
exceptions: {*ReadOnlyProperty*}

If the key identifies a property already defined for the object, and the property is not read only, the new value is added as the last element of the sequence of properties identified by *key*$_{in}$. If the key has not yet been defined for the object, a new property is created with the value of *key*$_{in}$, referring to sequence of length one and containing *value*$_{in}$.

Exceptions raised:

ReadOnlyProperty	*key*$_{in}$ refers to a read only property of the object.

removeValue

key_{in}: Key
$value_{in}$: Value
exceptions: {*ReadOnlyProperty, NoKey, InvalidValue*}

If the key identifies a property already defined for the object, and the property is not read only, and *$value_{in}$* is element of the sequence associated with *key_{in}*, the value is removed from the sequence of properties identified with *key_{in}*.

Exceptions raised:

ReadOnlyProperty	*key_{in}* refers to a read only property of the object.
NoKey	No property has been defined with *key_{in}*.
InvalidValue	*$value_{in}$* does not refer to a value associated with *key_{in}*.

inquireProperties

$keys_{out}$: seq (*Key* × *Boolean*)

This operation returns information on the properties defined for the object. The information includes the key and a boolean flag notifying whether the property is read–only or not. The order within the sequence is implementation dependent, and does not necessarily reflect the order in which properties have been defined.

Exceptions raised: None.

getProperty

key_{in}: Key
$value_{out}$: seq *Value*
exception: {*NoKey*}

This operation returns the value associated with the property identified by *key_{in}*.

Exceptions raised:

NoKey	No property has been defined with *key_{in}*.

getPairs

$pairs_{out}$: seq (*Key* × seq *Value*)

This operation returns the sequence of all properties associated with the object. The order within the sequence is implementation dependent, and does not necessarily reflect the order in which properties have been defined. (Note that *Value* structure tag in *$pairs_{out}$* may also refer to a sequence of values, see 9.2).

Exceptions raised: None.

matchProperties

constraintList$_{in}$: seq *RefConstraint* [Shallow Copy]
satisfied$_{out}$: seq (*Key* × seq *Value*)
unsatisfied$_{out}$: seq (*Key* × seq *Value*)

The properties defined for the object are matched against the property sequences in *constraintList*$_{in}$. For each key appearing in *constraintList*$_{in}$ the value is compared against the value or values stored with an identical key in the object. Comparison is based on the boolean operation defined by the non–object data type *ConstraintOp*, see 9.2 (page 27), and appearing in the corresponding structure tag of *constraintList*$_{in}$. The left operand of the operation is the property stored in the object, and the right operand of the operation is the value appearing in the *constraintList*$_{in}$ structure (if the operation does not make sense, e.g., the operands are of incompatible types, the result of the comparison is *FALSE*).
The structure *satisfied*$_{out}$ contains those keys with associated values for which the comparison has resulted in *TRUE*. The structure *unsatisfied*$_{out}$ contains those keys with associated values for which the comparison has resulted in *FALSE*.

Exceptions raised: None.

setPropertyCallback

key$_{in}$: *Key*
callback$_{in}$: *RefCallback*
eventName$_{in}$: *String*
exception: {*NoKey*}

The *eventName*$_{in}$ and *callback*$_{in}$ pair is stored, associated to *key*$_{in}$. If a new value is set for the property *key*$_{in}$, and the associated *callback*$_{in}$ is not *NULLObject*, the operation *callback*$_{in}$.*callback* is invoked. The event structure of the argument will be constructed with *eventName*$_{in}$, with a copy of the newly set key–value pair as the *eventData* structure tag.

Exceptions raised:

NoKey	No property has been defined with *key*$_{in}$.

EnhancedPREMOObject

9.8 *Controller* object

For a detailed description of the behaviour of controller objects, see 7.6 (page 8).

Controller$_{abstract}$

EnhancedPREMOObject
Callback redef *(callback)*

currentState: String	[Retrieve Only]
possibleStates: seq *String*	[Retrieve Only]

handleEvent$_a$

newEvent$_{in}$*: RefEvent*	[Shallow Copy]

Initiate state transition. See 7.4 (page 6) for a detailed specification of this operation. Note that the operation is asynchronous.

Exceptions raised: None.

callback == *handleEvent*

Ξ*checkTransition*

event$_{in}$*: RefEvent*
checkResult$_{out}$*: Boolean*

This operation checks whether the state transition, as requested by *event*$_{in}$, is possible. By default, this operation only checks whether the new state is part of the *possibleStates* sequence, in which case it returns *TRUE*; *FALSE* otherwise. Subtypes of *Controller* may add more complex checks.

Exceptions raised: None.

Ξ*onLeave*

event$_{in}$*: RefEvent*
oldState$_{in}$*: EventName*
newState$_{in}$*: EventName*
eventData$_{out}$: seq (*Key* × *Value*)

This operation is invoked before a state transition occurs from *oldState*$_{in}$ to *newState*$_{in}$. *event*$_{in}$ is the event structure which resulted in this state transition. By default, *eventDate*$_{out}$ is a copy of the event data in *event*$_{in}$; see 7.4 (page 6) for a detailed specification of how this data is used by the *Controller* object.

Exceptions raised: None.

ΞonEnter

$event_{in}$: *RefEvent*
$oldState_{in}$: *EventName*
$newState_{in}$: *EventName*
$eventData_{out}$: seq (*Key* × *Value*)

This operation is invoked after a state transition occurs from $oldState_{in}$ to $newState_{in}$. $event_{in}$ is the event structure which resulted in this state transition. By default, $eventDate_{out}$ is a copy of the event data in $event_{in}$; see 7.4 (page 6) for a detailed specification of how this data is used by the *Controller* object.

Exceptions raised: None.

ΞhandleUnknownEvent

$event_{in}$: *RefEvent*

This operation handles the case when the *Controller* object type receives an event that can not be handled in its current state. The default effect of this operation is to do nothing; subtypes of *Controller* may assign a more specific implementation to this operation.

Exceptions raised: None.

setAction

$state_{in}$: *EventName*
$action_{in}$: *RefActionElement*
$actionMode_{in}$: *ActionType*
exceptions: {*WrongState*}

An action is associated to the state $state_{in}$; see 7.4 (page 6) for a detailed specification of how this action is used by the *Controller* object.

Exceptions raised:

WrongState	$state_{in}$ does not identify a valid state for this object instance.

removeAction

$state_{in}$: *EventName*
$actionMode_{in}$: *ActionType*
exceptions: {*WrongState*}

If an action has been previously set by a *setAction* operation, it is removed.

Exceptions raised:

WrongState	$state_{in}$ does not identify a valid state for this object instance.

setActionOnPair

$stateOld_{in}$: EventName
$stateNew_{in}$: EventName
$action_{in}$: RefActionElement
exceptions: {*WrongState*}

An action is associated to the tuple $stateOld_{in} \times stateNew_{in}$; see 7.4 (page 6) for a detailed specification of how this action is used by the *Controller* object.

Exceptions raised:

WrongState	One of the states $stateOld_{in}$ or $stateNew_{in}$ does not identify a valid state for this object instance. The exception data contains the invalid state name(s).

removeActionOnPair

$stateOld_{in}$: EventName
$stateNew_{in}$: EventName
exceptions: {*WrongState*}

If an action has been previously set by a *setActionPair* operation to $stateOld_{in} \times stateNew_{in}$, it, is removed.

Exceptions raised:

WrongState	One of the states $stateOld_{in}$ or $stateNew_{in}$ does not identify a valid state for this object instance. The exception data contains the invalid state name(s).

Controller

9.9 *EventHandler* objects

9.9.1 Basic event handler objects

EventHandler

EnhancedPREMOObject
Callback redef *(callback)*

register

$eventType_{in}$: EventName
$constraints_{in}$: seq *RefConstraint* [Deep Copy]
$fullConstraintMatchMode_{in}$: AndOr
$objectRef_{in}$: RefCallback
id_{out}: EventId

This operation registers interest in an event. The name of the event, a constraint list that specifies which event data values are of interest, and an object reference (to a *Callback* object) are supplied. An identification of the event registration is supplied as an output value to uniquely identify this registration.
If the sequence *$constraints_{in}$* is empty, no constraint comparison occurs, and the event is always dispatched. Otherwise, see 7.7 (page 10) for a detailed description on how the constraint list is interpreted for an event instance to be dispatched.

Exceptions raised: None

unregister

id_{in}: EventId
exceptions: {*InvalidEventId*}

This operation reverses the effects of register. The single parameter is an *EventId* obtained from a previous register call.

Exceptions raised:

InvalidEventId	The *id_{in}* value is not valid for the event handler instance.

$dispatchEvent_a$

$newEvent_{in}$: RefEvent [Shallow Copy]

When an object (i.e., the event source) wishes to generate an event it calls this operation. The event (containing the event name, the event data, and the event source) is provided as parameter. The operation is asynchronous.

Exceptions raised: None.

callback == *dispatchEvent*

EventHandler

9.9.2 *SynchronizationPoint* object

SynchronizationPoint object is a subtype of *EventHandler* objects; it adds constraint on the behaviour of the *dispatchEvent* operation.

SynchronizationPoint

EventHandler redef *(initialize, register, unregister, dispatchEvent)*

Ξ*initialize*

initValue$_{in}$*: Value*

The internal array of synchronization events is initialized.

Exceptions raised: None.

register

eventType$_{in}$*: EventName*
constraints$_{in}$*:* seq *RefConstraint* [Deep Copy]
fullConstraintMatchMode$_{in}$*: AndOr*
objectRef$_{in}$*: RefCallback*
id$_{out}$*: EventId*
exceptions: {*InvalidEventId*}

This operation overloads the operation *EventHandler.register*. The only difference in behaviour is that the operation raises an exception if *eventType*$_{in}$ does not refer to an event which has been previously added to the internal set of synchronization events through *addSyncEvent*.

Exceptions raised:

InvalidEventId	The *eventType*$_{in}$ refers to an event which is not present in the internal set of synchronization events.

unregister

id$_{in}$*: EventId*
exceptions: {*InvalidEventId*}

This operation reverses the effects of register. The single parameter is an *EventId* obtained from a previous register call. The operation leaves the internal set of synchronization events unchanged.

Exceptions raised:

InvalidEventId	The *id*$_{in}$ value is not valid for the event handler instance.

addSyncEvent

syncEvent$_{in}$*: RefEvent* [Shallow Copy]
exceptions: {*RepeatedEvent*}

The event is registered in the internal set of synchronization events. The same event cannot be registered twice.

Exceptions raised:

RepeatedEvent	The event *syncEvent*$_{in}$ is already registered.

deleteSyncEvent

syncEvent$_{in}$*: RefEvent* [Shallow Copy]
exceptions: {*UnknownEvent*}

The event is deleted from the internal set of synchronization events.

Exceptions raised:

UnknownEvent	*syncEvent*$_{in}$ has not been registered in the internal set.

dispatchEvent$_a$

newEvent$_{in}$*: RefEvent* [Shallow Copy]
exceptions: {*UnknownEvent*}

The original behaviour of *dispatchEvent*, as defined for the object type *EventHandler*, is modified by filtering the incoming event: *newEvent*$_{in}$ is dispatched if and only if it has been registered in the internal set. If not, the event is ignored, and an exception is raised.

Exceptions raised:

UnknownEvent	*newEvent*$_{in}$ has not been registered in the internal set.

SynchronizationPoint

9.9.3 *ANDSynchronizationPoint* object

As a subtype of *SynchronizationPoint*, *ANDSynchronizationPoint* objects attach an additional boolean flag to each event stored in the internal event set of the object; settings of these flags may also influence dispatching of incoming events. Essentially, a logical “and” on the incoming event sequences is performed.

ANDSynchronizationPoint

SynchronizationPoint redef *(initialize, initializeOnCopy, addSyncEvent, deleteSyncEvent, dispatchEvent)*

Ξinitialize

$initValue_{in}$: *Value*

The internal array of synchronization events is initialized; all internal flags, attached to the elements of this set, are set to *FALSE*.

Exceptions raised: None.

ΞinitializeOnCopy

All internal flags, attached to the elements of this set, are set to *FALSE*.

Exceptions raised: None.

addSyncEvent

$syncEvent_{in}$: *RefEvent* [Shallow Copy]
exceptions: {*RepeatedEvent*}

The event is registered in the internal set of synchronization events. The same event cannot be registered twice. The value of the associated flag is set to *FALSE*.

Exceptions raised:

RepeatedEvent	The event $syncEvent_{in}$ is already registered.

deleteSyncEvent

$syncEvent_{in}$: *RefEvent* [Shallow Copy]
exceptions: {*UnknownEvent*}

The event is deleted from the internal set of synchronization events. Also, all events, having the same *eventName* and *eventData* structure tag values as $syncEvent_{in}$, have their associated flag set to *FALSE*.

Exceptions raised:

UnknownEvent	$syncEvent_{in}$ has not been registered in the internal set.

dispatchEvent$_a$

newEvent$_{in}$*: Event*
exceptions: {*UnknownEvent*}

The original behaviour of *dispatchEvent*, as defined for the object type *SynchronizationPoint*, is modified by possibly delaying the notification of event reception. The incoming event is compared with the events stored in the internal set of synchronization events; if a matching event is found (i.e., the event stored in the set is equal to *newEvent*$_{in}$), the flag associated with this element is set to *TRUE*. If all events, having the same *eventName* and *eventData* structure tag values, are flagged with a value *TRUE*, the original behaviour of *dispatchEvent* applies (i.e., the event clients are notified) and all registered events, having the same *eventName* and *eventData* structure tag values, have their associated flag cleared to *FALSE*.

If *newEvent*$_{in}$ is not registered in the internal set, the event is not dispatched, and an exception is raised.

Exceptions raised:

UnknownEvent	*syncEvent*$_{in}$ has not been registered in the internal set.

ANDSynchronizationPoint

9.10 Timing objects

9.10.1 *Clock* object

Clock$_{abstract}$

EnhancedPREMOObject

tickUnit: TimeUnit
accuracyUnit: TimeUnit
accuracy: Time [Retrieve Only]

Unit of ticks, of the accuracy measure, and the current accuracy value of the clock. See the description of the object behaviour in 7.8.1 (page 12) for further details.

inquireTick

tick$_{out}$*: Time*

Returns the number of ticks elapsed. Subtypes of *Clock* shall attach a precise semantics to this operation.

Exceptions raised: None.

Clock

9.10.2 *SysClock* object

SysClock

Clock redef *(inquireTick)*

inquireTick

tick$_{out}$*: Time*

Returns the number of ticks elapsed since the start of the PREMO era, i.e., 00:00am, 1st January 1970, UTC.

Exceptions raised: None.

SysClock

9.10.3 *Timer* object

The *Timer* object is described as a finite state machine; the state transition table is as follows:

From: \ To:	*TSTOPPED*	*TSTARTED*	*TPAUSED*
TSTOPPED	Y	Y	N
TSTARTED	Y	Y	Y
TPAUSED	Y	Y	Y

Timer

Clock redef *(inquireTick)*

timerCurrentState: State [Retrieve Only]

start == σ*(TSTARTED, TSTOPPED | TSTARTED)*
stop == σ*(TSTOPPED)*
pause == σ*(TPAUSED)*
resume == σ*(TSTARTED, TPAUSED | TSTARTED)*

reset

The internal time register is reset to value 0.

Exceptions raised: None

inquireTick

$tick_{out}$*: Time*

The operation returns the elapsed time the object spent in *TSTARTED* state since it the internal time register has been set to zero.

Exceptions raised: None.

Timer

9.11 Synchronization objects

9.11.1 *Synchronizable* object

The *Synchronizable* object is described as a finite state machine; the state transition table is as follows:

From: \ To:	*STOPPED*	*STARTED*	*PAUSED*	*WAITING*
STOPPED	Y	Y	N	N
STARTED	Y	Y	Y	I
PAUSED	Y	Y	Y	N
WAITING	Y	Y	Y	N

Synchronizable[*C*]

EnhancedPREMOObject redef *(initialize, initializeOnCopy)*
CallbackByName

currentState: State [Retrieve Only]
currentPosition: C [Retrieve Only]
minimumPosition: C [Retrieve Only]
maximumPosition: C [Retrieve Only]

Minimum position shall always be smaller than the maximum position.

startPosition: C
endPosition: C

End position of progression. The relation

$$minimumPosition \leq startPosition < endPosition \leq maximumPosition$$

shall always hold.

Exceptions raised (when setting the attributes):

WrongValue	The new position does not abide to the required relation, or is not allowed for the current object type.
WrongState	The object state should have been *STOPPED*

currentDirection: Direction
loopCounter: **N** [Retrieve Only]
repeatFlag: Boolean
nloop: **N**

The direction flag, loop counter, the repeat flag, and the number of loop of progression.

Exceptions raised (when setting the attributes):

WrongState	The object state should have been *STOPPED*.

Ξ*initialize*

initValue$_{in}$*: Value*

The start and end positions are set to their default values (defined in the subtypes of *Synchronizable*), the current position is set to the start position, the repeat flag is set to *FALSE*, and both the values of 'nloop' and the loop counter is set to 1. The state of the object is set to *STOP; initValue*$_{in}$ is disregarded.

Exceptions raised: None.

Ξ*initializeOnCopy*

initValue$_{in}$*: Value*

The start and end positions are set to their default values (defined in the subtypes of *Synchronizable*), the current position is set to the start position, the repeat flag is set to *FALSE*, and both the values of 'nloop' and the loop counter is set to 1. The state of the object is set to *STOP*. *initValue*$_{in}$ is disregarded.

Exceptions raised: None.

Ξ*progressPosition*

newPosition$_{out}$*: C*

A new possible position, as used by the object in state *STARTED*, is calculated. The new value shall be greater than the current position if the value of *currentDirection* is *Forward*, and smaller otherwise. The function shall return a value between *minimumPosition* and *maximumPosition*, and shall not return the values of "∞" and "-∞". This operation is protected; specific subtypes of *Synchronizable* should redefine this operation to implement progress on, e.g., various media.

Exceptions raised: None.

ΞprocessData

$intervalMin_{in}$: C
$intervalMax_{in}$: C

Data on the progression space between *$intervalMin_{in}$* and *$intervalMax_{in}$* are “processed”, for example, presented directly, or extracted for processing within some containing component such as a media device. If the value of *currentDirection* is *Forward*, data should be processed from *$intervalMin_{in}$* toward *$intervalMax_{in}$*, including the datum at *$intervalMin_{in}$* but not including *$intervalMax_{in}$*. Otherwise, data should be processed from *$intervalMax_{in}$* toward *$intervalMin_{in}$*, including the datum at *$intervalMax_{in}$* but not including *$intervalMin_{in}$*.
This operation is protected; specific subtypes of *Synchronizable* should redefine this operation to implement data processing on, e.g., various media.

Exceptions raised: None.

start == *σ(STARTED, STOPPED | STARTED)*
stop == *σ(STOPPED)*
pause == *σ(STOPPED, STOPPED) ⊕ σ(PAUSED)*
resume == *σ(STARTED, PAUSED | WAITING | STARTED)*

Exceptions raised:

WrongState	The state transition operation cannot be issued in the current state.

resetLoopCounter

exceptions: {WrongState}

The value of the loop counter is set to the current value of *nloop*.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.

jump

$newPosition_{in}$: C
exceptions: {WrongState, WrongValue}

The current position of the object is changed. No synchronization actions are performed during this change, even if the interval defined by the current position and *$newPosition_{in}$* includes reference points. The object’s state should be either *PAUSED* or *STOPPED*.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	The new position should be between the start and the end positions.

setSyncElement

refPoint$_{in}$*: C*
syncData$_{in}$*: RefSyncElement*
exceptions: {*WrongState, WrongValue*}

The synchronization element *syncData*$_{in}$ is stored at reference point *refPoint*$_{in}$.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	The value of *refPoint*$_{in}$ is not between the start and the end position, or a synchronization element was already defined for that position. The first tag in the exception data is set to the string "*WrongPosition*" or "*Overwrite*", respectively.

deleteSyncElement

refPoint$_{in}$*: C*
exceptions: {*WrongState, WrongValue*}

The synchronization element, stored at reference point *refPoint*$_{in}$ is deleted.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	The value of *refPoint*$_{in}$ is not between the start and the end position, or no synchronization elements have been stored at *refPoint*$_{in}$. The first tag in the exception data is set to the string *"WrongPosition"* or *"NoSync"*, respectively.

getSyncElements

refPoint1$_{in}$*: C*
refPoint2$_{in}$*: C*
syncData$_{out}$: seq (*RefSyncElement* × *C*) [Shallow Copy]
exceptions: {*WrongValue*}

All synchronization elements, together with the corresponding reference points defined between *refPoint1*$_{in}$ and *refPoint2*$_{in}$, are returned. The returned sequence includes the synchronization elements defined through the *setPeriodicSyncElement* operation, too.

Exceptions raised:

WrongValue	The values are invalid; either *refPoint1*$_{in}$ > *refPoint2*$_{in}$ or the values are not within the bounds of the object.

setPeriodicSyncElement

$startRefPoint_{in}$: C
$endRefPoint_{in}$: C
$periodicity_{in}$: C
$syncData_{in}$: RefSyncElement
exceptions: {WrongState, WrongValue}

The same synchronization element is defined for reference points at:

*$startRefPoint_{in}$, $startRefPoint_{in}$+$periodicity_{in}$, $startRefPoint_{in}$+2*$periodicity_{in}$, . . .,*

etc., until the end position of the object is reached or the value of *$endRefPoint_{in}$* is exceeded. The reference points may extend beyond the current span, but only those which are within the span are considered when the object is in *STARTED* mode.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	*$startRefPoint_{in}$* or *$endRefPoint_{in}$* is not between the minimum and maximum positions or their values are incorrect, the *$periodicity_{in}$* value is not positive, or a synchronization element would be overwritten. The first tag in the exception data is set to the string "*WrongPosition*", "*WrongPeriodicity*", or "*Overwrite*", respectively. In the last case, the rest of the exception tags list the conflicting positions.

deletePeriodicSyncElement

$startRefPoint_{in}$: C
$endRefPoint_{in}$: C
$periodicity_{in}$: C
exceptions: {WrongState, WrongValue}

The synchronization elements defined for reference points at:

*$startRefPoint_{in}$, $startRefPoint_{in}$+$periodicity_{in}$, $startRefPoint_{in}$+2*$periodicity_{in}$, . . .,*

etc., until the end position of the object is reached, or the value of *$endRefPoint_{in}$* is exceeded, are deleted.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	Either *$startRefPoint_{in}$* or *$endRefPoint_{in}$* is not between the minimum and maximum positions, the *$periodicity_{in}$* value is not positive, or no synchronization elements have been stored at these points. The first tag in the exception data is set to the string "*WrongPosition*", "*WrongPeriodicity*", or "*NoSync*", respectively.

setActionOnPair

$stateOld_{in}$: State
$stateNew_{in}$: State
$action_{in}$: RefActionElement
exceptions: {*WrongState, WrongValue*}

An action is associated to the tuple *$stateOld_{in}$* × *$stateNew_{in}$*; see 7.9.1.2 on how this action element is used by the *Synchronizable* object. This operation replaces any action which has been previously defined on the same tuple.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	One of the states *$stateOld_{in}$* or *$stateNew_{in}$* does not identify a valid state The exception data contains the invalid state name(s).

removeActionOnPair

$stateOld_{in}$: State
$stateNew_{in}$: State
exceptions: {*WrongState, WrongValue*}

If an action has been previously set by a *setActionPair* operation on the tuple *$stateOld_{in}$* × *$stateNew_{in}$*, it is removed.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	One of the states *$stateOld_{in}$* or *$stateNew_{in}$* does not identify a valid state. The exception data contains the invalid state name(s).

clearSyncElements

exceptions: {*WrongState*}

All synchronization and action elements are removed.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.

Synchronizable

9.11.2 *TimeSynchronizable* object

TimeSynchronizable[*C*]

Synchronizable[*C*] redef *(start, stop, resume, pause)*
Timer redef *(start, stop, resume, pause, reset)*

speed: **R**

Number of units on *C* for each tick in *Time*. The value depends on the current value of the attribute unit (inherited from the *Clock* object). See 7.9.2 (page 16) for further details.

Exceptions raised (when setting the speed):

WrongState	The object state should have been *PAUSED* or *STOPPED*.

currentPosition: Time	[Retrieve Only]
minimumPosition: Time	[Retrieve Only]
maximumPosition: Time	[Retrieve Only]
startPosition: Time	
endPosition: Time	

These attributes have the same semantics as the attributes with similar names of *Synchronizable*, except that the values are expressed in relative *Time* rather than progression space. See page 47 for further details.

start	==	*Timer.start* ; *Synchronizable.start*
resume	==	*Timer.resume* ; *Synchronizable.resume*
pause	==	*Timer.pause* ; *Synchronizable.pause*

The transition operations affect the merged finite state machine of the object. See 7.9.2 (page 16) for further details.

stop

The *Timer.stop* ; *Synchronizable.stop* action is performed; furthermore, a marker is put against the default start position in the progression space. This marker serves as a zero point for relative positioning expressed in time values.

Exceptions raised: None

reset

The internal time register is set back to zero (this behaviour is inherited from *Timer*), and a marker is put against the current position in the progression space. This marker serves as a zero point for relative positioning expressed in time values.

Exceptions raised: None

timeToSpace

$positionTime_{in}$: *Time*
$positionSpace_{out}$: *C*

Returns the transformed value; takes into account the current value of *speed*, and the marker put by a previous reset operation, or the value of *startPosition* (if no *reset* has not been invoked).

Exceptions raised: None.

spaceToTime

$positionSpace_{in}$: *C*
$positionTime_{out}$: *Time*

Returns the transformed value; takes into account the current value of *speed*, and the marker put by a previous reset operation, or the value of *startPosition* (if no *reset* has not been invoked).

Exceptions raised: None.

jump

$newPosition_{in}$: *Time*
exceptions: {*WrongState*, *WrongValue*}

The value of $newPosition_{in}$ is transformed onto the progression space (using *timeToSpace*), and the resulting value is used to invoke *Synchronizable.jump*.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	The new position, calculated by *timeToSpace*, should be between the start and the end positions.

setSyncElement

$refTime_{in}$: Time
$syncData_{in}$: RefSyncElement
exceptions: {*WrongState*, *WrongValue*}

The synchronization element *$syncData_{in}$* is stored at the time value *$refTime_{in}$*. Using the values of *speed* and the time register, a corresponding reference point is also set on the progression space; however, if the *speed* and/or the time register change, the reference point on the progression space must be re–set.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	A synchronization element is already defined at the time value *$refTime_{in}$*. The first tag in the exception data is set to the string "*Overwrite*".

deleteSyncElement

$refTime_{in}$: Time
exceptions: {*WrongState*, *WrongValue*}

The synchronization element, stored at the time value *$refTime_{in}$* is deleted.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	No synchronization elements have been stored at the time value *$refPoint_{in}$*. The first tag in the exception data is set to the string "*NoSync*".

getSyncElements

$refTime1_{in}$: Time
$refTime2_{in}$: Time
$syncData_{out}$: seq *(RefSyncElement* × *Time)* [Shallow Copy]
exceptions: {*WrongValue*}

All synchronization elements, together with the corresponding reference points defined between *$refTime1_{in}$* and *$refTime2_{in}$*, are returned. The returned sequence includes the synchronization elements defined through the *setPeriodicSyncElement* operation, too.

Exceptions raised:

WrongValue	The values are invalid; i.e., *$refTime1_{in}$* > *$refTime2_{in}$*.

setPeriodicSyncElement

$startRefTime_{in}$: Time
$endRefTime_{in}$: Time
$periodicity_{in}$: Time
$syncData_{in}$: RefSyncElement
exceptions: {WrongState, WrongValue}

The same synchronization element is defined at the time values:

*$startRefTime_{in}$, $startRefTime_{in}$+$periodicity_{in}$, $startRefTime_{in}$+2*$periodicity_{in}$, . . .,*

etc., until the value of *$endRefTime_{in}$* is exceeded. Using the values of *speed* and the time register, corresponding reference points are also set on the progression space; however, if the *speed* and/or the time register change, the reference points on the progression space must be redefined.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	*$startRefTime_{in}$* or *$endRefTime_{in}$* have incorrect values, the *$periodicity_{in}$* value is not positive, or a synchronization element would be overwritten. The first tag in the exception data is set to the string *"WrongPosition"*, *"WrongPeriodicity"*, or *"Overwrite"*, respectively. In the last case, the rest of the exception tags list the conflicting time values.

deletePeriodicSyncElement

$startRefTime_{in}$: Time
$endRefTime_{in}$: Time
$periodicity_{in}$: Time
exceptions: {WrongState, WrongValue}

The synchronization elements defined at the time values:

*$startRefTime_{in}$, $startRefTime_{in}$+$periodicity_{in}$, $startRefTime_{in}$+2*$periodicity_{in}$, . . .,*

etc., until the value of *$endRefTime_{in}$* is exceeded, are deleted.

Exceptions raised:

WrongState	The object state should have been in *PAUSED* or *STOPPED*.
WrongValue	The values of *$startRefTime_{in}$* or *$endRefTime_{in}$* are incorrect, the *$periodicity_{in}$* value is not positive, or no synchronization elements have been stored at these time values. The strings *"WrongPosition"*, *"WrongPeriodicity"*, or *"NoSync"*, respectively, are set as the first tag in the exception data.

TimeSynchronizable