

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

Part 2: Foundation component

ICS 33.160.60; 35.140

National foreword

This British Standard reproduces verbatim ISO/IEC 14478-2:1998 and implements it as the UK national standard.

The UK participation in its preparation was entrusted to Technical Committee IST/31, Computer graphics and image processing, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC title page, pages ii to vi, pages 1 to 71 and a back cover.

Amendments issued since publication

Amd. No.	Date	Text affected

This British Standard, having been prepared under the direction of the DISC Board, was published under the authority of the Standards Committee and comes into effect on 15 March 1999

© BSI 03-1999

ISBN 0 580 30076 5

INTERNATIONAL STANDARD

ISO/IEC 14478-2

First edition
1998-12-15

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'objets multimédia (PREMO) —*

Partie 2: Composant fondamental



Contents

Foreword.....v

Introduction vi

1 Scope 1

2 Normative references.....1

3 Definitions.....1

3.1 PREMO Part 1 definitions 1

3.2 Additional definitions.....1

4 Symbols and abbreviations.....3

5 Conformance 3

6 Foundation non-object types.....3

7 Foundation object types 5

7.1 Introduction 5

7.2 PREMO objects and fundamental object behaviour 5

7.2.1 Creation and destruction of objects.....5

7.2.2 Inquiries on types 5

7.3 Simple PREMO objects 6

7.3.1 Structures.....6

7.4 Callback objects 6

7.5 Enhanced PREMO Objects 7

7.5.1 Object properties.....7

Descriptors: data processing, information interchange, graphic data processing, image processing, video data, audio data, coding (data conversion), coded representation.

7.6	Controller objects	8
7.7	Event handler objects	10
7.7.1	Basic Event Handler objects	10
7.7.2	Synchronization Points	11
7.8	Time objects	12
7.8.1	Clock object	12
7.8.2	System clock object	12
7.8.3	Timer object	12
7.9	Synchronization	12
7.9.1	Event Synchronizable objects	12
7.9.2	Time synchronizable objects	16
7.9.3	Time slave objects	17
7.9.4	Time line objects	18
8	Enhanced property management and factories	19
8.1	Enhanced Property management	19
8.1.1	Motivation	19
8.1.2	Capabilities and native property values: the <i>PropertyInquiry</i> type	20
8.1.3	Property constraint and selection: the <i>PropertyConstraint</i> type	21
8.2	Creating PREMO objects	25
8.2.1	Generic Factory objects	25
9	Functional specification	27
9.1	Introduction	27
9.2	Common non-object data types	27
9.3	Exceptions	29
9.4	<i>PREMOObject</i> and fundamental object behaviour	30
9.5	Simple PREMO object and structures	31
9.5.1	<i>SimplePREMOObject</i>	31
9.5.2	Event structure	31
9.5.3	Constraint structure	31
9.5.4	Action Element	32
9.5.5	Synchronization Element	32
9.6	Callback objects	33
9.7	Enhanced PREMO object	34
9.8	<i>Controller</i> object	37
9.9	<i>EventHandler</i> objects	40
9.9.1	Basic event handler objects	40
9.9.2	<i>SynchronizationPoint</i> object	41
9.9.3	<i>ANDSynchronizationPoint</i> object	43
9.10	Timing objects	45
9.10.1	<i>Clock</i> object	45
9.10.2	<i>SysClock</i> object	45
9.10.3	<i>Timer</i> object	46
9.11	Synchronization objects	47
9.11.1	<i>Synchronizable</i> object	47
9.11.2	<i>TimeSynchronizable</i> object	53
9.11.3	<i>TimeLine</i> object	57
9.11.4	<i>TimeSlave</i> object	58
9.12	Enhanced Property management	59
9.12.1	<i>PropertyInquiry</i> object	59
9.12.2	<i>PropertyConstraint</i> object	60
9.13	Creating PREMO objects	63
9.13.1	<i>GenericFactory</i> object	63
9.13.2	<i>FactoryFinder</i> object	64
10	Component specification	65
A	Overview of PREMO Foundation Object Types	66

B Extensibility for PREMO objects.....70

C An example for event-based synchronization.....71

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, government and non–governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee ISO/IEC JTC1. Draft International Standards adopted by the joint technical committees are circulated to the national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

ISO/IEC 14478–2 was prepared by Technical Committee ISO/IEC JTC1, *Information technology*, Subcommittee SC24, *Computer graphics and image processing*.

ISO/IEC 14478 consists of the following four parts under the general title *Information technology — Computer graphics and image processing — Presentation environments for multimedia objects (PREMO)*:

- *Part 1: Fundamentals of PREMO*
- *Part 2: Foundation Component*
- *Part 4: Multimedia Systems Services*
- *Part 3: Modelling, Rendering, and Interaction Component*

Annexes A and B form an integral 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 this international standard 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 10744:1992, *Information technology — Coding of Moving Pictures and Associated Audio for Digital Storage up to about 1.5 Mbit/s (MPEG)*.

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):

$$\begin{aligned} \text{ConstraintOp} ::= & \text{Equal} \mid \text{NotEqual} \\ & \mid \text{GreaterThan} \mid \text{GreaterThanOrEqual} \mid \text{LessThan} \mid \text{LessThanOrEqual} \\ & \mid \text{Prefix} \mid \text{Suffix} \mid \text{NotPrefix} \mid \text{NotSuffix} \\ & \mid \text{Includes} \mid \text{Excludes} \end{aligned}$$

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

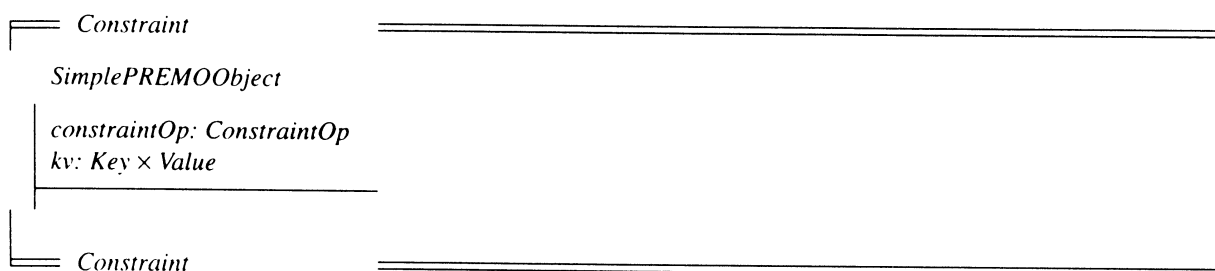
The term “structure” does not denote a specific object type in PREMO but, instead, a category of types. These object types are characterized by:

- 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*.

EnhancedPREMOObject 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 values¹⁾ 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.

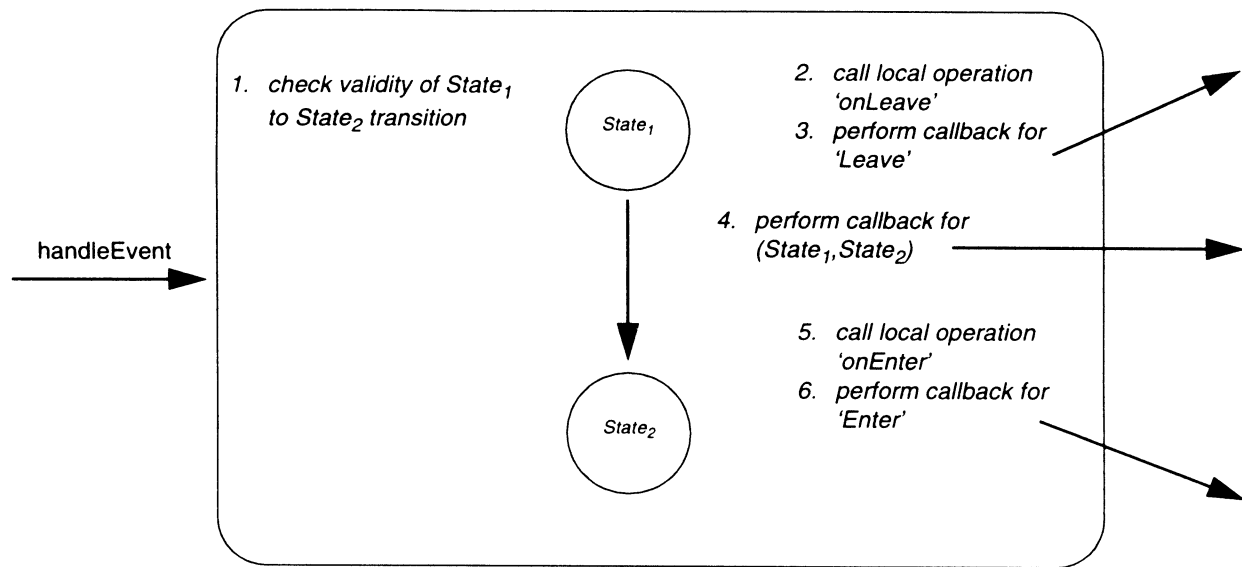


Figure 1 — Controller objects

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, *State₁* is the name of the current state of the object, and *State₂* 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) *State₁* and *State₂*. The operation returns data suitable for an event data tag in an event structure.
 - ii) If there is no *ActionElement* structure associated to *State₁* labelled as 'Leave', this step is ignored. Otherwise, an event instance is created, using the event name in the *ActionElement* structure associated to *State₁* 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 *State₂*.

- 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.

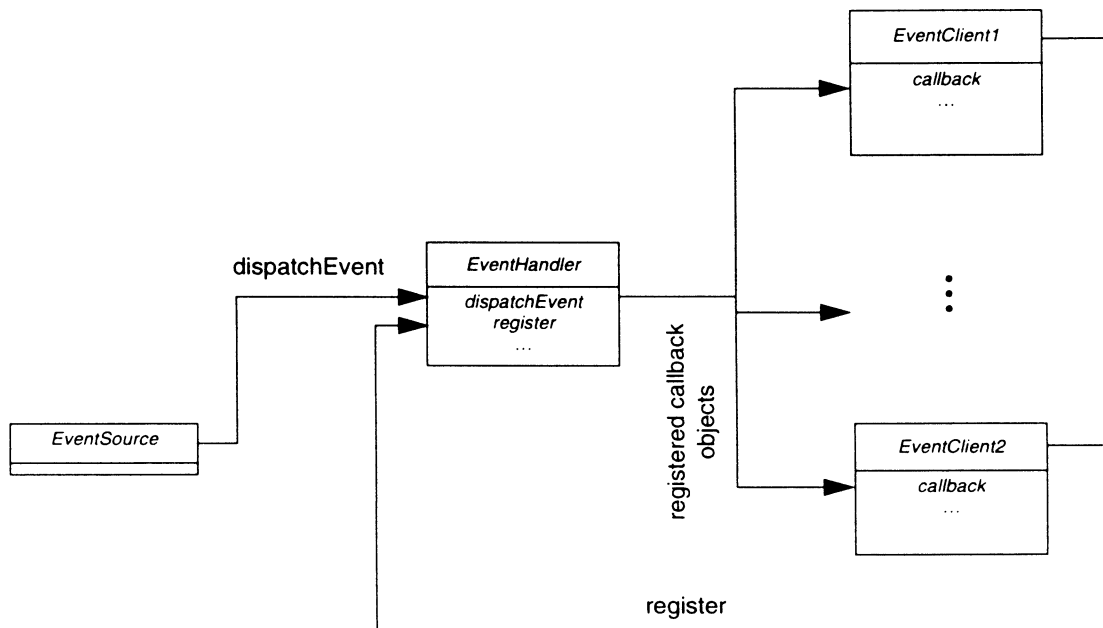


Figure 2 — PREMO event model

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 ($Time_\infty$).

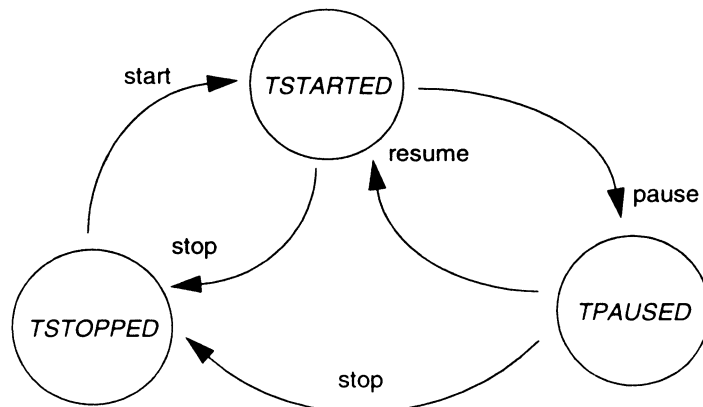


Figure 3 — State transitions in a *Timer* object

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:

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

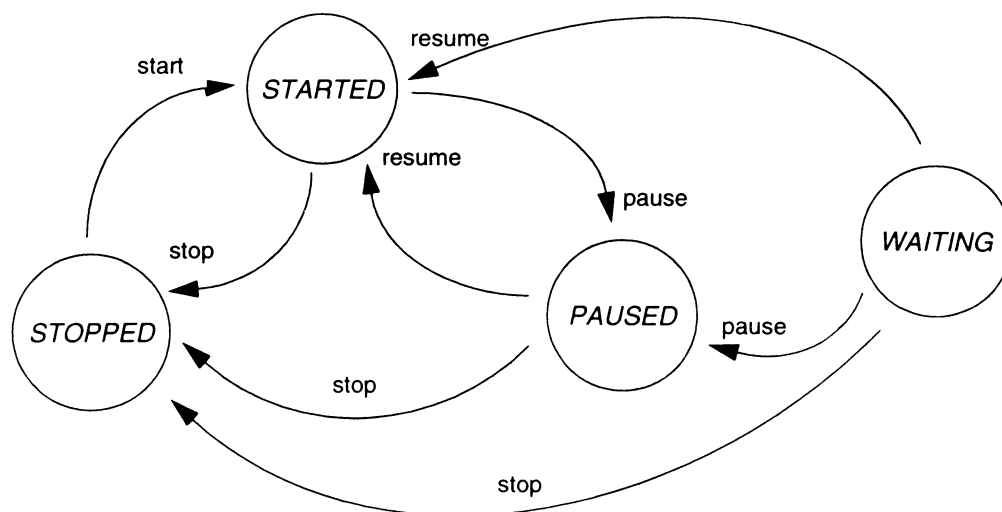


Figure 4 — State transition diagram for a *Synchronizable* object

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 *NULLObj*ect, 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.

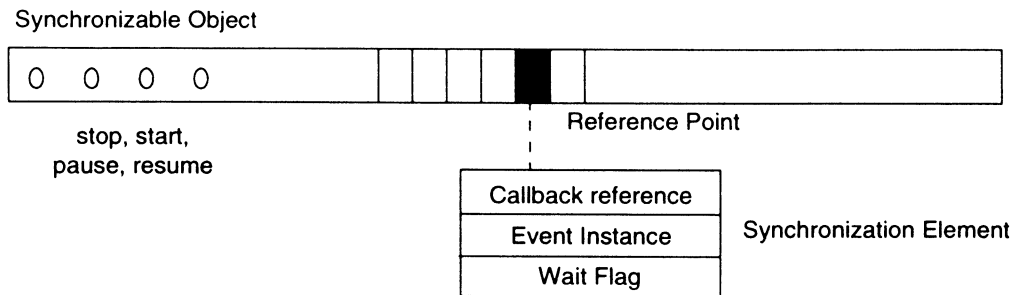


Figure 5 — Synchronizable object

- 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