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, 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 14478–1 was prepared by Joint Technical Committee ISO/IEC JTC1, *Information technology*, Subcommittee SC24, *Computer graphics and image processing*.

ISO/IEC 14478consists 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*

Additional parts may be defined as this work progresses.

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

Introduction

The need for a coordinated method for addressing all aspects of the construction of, presentation of, and interaction with multimedia objects has led to the standardization of this Presentation Environment for Multimedia Objects. Multimedia means objects consisting of still computer graphics, moving computer graphics (animation), synthetic graphics of all types, audio, still images, moving images (including video), images coming from imaging operations, and any other content type or combination of content types that can be "presented". ISO/IEC 14478 is extensible and configurable, and allows the separate, incremental development of additional standardized and non–standardized components to meet the needs of application communities.

PREMO currently consists of the following parts:

Part 1: Fundamentals of PREMO

Contains a motivational overview of PREMO giving its scope, justification, and an explanation of key concepts, describes the overall architecture of PREMO, and specifies the common semantics for specifying the externally visible characteristics of PREMO objects in an implementation-independent way.

Part 2: Foundation component

This component lists an initial set of object types and non–object types useful for the construction of, presentation of, and interaction with multimedia information. Any conforming PREMO implementation shall support these object types.

Part 3: Multimedia Systems Services Component

Describes objects that provide an infrastructure for building multimedia computing platforms that support interactive multimedia applications dealing with synchronized, time-based media in a heterogeneous distributed environment.

Part 4: Modelling, Presentation, and Interaction Component

Describes objects which are needed for advanced computer systems using graphics, video, audio, or other types of presentable media enhanced by time aspects.

NOTE — Further internationally standardized components are expected to be developed within ISO/IEC JTC1/SC24 and by other subcommittees.

Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 1: Fundamentals of PREMO

1 Scope

ISO/IEC 14478 specifies techniques for supporting interactive single, and multiple media applications which recognize and emphasize the interrelationships among user interfaces, multimedia applications, and multimedia information interchange.

ISO/IEC 14478 defines a flexible environment to encompass modular functionality and is extensible through the creation of future components, both within and outside of standards committees. It supports a wide range of multimedia applications in a consistent way, from simple drawings up to full motion video, sound, and virtual reality environments.

ISO/IEC 14478 is independent of any particular implementation language, development environment, or execution environment. For integration into a programming environment, the standard shall be embedded in a system dependent interface following the particular conventions of that environment. ISO/IEC 14478 provides versatile packaging techniques beyond the capabilities of monolithic single–media systems. This allows rearranging and extending functionality to satisfy requirements specific to particular application areas. ISO/IEC 14478 is developed incrementally with parts 1 through 4 initially available. Other components are expected to be standardized by ISO/IEC JTC1 SC24 or other subcommittees.

ISO/IEC 14478 provides a framework within which application–defined ways of interacting with the environment can be integrated. Methods for the definition, presentation, and manipulation of both input and output objects are described. Application–supplied structuring of objects is also allowed and can, for example, be used as a basis for the development of toolkits for the creation of, presentation of, and interaction with multimedia and hyper–media documents and product model data.

ISO/IEC 14478 is able to support construction, presentation, and interaction with multiple simultaneous inputs and outputs using multiple media. Several such activities may occur simultaneously, and the application program can adapt its behaviour to make best use of the capabilities of its environment.

ISO/IEC 14478 includes interfaces for external storage, retrieval and interchange of multimedia objects.

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 14478-2:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 2: Foundation Component.*

ISO/IEC 14478-3:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 3: Multimedia Systems Services.*

ISO/IEC 14478-4:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 4: Modelling, Rendering, and Interaction Component.*

3 Definitions

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

3.2.1 multimedia: The creation, editing, composing, and/or presentation of products consisting of any combination of *media*.

3.2.1.1 multimedia object: An *object* consisting of one or more types of media that can be presented to a user.

3.2.2 medium (plural media): A means by which information is perceived, expressed, stored, or transmitted. Examples include audio, video, (animated) graphics, images, text.

3.2.3 dynamic interactive graphics: Graphics applications where the graphics varies in real–time in response to user inputs.

3.2.4 animation: Series of pictures in a time–ordered sequence to display as a *video* medium. This covers all changes that have a visual effect. It thus includes the time–varying position, shape, colour, transparency, structure, and texture of an object, and changes in lighting, camera position, orientation, and focus, and even changes of rendering techniques.

3.2.5 text: A medium encompassing a character–based encoding only.

3.2.6 audio: A medium encompassing all forms of information transmitted by sound.

3.2.7 video: A medium encompassing a continuous series of pictures typically depicting motion or time sequenced events.

3.2.8 co–representation: A representation of information where the same information is presented in several different ways from which the most appropriate is chosen.

3.2.9 component: A PREMO component is a set of related object types and non–object types. The object and non–object types of a component are clustered into (component) profiles.

3.2.9.1 standard component: A component that is defined in one of the parts of ISO/IEC 14478, or a component that has been registered by an approved registration authority, and conforming to the rules defined for components in PREMO.

3.2.9.2 non–standard component: A component that is not a standard component, but which obeys the rules defined by PREMO for conforming components.

3.2.10 profile: A profile is set of related object types and non–object types from which objects can be instantiated, and a configuration specification which defines dependencies between object types and other profiles. Profiles offer a set of services embodied by the operations defined on its constituent object types.

3.2.10.1 basic profile (of a component): A mandatory set of object and non–object types for a specific component which shall be provided by all complying implementation of a component.

3.2.11 object: An entity that encapsulates some private *state* information or data, a set of associated operations that manipulate the data, and possibly a thread of control so that collectively they can be treated as a single unit.

3.2.11.1 object type: It defines the *operations* of objects; these operations collectively characterize the object's behaviour.

3.2.11.2 object reference: An object name which reliably denotes a particular object instance. This is a non–object.

3.2.12 state: Information encapsulated within an object that has to be remembered when one operation alters the future behaviour of future operations.

3.2.13 identity (of objects): Objects have a unique, immutable identity which provides a means to denote or refer to the object independent of its state or behaviour.

3.2.14 attribute (of objects): A pair of accessor and mutator functions, to retrieve the value of to set the value of the attribute.

3.2.14.1 retrieve only attribute: An object attribute where the mutator function to set the value, though conceptually present, does not change the value of the attribute, and raises an exception.

3.2.15 signature (of operations): An operation's signature consists of a list of parameter types, and a list of result types.

3.2.16 interface (of objects): The set of operation signatures defined for a type.

3.2.17 non–object types: Things that are not objects, e.g., integers, real numbers.

3.2.18 operation: Describes an action that can be applied to an object, using parameters.

3.2.18.1 operation invocation: Activation of an object's action, by describing its parameters, possibly causing results to be returned.

3.2.18.2 operation request: Synonym for operation invocation.

3.2.18.3 operation dispatching: The selection process which selects a specific operation implementation for execution.

3.2.18.4 operation overloading: The implementation of an operation defined for a *derived interface* supersedes the implementation of the same operation defined for a *base interface*. This effect is called operation overloading.

3.2.19 client: (of another object) An object issuing an operation request.

3.2.20 exception: Information returned if an error condition has occurred during the execution of a request of an operation.

3.2.21 controlling parameter: Special parameter conceptually present for all operations, used to control the way operations are dispatched. It refers to the object type on which the operation is defined.

3.2.22 subtyping: A relationship between types. It defines the rules by which objects of one type are determined to be acceptable in contexts expecting another type.

3.2.23 subtype: A type *S* is a subtype of another type *T* if any object of type *S* can be used in any context that expects an object of type *T*.

3.2.23.1 immediate subtype: A type *S* is an immediate subtype of another type *T*, if *T* is the immediate supertype of *S*.

3.2.24 supertype: A type *T* is a supertype of *S*, if *S* is a subtype of *T*.

3.2.24.1 multiple supertype: An object type can have more than one supertype; they are referred to as multiple supertypes.

3.2.24.2 immediate supertype: A type *T* is an immediate supertype of type *S*, if it is a supertype of *S*, and there is no type *Q* such that *Q* is a supertype of *S* and *T* is a supertype of *Q*.

3.2.25 direct instance: An object is a direct instance of a type *T*, if it is an instance of *T* and not an instance of any subtype of *T*.

3.2.26 immediate type: The type of the direct instance of an object instance.

3.2.27 type graph (of a type): The set of all supertypes of a type (including the recursively defined supertypes) plus the type itself.

3.2.28 inheritance : A notational mechanism for defining operation reuse. It is a relationship on interfaces.

3.2.28.1 multiple inheritance: A notational mechanism for defining operation reuse on multiple base interfaces.

3.2.28.2 single inheritance: As opposed to multiple inheritance; denotes an interface having only one base interface.

3.2.29 derived interface: If the interface *P* inherits from *Q*, *P* may also be referred to as a derived interface.

3.2.30 base interface: If the interface *P* inherits from *Q*, *Q* is a base interface (of *P*).

3.2.31 abstract object type: Non–instantiable object type.

3.2.32 operation request semantics: A finer control an object has to service an operation request.

3.2.32.1 operation receptor: A holder conceptually assigned to each operation in which operation requests are placed.

3.2.32.2 operation receptor mode: An immutable characteristics of an operation receptor.

3.32.2.1 synchronous operation receptor mode: One of the three possible modes of an operation receptor; callers are suspended on operation requests.

3.32.2.2 asynchronous operation receptor mode: One of the three possible modes of an operation receptor; callers are not suspended on operation requests, and the request's arguments are stored.

3.32.2.3 sampled operation receptor mode: One of the three possible modes of an operation receptor; callers are not suspended on operation requests, and only one requests argument is stored.

3.2.33 protected operation: An operation which can be invoked from within the object instance only; other object instances cannot request this operation.

3.2.34 life cycle (of objects and references): The mechanisms whereby objects and object references are created and destroyed.

3.2.34.1 life cycle facilities: The boundary between PREMO and its implementation environment, providing life cycle related services.

3.2.34.2 create facility: Facility to create objects possibly using initialization variables.

3.2.34.3 copy facility: Facility to create objects as copies of already existing objects.

3.34.3.1 shallow copy: Version of the copy facility when attribute values are set in the newly created object using the values of the attributes in the original object.

3.34.3.2 deep copy: Version of the copy facility when attribute values are set in the newly created object using the values of the attributes in the original object except for object references; in the case of object references, the referred objects are (deep) copied, and the new reference values are used to set the attributes.

3.2.34.4 cast facility: Facility to create an object reference to an already existing object, referring to a different immediate type.

3.2.34.5 destroyReference facility: Facility to destruct an object reference.

3.2.34.6 destroyObject facility: Facility to destruct an object instance.

3.2.35 type schema: A notational convention used to describe object types (see clause A.3).

3.2.36 operation schema: A notational convention used to describe an operation within a type schema.

3.2.37 component schema: A notational convention to describe components (see clause A.7).

3.2.38 profile schema: A notational convention to describe profiles within a component schema (see clause A.7).

3.2.39 generic type (schema): A notational convention used to describe a family of PREMO types, based on the general notational conventions of type schema (see clause B).

3.2.40 formal types: Symbols used in generic type schemas to denote non–specified object or non–object types.

3.2.41 actualization: A notational convention whereby generic type schema are used to define PREMO object types through replacing formal types by PREMO object or non–object types.

The following alphabetical list gives the subclause of each definition.

abstract object type	3.2.31
actualization	3.2.41
animation	3.2.4
asynchronous operation receptor mode	3.32.2.2

attribute (of objects)	3.2.14
audio	3.2.6
base interface	3.2.30
basic profile (of a component)	3.2.10.1
cast facility	3.2.34.4
client	3.2.19
co–representation	3.2.8
component	3.2.9
component schema	3.2.37
controlling parameter	3.2.21
copy facility	3.2.34.3
create facility	3.2.34.2
deep copy	3.34.3.2
derived interface	3.2.29
destroyObject facility	3.2.34.6
destroyReference facility	3.2.34.5
direct instance	3.2.25
dynamic interactive graphics	3.2.3
exception	3.2.20
formal types	3.2.40
generic type (schema)	3.2.39
identity (of objects)	3.2.13
immediate subtype	3.2.23.1
immediate supertype	3.2.24.2
immediate type	3.2.26
inheritance	3.2.28
interface (of objects)	3.2.16
life cycle (of objects and references)	3.2.34
life cycle facilities	3.2.34.1
medium (plural media)	3.2.2
multimedia	3.2.1
multimedia object	3.2.1.1
multiple inheritance	3.2.28.1
multiple supertype	3.2.24.1
non–object types	3.2.17
non–standard component	3.2.9.2
object	3.2.11
object reference	3.2.11.2
object type	3.2.11.1
operation	3.2.18
operation dispatching	3.2.18.3
operation invocation	3.2.18.1
operation overloading	3.2.18.4
operation receptor	3.2.32.1
operation receptor mode	3.2.32.2
operation request	3.2.18.2
operation request semantics	3.2.32

operation schema	3.2.36
profile	3.2.10
profile schema	3.2.38
protected operation	3.2.33
retrieve only attribute	3.2.14.1
sampled operation receptor mode	3.32.2.3
shallow copy	3.34.3.1
signature (of operations)	3.2.15
single inheritance	3.2.28.2
standard component	3.2.9.1
state	3.2.12
subtype	3.2.23
subtyping	3.2.22
supertype	3.2.24
synchronous operation receptor mode	3.32.2.1
text	3.2.5
type graph (of a type)	3.2.27
type schema	3.2.35
video	3.2.7

4 Symbols and abbreviations

CAD/CAM:	Computer-Aided Design / Computer-Aided Manufacturing.
IEC:	International Electrotechnical Commission.
IS:	International Standard.
ISO:	International Organization for Standardization.
JTC:	Joint Technical Committee.
PREMO:	Presentation Environments for Multimedia Objects.
2.5D:	Two and a half dimensional.
2D:	Two-dimensional.
3D:	Three-dimensional.

5 Conformance

A component is defined in PREMO to be a set of related object types and non–object types and a set of profile specifications. A component is considered to offer a set of services, embodied by the operations on the objects, and may also depend on services provided by other components.

PREMO defines conformance with respect to components as follows:

a) PREMO specifies conformance rules that shall apply for any definition of a conforming component;

b) PREMO specifies conformance rules that shall apply for any implementation of a conforming component;

c) PREMO specifies conformance rules that shall apply for any implementation of a conforming PREMO system.

A conforming component shall be defined according to the rules described in clause 9. It may also include additional requirements that shall apply to implementations of the component.

An implementation of a conforming component shall provide the mandatory set of functionality designated as a basic profile for that component, realize the configuration specification defined for that component, and in addition provide for any other implementation requirements that are given as part of the component's definition.

A standard component is a conforming component defined in one of the parts of PREMO or one that has been registered by an approved registration authority.

An implementation of a conforming PREMO system (i.e., a system implemented using PREMO components) shall obey the conformance requirements of each of the components from which it is constituted.

6 Requirements for PREMO

6.1 Introduction

Technology has evolved to the point that digital media has become an inherent part of most applications. In addition, many applications use multiple presentation media simultaneously. This combination has resulted in a large number of diverse requirements. ISO/IEC 14478 is intended to address the presentation requirements of such diverse application areas as:

a) medical systems,

b) education/training,

c) virtual reality,

d) geographic information systems,

e) digital publications,

f) scientific visualization and data exploration,

g) entertainment,

h) realtime command control systems, and

i) simulation;

and such presentation and interaction techniques as:

j) animation,

k) simultaneous use of multiple media,

l) multimodal user interfaces,

m) realistic rendering (including various dimensionalities, such 2D, 2.5D, 3D, and incorporating various media, such as video, sound, and other non–visual data).

ISO/IEC 14478 provides a common underlying functional nucleus to support these application areas and presentation techniques, as well as future areas and techniques. PREMO also enables the use of interaction techniques appropriate for specific applications, such as those listed above.

PREMO provides a generic framework, into which various organizations or applications may place their specialised objects with specific behaviour, thereby enabling interoperability. In this sense, PREMO is intended to serve as middleware, coordinating multimedia components.

6.2 Extensibility

ISO/IEC 14478 is extensible in that it makes provisions for extending the functionality specified in the standard via a number of standardized mechanisms. In particular, additional components may be developed which respond to the needs of specific application areas. See also clause B in ISO/IEC 14478-2 for further details on the way PREMO objects may be extended.

Many aspects of ISO/IEC 14478 are extensible by an ISO–administered registration mechanism, so that a uniform description of the extension is available to all implementations.

6.3 Configurability

The need for configurability arises because different application areas have different demands on the task of presenting their data. PREMO embodies a configurable system design which offers a foundation from which specific requirements for object support and interrelationships can be realized. This configurable system design is based on the concepts of components and profiles (see clause 7.3.4 and clause 9). In a configurable system, profiles can be chosen according to the special needs of particular applications. The advantages of a configurable system design are:

a) Applications do not reference the whole system but only the specific component profiles they require. For example, an application might need only an audio or video profile.

b) When introducing new techniques, e.g., shading methods within a graphics system, or a special purpose graphics data storage, there is no need to implement a completely new graphics system for the realization of these new approaches. They can be integrated as new object types that fit within the existing foundation.

6.4 Incremental, separable development

ISO/IEC 14478 is described and structured in such a way that it can be developed incrementally. The chosen architecture provides a foundation for an evolvable family of standards.

6.5 Simplicity

Aspects, such as portability and maintenance, are greatly enhanced by keeping underlying concepts simple. Simplicity means that PREMO is based on a general architecture under which various sets of objects may be utilized. Objects are defined in terms of their externally visible behaviour, thereby hiding implementation details. Hierarchical structuring of and within objects is possible, thereby allowing more complex entities to be assembled from simpler parts.

6.6 Ease of use

PREMO is easy to use for at least the following classes:

a) end users (individuals or groups), who work with information processing applications based on PREMO;

b) programmers, who use PREMO components to build applications;

c) vendors, who develop, sell, and service implementations of PREMO; and

d) system administrators, who control and manage multimedia systems.

6.7 Other influences

6.7.1 Application development environment

PREMO takes advantage of an object–oriented design philosophy. This aspect is differentiated from different kinds of language bindings addressed below. Thinking in terms of objects leads to a natural description of the functionality of distributed multimedia systems entailing:

a) uniform mechanism to access passive or active entities;

b) the treatment, description, assignment, and modification of the attributes of objects as inherent information;

c) a clear understanding concerning the concurrent existence of objects that the user may freely select and modify; and

d) the definition of different objects with equivalent behaviour as instances of a common type.

Many benefits of the object–oriented approach taken by PREMO can be realized without requiring that applications be implemented in object–oriented or object–based languages. PREMO can be bound to both object–oriented and non object–oriented languages. However, the functionality of PREMO is defined so that the advantages of object–oriented environments — especially subtyping and inheritance and the ability to define mutual communication between application objects and multimedia objects — can be used. Subtyping and inheritance offer mechanisms to extend PREMO functionality and to adapt it to specific areas of applications, whereas mutual communication supports a stronger symmetry between input and output. Notwithstanding the above, one main goal for describing the functionality is that it is not limited to only object–oriented environments.

6.7.2 Execution environment

PREMO allows applications to take advantage of distributed environments. It allows distribution of functionalities across multiple processors. Where and when applicable, PREMO is compatible with, or is defined in terms of, other international standards.

6.8 Functionality

6.8.1 Introduction

PREMO supports the construction of, presentation of, and interaction with objects consisting of still computer graphics, moving computer graphics (animation), audio, text, still images, moving images (including video), images coming from imaging operations, and any other media type or combination of media types that can be presented.

6.8.2 Computer graphics

The objective of PREMO is to consider the evolving needs of the computer graphics and applications communities. Increasingly this means catering to the integration of computer graphics in multimedia applications. The underlying concepts and functionality of PREMO provide a visible route which the community can follow to take advantage of ISO/IEC 14478. This shall be accomplished by providing simple yet extensible functionality in a well–defined manner.

6.8.3 User interfaces

PREMO goes beyond the presentation and interaction requirements of simple graphical user interfaces by supporting the integrated use of multiple media, for example, through simultaneous presentations.

6.8.4 Dynamic interactive graphics

PREMO provides for real–time control and presentation of dynamic interactive computer graphics applications, where the graphics varies in real–time in response to external inputs. It allows an application to integrate dynamic computer graphics with other media.

6.8.5 Animation

PREMO provides for real–time control of media presentation, including the presentation of time–varying information, e.g., slide shows, smoothly moving objects. It allows an application to integrate animation with other media. PREMO provides mechanisms that can be used to create animated audio–visual applications.

6.8.6 Audio

PREMO provides for real–time control of both analogue and digital audio information presentation. It allows an application to integrate audio with other media.

6.8.7 Video

PREMO provides for real–time control of the presentation of both analogue and digital video information. This includes both single frame and time–varying video data. It allows an application to integrate video with other media.

6.8.8 Other and future media

PREMO supports media besides the ones listed in this clause. PREMO is extensible to support any media type or combination of media types that can be presented. Examples of such media include haptic and thermal sensory presentations.

6.8.9 Co–representations

PREMO supports co–representations. That is, the same information can be presented in several different ways from which the most appropriate is chosen. For example, PREMO allows a modeller to choose the most appropriate renderer from among those available. It allows information to be represented in a variety of media — for example audio and video — from which the most suitable can be chosen based on the application’s needs and the capabilities of available devices.

6.8.10 Cooperating applications

PREMO supports real–time exchange among applications cooperating in interaction with the same scene. This includes both co-operation between different media and the exchange of single media and multimedia objects.

7 Architecture of PREMO

7.1 Introduction

PREMO can be conceptually described in at least three different ways. These three architectural perspectives each have their own way of describing the parts of PREMO and their interrelations.

The three architectural perspectives are:

a) the standards perspective, which explains how PREMO is organized as a multi–part standard.

b) the functional perspective, which introduces the organization of PREMO as a collection of components, profiles, and object and non–object types. It also includes types related to non-presentation functionalities. Emerging ISO and other object technology standards — such as standardized object description techniques, object request brokers, traders, and object services — should be influenced by PREMO requirements and provide capabilities that PREMO can utilize.

c) the system perspective, which explains what systems may result from PREMO implementations and how different PREMO implementations shall interoperate. It explains what aspects of an environment should be brought under control of PREMO as opposed to transparent aspects (e.g., distribution). This architecture also allows precise formulation of portability and interoperability requirements for PREMO implementations.

7.2 The standards perspective

PREMO envisages a broad scope of functionality which cannot be covered by a single activity. Therefore, PREMO has been designed as a multipart standard. The first part (ISO/IEC 14478-1) explains key concepts, describes the overall architecture of PREMO, and specifies the common semantics for specifying the externally visible characteristics of PREMO objects in an implementation–independent way. The second part (ISO/IEC 14478-2) defines those objects that any conforming PREMO implementation shall support. The third part (ISO/IEC 14478-3) defines a Multimedia Systems Services component which provides an infrastructure for building computing platforms that support interactive multimedia applications dealing with synchronized, time–based media in a heterogeneous distributed environment. The fourth part (ISO/IEC 14478-4) defines a Modelling, Presentation, and Interaction component which is targeted at providing paradigm independent support for high–level modelling and presentation, enhanced by time control and interaction, and using various media.

Further parts are anticipated that will be appropriate for specific application areas. For example, a PREMO–based mapping from existing computer graphics standards will allow many existing applications to be integrated into multimedia presentations. Also, the development of toolkits and highly interactive and portable authoring tools for the generation and presentation of multimedia documents could be defined based on functionality specified by other international standards committees.

7.3 The functional perspective

7.3.1 Introduction

The functional perspective groups presentation and interaction functionality in terms familiar to an application programmer (e.g., aspects of time, geometry, etc.) and functionality necessary to achieve certain effects, where the programmer is possibly unaware of the underlying techniques used to achieve these effects (e.g., forms of anti–aliasing, colour representations, structuring of presentation data).

The functional architecture is restricted to a conceptual description of the functionality, leaving further detail or realization to the foundation and non–foundation components. It identifies the functional areas common to all media components, but possibly having different realizations in each environment.

The functional architecture explains the nature of the rules for components and profiles such that they can be combined (configured) and linked to other standardized or external non–standard components. The selection of components to assemble the functionality needed for a given application and in order to be able to run on a target platform is based on criteria such as:

— how components can interface to one another,

— how new components may reuse parts of older components, and

— how components can be realized on the corresponding part of the implementation platform.

7.3.2 Description techniques

PREMO functionality is described in terms of object behaviour. Each PREMO object is specified by giving:

a) a definition of its interface;

b) a description of the object's behaviour. Such a description specifies the object's visible behaviour, the effects of operations on the object's internal state, the output parameters of each operation request it is capable of receiving, and the input parameters of these requests[1].

The notational conventions used for object specifications are described in clause A.3 and Annex B.

[1] Descriptions of the objects' behaviour, using formal description techniques, are being developed separately and may be published as accompanying technical reports.

7.3.3 The object model

An object–oriented description technique is used to specify PREMO functionality. The object–oriented terminology and formalism constitute an object model in which objects and object types are composed and made to interact. The manner in which PREMO is described is independent of both the techniques used to construct PREMO implementations and the languages to which those implementations may be bound. PREMO can be bound to, and implemented in, both object–oriented and non object–oriented programming languages. The object model is described in clause 8.

7.3.4 Components

A component is a collection of profiles where specific profiles are defined to offer particular sets of functionality in the form of object and non–object types. Each profile is defined in terms of its dependencies on other profiles (either internal, i.e., dependencies on profiles belonging to the same component, or external, i.e., dependencies on profiles belonging to another component) and the object and non–object types it provides for other components.

The exact rules for components and profiles are given in clause 9 with the notational conventions described in clause A.7 of this part. These rules form the basis for the properties of configuration, customization, extensibility, and interoperability.

7.4 The system perspective

7.4.1 Configuring PREMO–based applications

The functionality of PREMO is realized by a set of object types. The PREMO architecture provides mechanisms for creating, using, and extending the standardized object types and non–object types.

NOTE — For example, in a PREMO implementation that provides traditional computer graphics functionality, there could be modelling and rendering functions with different capabilities. Modellers range from powerful geometric modellers which perform union and intersection operations on regular and free form 3D bodies to simple 2D graphics modellers. Renderers can include high performance renderers for photorealistic images and generators for an animated sequence of images from key frames as well as generators for 2D business graphics diagrams.

7.4.2 Distributed multimedia

Instances of PREMO object types are conceptually location independent. There are no inherent constraints on the location of these object instances, thus allowing a PREMO implementation to be easily distributed.

NOTE — For example, an application built using PREMO might create a multi–user interface in a virtual reality setting where several players are moving and manipulating within the same scene. Each player uses his/her own renderers with his personal viewing parameters to generate his/her view of the scene, hears his/her personal version of sound associated with the objects in the scene, and activates his/her own input devices to communicate with the common scene. If this example were realized in a distributed environment, different renderers might be installed as remote processes each with a copy of the scene. In this case, scene updates could be performed in such a way that the different copies of the scene remain consistent with each other.

7.4.3 Communication in PREMO

Since PREMO supports distributed applications, as well as multiple processor implementations, the invocation of PREMO operations may involve communication. Objects can learn of each other's existence and invoke each other's operations. Synchronization may be provided, since two objects could invoke operations on a third object concurrently. Communication among PREMO objects and between PREMO objects and their client applications requires the use of underlying support facilities that are not addressed in this standard.

8 Object model

8.1 Introduction

This clause describes the common semantics for specifying the externally visible characteristics of PREMO objects in an implementation–independent way. It specifies the common features that all conforming systems shall support. It includes a formal model of types, operations, and subtyping.

PREMO uses an object model to support design portability and reuse of object definitions. The use of an object–oriented design leads to a natural description and provides, in particular, a way for explaining extensibility and configuration aspects for PREMO objects. It should be noted that the description techniques used in this document focus on design and allow different bindings of an object's interface (i.e., the declaration of its operations) to programming languages, communication techniques, and implementations. Although the PREMO Object Model defines types and operations as concepts, systems that conform with the model need not provide objects that correspond to these concepts (e.g., if PREMO was implemented using a non object–oriented programming language, like Fortran).

It must be emphasized that the requirements described in this clause are valid for PREMO objects only, i.e., for objects defined in this or subsequent parts of ISO/IEC 14478. Even if PREMO is implemented in an object oriented environments, it is *not* mandated that *all* objects in this environment behave exactly as described in this clause, only those which are defined by PREMO.

8.2 Basic concepts

The PREMO Object Model is based on a small number of basic concepts: objects, object types, and subtyping. An *object* can model any kind of entity, e.g., a person, a ship, a document, a graphic segment, or a colour value. A basic characteristic of an object is its distinct identity, which is immutable, persists for as long as the object exists, and is independent of the properties and behaviour of the object.

Operations are applied to objects. Thus, for example, to determine the colour of a graphic segment, the colour operation might be applied to the segment object. As another example, in a windowing system a relationship between two windows may be defined as an operation *parent*, which, when applied to one window object, returns another window object. The operations associated with an object collectively characterize its behaviour.

Objects are created as instances of object types (e.g., person, colour, segment). An *object type* defines the behaviour of its instances by describing operations that can be applied to those objects. Types can be related to one another through *supertype/subtype* relationships.

State is required in an object system, because it captures internal information that may affect the outcome of operations. For example, an operation *setColour* might take a segment object and a colour object as input arguments and produce side effects on the latter object. State captures these side effects, and a subsequent application of the colour operation will presumably yield a result that differs from a previous invocation. In the PREMO Object Model, operations are used to model the external interface to state.

An object type can also be described in terms of *attributes*. Attributes represent a notational convention only; an attribute is functionally equivalent to declaring a pair of accessor and mutator operations, to retrieve the value of the attribute and to set the value of the attribute. As a further notational convenience, attributes may also be labelled as "retrieve only", which means that the mutator operation, though conceptually present, does not change the value of the attribute. In other words, the corresponding value can only be changed by the object itself, based on the object's internal state transition, or through other, dedicated operations.

8.3 Non–object types

Things that are not objects are called *non–objects*. These do not have an object reference, and therefore cannot be the controlling parameter for an operation request (see 8.6). Each non–object can be considered to belong to a type, called a non–object (data) type. This is analogous to objects being instances of types. Non–object types, however, do not belong to the PREMO object type hierarchy. Examples of non–objects are, e.g., an integer or a real number.

8.4 Object types

Objects support only certain operations. The object type defines these operations, and thus characterizes the behaviour of objects. Objects are created as instances of their object types, and, in the PREMO Object Model, objects shall not change their type.

Each operation has a *signature*, which consists of a list of input and output parameter types (see 8.6). The set of operation names with their respective signatures, defined for a type, is called the *interface* of that type, which is a distinct notion from the type itself. The interface includes signatures that are inherited from supertypes (see 8.7). The interface of a type can be applied to all instances of that type.

Types are arranged into a type hierarchy that forms a directed, acyclic graph. PREMO objects inherit from type *PREMOObject*, which defines the basic operations required for all objects in PREMO. Applications may introduce new types by subtyping from *PREMOObject* or its subtypes. Having a single root allows programs to declare a parameter that takes an object of any type as a value. The set of all object types is referred to as *OTypes*.

8.5 Object identity and object reference

In the PREMO Object Model, an *object reference* is an object name which reliably denotes a particular object. Specifically, an object reference identifies the same object each time the reference is used in a request. The type of the object may be inferred from an object reference.

Object references are represented in PREMO by opaque non–object types. For each object of type *T*, an object reference 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*.

An object reference referring to no object instances has a distinguishable value, which is referred to as *NULLObject*. Operations are available among those defined for PREMO objects in general to check whether a reference has a *NULLObject* value or not.

8.6 Operations

An *operation* describes an action that can be applied to an object, using parameters. An operation *invocation*, also called an operation *request*, specifies the operation and parameters, possibly causing results to be returned.

The consequences of a request can include:

a) an immediate set of results;

b) side effects, manifested as changes in the state of the object; and

c) exceptions.

Formally, an operation Ω has the signature:

$$\omega : (x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n) \rightarrow (y_1 : \rho_1, y_2 : \rho_2, \ldots, y_m : \rho_m)$$

where ω is the name of the operation. The operation signature specifies $n \geq 1$ input parameters with names x_i and types σ_i and $m \geq 0$ output parameters with names y_j and types ρ_j.

In the PREMO Object Model, operations are always specified with a special parameter called the *controlling parameter*, which is used to control the way operations are dispatched: its role is to differentiate among several possible implementations of an operation with the same name and signature (see 8.7.4). For discussion purposes in this clause, we assume that the first parameter x is the controlling parameter, although this choice is not required by the model. Each object type $T \in OTypes$ has a set of operations:

$$Ops(T) = \{\Omega_1^T, \Omega_2^T, \ldots\} .$$

An operation is part of the interface of the type *T* of its controlling parameter and of all subtypes of *T* (see 8.7). An operation is *defined on* the type of its controlling parameter; e.g., Ω is defined on σ . All operations defined on a type have either distinct names or, if names are identical, have a distinct signature. In the PREMO Object Model an operation is defined on a single type (the type of the controlling parameter), so there is no notion of an operation independent of a type, or of an operation defined on two or more types.

In the PREMO Object Model, operations can only be defined on object types, not on non–object types. The controlling parameter type shall be an element of *OTypes*. All other parameters are restricted to be non–object data types. Note that references to objects are expressed by non–object data types; hence, this restriction does not introduce a limitation on functional expressiveness.

An operation may have side effects. The PREMO Object Model does not specify anything about the execution order for operations. For example, whether or not callers issue requests sequentially or concurrently and whether or not requests get serviced sequentially or concurrently is not part of the PREMO Object Model. Although the PREMO Object Model does not specify support for multi–process synchronization, it does allow several styles of operation request semantics (see 8.9).

In the PREMO Object Model operations are not objects, neither are requests (i.e., operation invocations).

NOTE — In line with a common practice, the object issuing an operation request is also referred to as a *client* (of the object whose operation is invoked).

8.7 Subtyping and inheritance

8.7.1 Overview

Subtyping is a relationship between types, based on their interfaces. It defines the rules by which objects of one type are determined to be acceptable in contexts expecting another type. *Inheritance* is a mechanism for reuse; as a notational convenience a type may be defined in terms of another type. This clause defines the two concepts separately, but then explicitly states how they are related in the PREMO Object Model.

8.7.2 Subtyping

The PREMO Object Model supports subtyping for object types. Intuitively, one type *S* is a subtype of another type *T*, if *S* is a specialization or a refinement of *T*. Operationally, this means that any object of type *S* can be used in any context that expects an object of type *T*. In other words, objects of type *S* are also of type *T*. Subtypes can have multiple supertypes, with the implication that an object that is an instance of a type *S* is also an instance of all supertypes of *S*. The relationships between types define a type hierarchy, which can be drawn as a directed, acyclic graph (see Figure 1; see also clause C for the graphical conventions used in the figures in ISO/IEC 14478).

An object is a *direct instance* of a type *T*, if it is an instance of *T* and not an instance of any subtype of *T*. The PREMO Object Model restricts objects to be direct instances of exactly one type. That one type is the *immediate type* of the object. The PREMO Object Model has no mechanism to change the immediate type of an object.

An object type *T* is an *immediate supertype* of *S*, if it is a supertype of *S*, and there is no type *Q* such that *Q* is a supertype of *S* and *T* is a supertype of *Q*.

In the PREMO Object Model, the type designer is required to declare the intent that a type *S* is a subtype of *T*. Formally, if *S* is declared to be a subtype of *T*, then for each operation $\Omega^T \in Ops(T)$ there exists a corresponding operation $\Omega^S \in Ops(S)$ such that the following conditions hold:

a) the names of the operation match;

b) the number and types of the parameters shall be the same (except that the controlling parameter types shall differ);

c) the number and types of the results shall be the same.

Thus, for every operation in *T* there shall be a corresponding operation in *S*, though there may be more operations in *Ops(S)* than in *Ops(T)*.

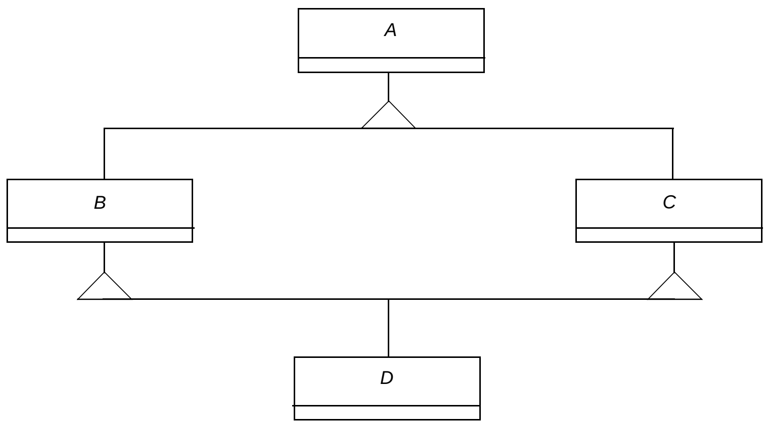


Figure 1 — Type graph

The set of all supertypes of a type, including the recursively defined supertypes, plus the type itself, form the *type graph* of the type. For example, if the object type *D* is a subtype of *B and C*, and both *B* and *C* are subtypes of *A* (see Figure 1), the set consisting of *A, B, C*, and *D* forms the type graph of *D*. The PREMO Object Model does not provide a name–conflict resolution mechanism, nor does it allow subtypes to redefine inherited operation *signatures* (although it is allowed to add operations in the subtype with identical name but a different signature).

The above typing rules prevent two types that have an operation with the same name but different signatures from having a common subtype. *Supertypes* are used to characterize functionality that is common to their subtypes.

8.7.3 Inheritance

Inheritance is a notational mechanism for defining operation reuse. If a type *S* inherits from another type *T* then the definition of *S* inherits all the operations of *T* and may provide other operations. Intuitively, inheritance means that the operations defined for *T* are also defined for, and can be used by, *S*. When inheriting an interface, type *S* may include its own implementation for an operation inherited from *T* or, alternatively, may rely on the implementation provided with *T*. The choice among the various implementations of an operation when the operation is invoked is referred to as *operation dispatching*, as is described in more detail in 8.7.4.

If the interface *P* inherits from *Q*, *P* may also be referred to as *derived interface*, whereas *Q* is the *base interface* (of *P*).

The PREMO Object Model relates subtyping and inheritance. If *S* is declared to be a subtype of *T*, then *S* also inherits from *T*.

The PREMO Object Model supports *multiple inheritance*, which is a notational mechanism for the definition of inheritance of multiple types (see 8.7.2).

Subtyping is a relationship between types. Inheritance can be applied to both interfaces and implementations; i.e., both interfaces and implementations can be inherited.

The PREMO Object Model does not provide a conflict resolution mechanism in the case of name clashes.

8.7.4 Operation dispatching

When an operation request is issued, a specific operation implementation is selected for execution. This selection process is called *operation dispatching*.

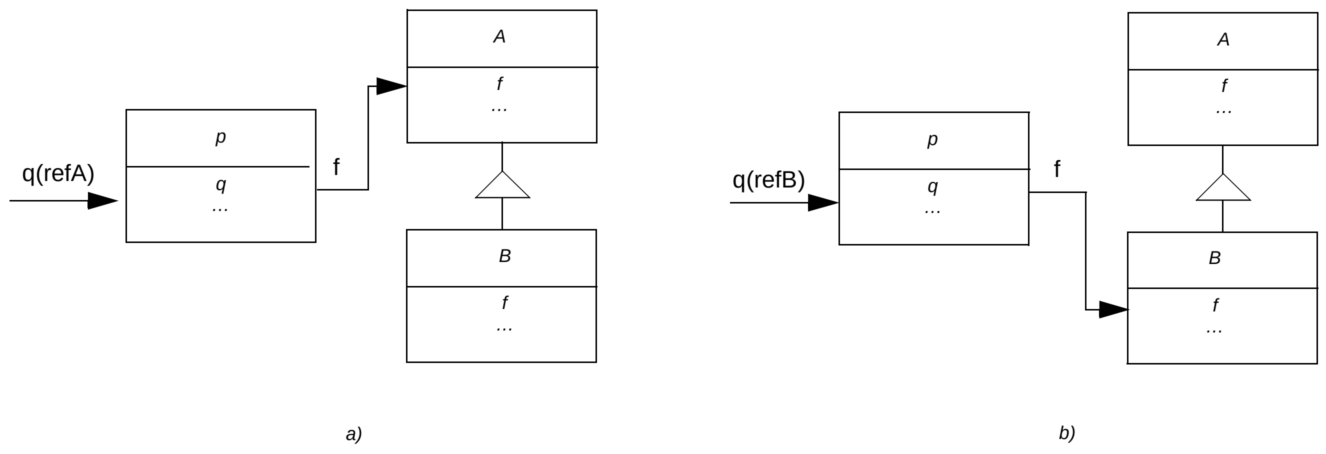


Figure 2 — Example for operation dispatching

The selection mechanism is based on the name and the signature of the operation: the name of the operation and the non–object types used as actual parameters used at the operation invocation (except for the controlling parameter) shall match the signature of the operation specified for the object whose operation is invoked. An exception to this rule applies for parameters of an object reference type. If the signature of an operation is:

$$\omega : (x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n) \rightarrow (y_1 : \rho_1, y_2 : \rho_2, \ldots, y_m : \rho_m)$$

and, say, σ_i is of type *RefT* where *T* is an object type, and *S* is a subtype of *T*, the actual parameter x_i is permitted to be of type *RefS* when invoking the operation ω. Also, if, say, ρ_j is of type *RefQ* where *Q* is an object type, and *R* is a subtype of *Q*, the actual returned value for parameter y_j is permitted to be of type *RefR* when the operation ω returns.

The process of selecting which operation to invoke is based on the type of the object supplied as the controlling parameter of the actual call. The operation of the given name defined on the immediate type of the controlling argument is chosen for invocation. In some cases this choice can be done at compile time with no loss of flexibility, whereas in others it must be delayed until execution time.

The effect of this selection mechanism is as follows. From the perspective of the object which requests an operation, only interface inheritance is of importance. However, from the perspective of the object which implements the operation, both interface inheritance and implementation inheritance are of importance. In the PREMO Object Model the implementation of the derived interface *supersedes* (or *overloads*) the implementation of the base interface. For example, if type *S* is a subtype of *T*, and the interface of both *S* and *T* contains the operation ω, then for all instances of type *S* the implementation of operation ω implemented for *S* will prevail, even if used in an operational context that expects an object of type *T*.

NOTE — An example will clarify the selection mechanism (see also Figure 2). Let *A* and *B* be PREMO object types, such that *B* is a subtype of *A*; furthermore let the operation *f* be defined both in *A* and in *B*, i.e., both types provide an implementation for this operation. As a convention, the two implementations will be denoted by *A.f* and *B.f*, respectively. Let *P* be another PREMO type, and *p* an instance of this type; finally let the operation *q* be defined for *P*. The parameter list of *q* should include one of type *RefA*. Finally, let us suppose that the implementation of *P.q* issues an operation request on the operation *f* using its argument of type *RefA*. The operation dispatching rules determine which of the possible implementations of *f* (i.e., *A.f* or *B.f*) will be invoked. Following the rules above, the operation *p.q* may be invoked with actual parameters of non–object types of both *RefA* and *RefB*. In the former case the implementation of *p.q* invokes the operation *f* on the referred object instance: the selected implementation should be *A.f* (case ‘a’ in Figure 2). In the case when *p.q* is invoked with an actual parameter of type *RefB*, the selection mechanism described above will choose *B.f* (case ‘b’ in Figure 2).

In the case an operation wishes to have a finer control over the selection mechanism, it can inquire the immediate type and the type graph of the object referred to by its argument (see clauses 7.2 and 9.4 of ISO/IEC 14478-2), and it can use the `cast` facility, (see 8.11). In the example above, if the implementation of *p.q* needs to access *A.f* in any case, it should inquire the type of the object the actual parameter refers to and, if necessary, should generate a new object reference of type *RefA* with the help of the `cast` facility.

These rules do not disambiguate all possible situations in the case of multiple inheritance. For example, let *B*, *C*, and *D* be types such that *D* is a subtype of both *B* and *C* (see Figure 1). Let *B* and *C* have an implementation for the operation *f* (with identical signature, except for the respective controlling parameter). Finally, let *D* simply inherit the operation *f*, without providing an implementation. In this case, when using an object reference *RefD*, there is no unambiguous choice between the two possible implementations of the operation *f*. In the PREMO Object Model, an exception is raised when no unambiguous choice exists. Note that some environments may also offer compile time checking for such cases.

8.8 Abstract Types

An *abstract type* in PREMO is a non–instantiable type, i.e., if *T* is defined to be abstract, then no PREMO object may be a direct instance of *T* (see 8.7.2). Usage of abstract types makes the specification of object type hierarchies clearer and more succinct, allowing the specifications to “abstract out” identical interfaces without forcing implementations to provide realizations for these objects in isolation.

8.9 Operation request semantics

The external behaviour of PREMO objects is based on the operations defined for the object. Requests for operations provide the only means of information transfer among PREMO objects. All requests are delivered *at most* once to the object.

Internally, an object has a finer control over the actions it has to perform to service the request. Conceptually, each operation has an *operation receptor*, and an operation request amounts to putting a request into this receptor. Each receptor of an operation may be in one of three modes: *synchronous*, *asynchronous*, or *sampled*. This mode is specified as part of the operation specification and is immutable during the lifetime of the object. The default mode is synchronous (see also clause A.3 for the notational conventions used to define operations). The detailed semantics of the three modes are as follows:

— The operation receptor is *synchronous*. This means that the request is placed in the operation’s receptor and the caller is then suspended until the callee has serviced the request. Data may be associated with the request, and the request may have return values. The PREMO Object Model makes no assumptions on the order of servicing of these requests if there are several waiting to be serviced.

— The operation receptor is *asynchronous*. In this case, the request is placed in the operation’s service receptor but the caller is not suspended. Data may be associated to the request, but the request shall not have return values. The PREMO Object Model makes no assumptions on the order of servicing of these requests.

— The operation receptor is *sampled*. In this case, the receptor may hold *at most one* request; if several requests arrive at the receptor without being serviced, their respective requests overwrite one another. The caller is not suspended. Data may be associated with the request, but the request shall not have return values.

While suspended, an object can receive operation requests from other objects. These requests are managed in accordance with the behaviour described above. A suspended object may also carry out internal processing, but shall not access information related to its own operation receptors.

NOTE — The ability of objects for continuing internal processing while being suspended is important for the support of multimedia synchronization (see 7.9.1 of ISO/IEC 14478-2).

Apart from having several outstanding service requests on the *same* operation of an object, there may be several operation requests on *different* operations waiting to be serviced. In this case, the object chooses one of these requests non–deterministically. An object has also means to control which requests it wishes to service, depending on its internal state.

An object instance can also issue an operation invocation to one of its own operations. In this case, all receptors are bypassed, and the implementation of the operation is immediately invoked, i.e., deadlock shall not occur.

A subtype may provide a new implementation for an inherited operation. However, the operation request mode of an operation shall not be changed in the subtype.

8.10 Protected operations

An operation may be declared to be *protected*. The name and signature of a protected operation is part of the interface of the object type just as with any other operations. However, for any object instance, a protected operation can only be invoked by the instance itself; no other object instance can invoke such an operation. Being part of the interface of the object, it is possible to modify the behaviour of a protected operation in the subtype of the object, thereby specializing the object's behaviour. The fact whether an operation is protected or not is an immutable characteristics of the operation during the lifetime of the object. Also, a subtype cannot declare an inherited protected operation to be unprotected.

Attributes can also be declared to be protected; this simply means that the accessor and mutator operations for attributes are protected.

8.11 Object and object reference life cycles

PREMO objects are created and destroyed by facilities which are part of the boundary between PREMO and its environments.

Objects may be *created* through the use of the `create` facility. If successful, this facility returns an object reference to an object of the requested type. If the creation is successful, it can be safely assumed that this object reference refers to an existing object instance of the type specified by the argument of the `create` facility. Using this object reference, operations on the object may be immediately requested. This facility shall also invoke the (protected) *initialize* operation of the newly instantiated object (see clauses 7.2 and 9.4 in ISO/IEC 14478-2), which is available for all PREMO objects; this invocation is done prior to the return of the object reference to the caller. Parameters of the `create` facility include the necessary initialization parameters. Creation of an object might be unsuccessful; it is environment dependent how the caller of the facility is notified about this and it is the responsibility of the caller to check the success of object creation.

NOTE — The `create` facility may, for example, return *NULLObject* in the case of an unsuccessful object creation.

Objects may also be *copied* through the use of the `copy` facility. `copy` also returns a reference to a new object, but it also receives a reference to an already existing object. The newly created object has the following characteristics:

— It has the same immediate type as the object referred to by the argument of `copy`.

— The *initialize* operation is not invoked on the new object; however, another (protected) operation is defined on all PREMO objects, called *initializeOnCopy*, which is invoked by the `copy` facility. The signature of *initializeOnCopy* is void of any arguments.

— If `copy` is *shallow*, all attributes of the new object are set to the values as retrieved from the object referred to by the argument of `copy`.

— If `copy` is *deep*, all attributes of the new object, except object references, are set to the values as retrieved from the object referred to by the argument of `copy`. For object reference attributes, a deep copy is made for all referred objects (recursively) and the corresponding reference attributes of the newly created object will refer to these new objects.

NOTE — If the implementation of PREMO is based on a distributed environment, the caller of the `create` and `copy` facilities may also control "where" the new object is created.

An object reference is of a non–object type and, as such, can be included as a parameter for other operation requests. The definition of operations shall specify whether the reference is used directly (in which case, conceptually, each such call constitutes the creation of a new instance of these object references), or whether new objects and object references should be created by a (deep or shallow) copying of the referred object. Whether this copying is performed by the caller (i.e., before the invocation of the operation) or the callee (i.e., as a first action of the operation being invoked) is dependent on the implementation environment and the programming language. This part of the operation specification is conceptually part of the signature of the operation, i.e., subtypes cannot change this in the case of overloaded operations.

NOTE — An implementation may have a finer control over argument copying. For example, it may decide to copy part of a structure only, or to stop recursive copying at some level. These decisions may be based on the semantics of the operation and/or the particularities of the implementation. PREMO standardizes the minimum level of copying only, i.e., most of the time only shallow copy is required.

The facility `cast` can also be used to create new object references. This facility receives an object reference and an object type as parameter, and creates an object reference which still refers to the same object instance, but of the type given as parameter. PREMO requires that, if type *A* is a subtype of *B*, *a* is an instance of *A*, and the reference to *a* is given in the form of *RefB*, then a new reference, of type *RefA* and referring to the same instance, can be created using *a* and *A*.

NOTE — Some object environments may offer richer casting facilities, but PREMO does not rely on those. An example for the usage of the `cast` facility is when the caller receives an object reference to a type which is the supertype of the expected type. By inquiring the type graph of the object and using the `cast` facility using a type appearing on this type graph, the caller may create an object reference with the appropriate (sub)type.

Object references may conceptually be destroyed using the `destroyReference` facility. As a result of this conceptual destruction, the object instance is no longer accessible *through this object reference*. Other object references created, e.g., by argument passing or the usage of `cast`, may still be used to refer to the object instance. It is incumbent on the application or the implementation to delete object references which are no longer in use. Objects are automatically destroyed when no valid reference exist for them any more, but not before. In other words, the PREMO specification makes the assumption that all valid references also remain valid.

The specification of the facilities `create`, `copy`, `cast`, and `destroyReference` is independent of any particular implementation language, development environment, or execution environment. For integration into a programming environment, these facilities shall be realized through an appropriate language binding.

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

8.12 Exceptions

When servicing a particular request, a PREMO object may detect error conditions which make it impossible to fully service the request. In such a case the PREMO object shall, instead of servicing the request, raise an exception. Exceptions are defined as special data structures which may convey additional information on the error condition leading to the raise of the exception.

Details of how exceptions are raised are part of the boundary between PREMO and its environment (see also 8.11). The detailed specification depends on a particular implementation language, development environment, or execution environment. There are, however, some simple rules which shall always apply:

— If an exception is raised by an object, the state of the object shall not change.

— Different error conditions result in different exceptions; the set of possible exceptions for a specific operation is part of the detailed specification of the operation. When operations are redefined through subtyping, the set of possible exceptions may be extended with new exceptions.

— Facilities shall be available for the caller of an operation to detect whether an exception has been raised when the operation was executed or not, as well as to access the information stored in the exception data structure.

A special case is when an error condition occurs through the *initialize* or the *initializeOnCopy* operations of an object, i.e., when the object is created (see 7.2.1 of ISO/IEC 14478-2). In such a case the corresponding exception is raised by the facilities managing the object life cycle (see 8.11), the object is not created, and the `create` or the `copy` facilities shall return a *NULLObject*.

9 How PREMO components are described

Fundamentally, a component in PREMO is a set of related object types and non–object types that comply with the PREMO Object Model. Components organize these object and non–object types in terms of *profiles*, whereby some set of the types defined in the component are collected together for a particular view of their usage. A profile may be tailored towards a particular constituency or application domain, for example.

A component may contain one or more profiles, one of which shall be designated as the *basic profile* for the component. The basic profile of a component represents the minimal and hence mandatory set of types provided by the component implementation. All other profiles defined within the component shall be defined with respect to the component's basic profile.

The specification of a profile shall make explicit the dependencies that the profile has with respect to other profiles within its own component and with profiles defined in other components. These dependencies between profiles is expressed as follows.

a) A profile ***P*** belonging to component ***A*** may depend on profile ***Q*** of the same component if there are object types in ***P*** that are either:

1) subtyped from object types defined within ***Q*** (*type dependency*), or
2) whose behaviour depends on operations defined by object types in ***Q*** (*service dependency*).

This form of dependency is referred to as *internal dependency*.

b) A profile ***P*** belonging to component ***A*** may depend on profile ***R*** of component ***B*** if there are object types in ***P*** that are either:

1) derived from other object types defined within ***R*** (*type dependency*), or
2) whose behaviour depends on services provided by object types defined within ***R*** (*service dependency*).

This form of dependency is referred to as *external dependency*.

The various possible dependencies are non–exclusive; a component profile may have internal and external dependencies that may be in terms of both type and service dependencies.

The specification of a profile also includes the list of types which can be used to resolve type or service dependencies by other profiles or by applications in general. A profile can thereby restrict the usage of a type to, e.g., as a service provider only, i.e., the operations of the type are available for operation requests, but no subtyping of this type is possible. The PREMO specification also includes restrictions describing which part of the full PREMO object hierarchy can be used to resolve type dependencies (see 7.5 of ISO/IEC 14478-2).

NOTE — Language bindings to PREMO shall provide details on how these notions are mapped onto a particular programming environment. In some cases the programming environment may not make it possible to enforce the difference between service and type dependencies.

The notational conventions used for component and profile specification are defined in clause A.7.

The profile specification of a PREMO component makes provision for PREMO implementations to offer automatic configuring mechanisms. Such mechanisms may allow for an implementation of a component and/or a profile to interoperate with other component implementations.

Annex A
(normative)
Notational conventions

A.1 Type declarations

The notation:

$$A, B : Type$$

is used to denote that *A* and *B* are of type *Type*. In this notation, *Type* may be either a non–object or an object type. For example, the declaration

$$InvalidReference, InvalidOperations : Exceptions$$

declares two values of type *Exceptions*.

Constant values may be specified as follows:

$$A : Type \mid A = Value$$

The symbol '$\mid$' separates the type specification and the constant definition. For example, the declaration

$$A : \mathbf{Z} \mid A = 12$$

declares *A* to be a constant of an integer type with a constant value of 12.

A.2 Data type definitions

The definitions of operations in PREMO rely heavily on the conventions in this clause. Abstract data types are constructed from basic data types with a number of data type constructors. Values of these types for various programming languages are determined by the PREMO language binding standards.

A.2.1 Simple data type definitions

The simplest data type definition is exemplified by:

$$ConstraintOp ::= Equal \mid NotEqual$$

This lists the values the data type has, in this case *Equal* and *NotEqual*. Values are separated by the symbol '$\mid$'.

NOTE — This data type construction is often referred to as "enumeration" in various programming languages.

Type synonyms are sometimes used to make specifications more readable. For example, the definition:

$$State == Integer$$

means that, in what follows, the name *State* may be used as a synonym for *Integer*.

Data types which consist of sets of values of another type are defined using the powerset constructor, 'IP', for example:

$$NameSet == \mathbb{P}\, GenName$$

means that values of type *NameSet* consist of sets of values, each of type *GenName*.

Data types which consist of subranges of other data types are defined using a set comprehension construct. The general form of this construct is:

$$TypeName == \{\texttt{declarations} \mid \texttt{predicate} \bullet \texttt{expression}\}$$

The values of *TypeName* are just those values of ‘expression’ which satisfy the condition expressed by ‘predicate’. The ‘declaration’ part declares the types of the variables used in the predicate and expression parts. For example, to define a tuple whose components are restricted values of other types:

$$WCOrd == \mathbf{R}$$
$$PatSize == \{x, y : WCOrd \mid x > 0 \land y > 0 \bullet (x, y)\}$$

This states that the values of the data type *PatSize* consists of pairs of values each of type *WCOrd* $\times$ *WCOrd* which satisfy the predicate $x > 0 \land y > 0$. This approach is used in preference to the equivalent approach of defining a subtype of *WCOrd* for strictly positive ordinate values and then defining *PatSize* as a Cartesian product on this data type.

Another example is:

$$WCOrdPos == \{r : WCOrd \mid r > 0 \bullet r\}$$

This declares *WCOrdPos* to be a subtype of *WCOrd*. *WCOrdPos* consists of those values of the data type *WCOrd* which are positive.

For simple cases, shorthand forms of the declarations are used. If the predicate is omitted, the default predicate *true* is assumed. If the expression is omitted, the default is the characteristic tuple of the declaration part (the tuple of declared variables in the order in which they are declared). For example, the type *WCOrdPos* could be written as:

$$WCOrdPos == \{r : WCOrd \mid r > 0\}$$

Here the expression is omitted and the characteristic tuple of the declaration part *(r)* is assumed.

Functions are defined using the notation:

$$Selects == SelectCritType \rightarrow SelectCrit$$

This defines *Selects* as a function from values of type *SelectCritType* to values of type *SelectCrit*.

The set of all functions mapping the set *X* to set *Y* can be expressed as:

$$X \rightarrow Y$$

Where it is necessary to define a function whose source data type is the subtype of some other data type, a further extension of the predicate notation is used. For example, the notation $(i, j) \rightarrow m$ means that the value m is associated with the ordered pair (i, j) . Also:

$$Matrix23 == \{i, j : \mathbf{N}, m : \mathbf{R} \mid i \leq 2, j \leq 3 \bullet (i, j) \rightarrow m\}$$

defines a subtype of:

$$Matrix == \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{R}$$

which describes a matrix of dimension $p \times q$. For this*Matrix23* the first index of the source is restricted to the values $\{1, 2\}$ and the second to the values $\{1, 2, 3\}$.

A.2.2 Constructed type definitions

In order to describe some of the PREMO operations in a concise manner, it is convenient to define data types which may take any of the values of other types.

A.2.2.1 Tuples

Ordered tuples are defined using the Cartesian product constructor ‘×’. For example:

$$Constraint == String \times ConstraintOp$$

defines values of the data type *Constraint* to consist of a pair of values, the first of type *String*, the second of type *ConstraintOp*.

A.2.2.2 Discriminated unions

One such construction is the (discriminated) union. An abbreviated example is:

$$\begin{aligned} IntegerValues ::= {} & \mathtt{short} \langle\langle Short \rangle\rangle \\ & | \ \mathtt{long} \langle\langle Long \rangle\rangle \\ & | \ \mathtt{unsignedShort} \langle\langle UShort \rangle\rangle \end{aligned}$$

The definition states that the values of the data type *IntegerValues* are either of the type *Short*, *Long*, or values of type *UShort*. The constructor names ‘`short`’, ‘`long`’, ‘`unsignedShort`’ are chosen to describe usage of the type following. This is done because in some cases, the type name alone either does not provide an adequate explanation, or provides an ambiguous explanation. Types defined in this way are disjoint unions of the component types. A given value of the type corresponds to a value in only one of the component types. Thus in the example here, a value of the data type *IntegerValues* arises from only one of the component data types *Short*, *Long*, and *Ushort*.

The main usage of such union types is exemplified by the following:

$$IntegerTypes ::= SHORT \mid LONG \mid UNSIGNEDSHORT$$
$$Integer == IntegerTypes \rightarrow IntegerValues$$

The type *Integer* is a function from the source data type *IntegerTypes* to range data type *IntegerValues*. The source type denotes the names of various variants of integers and the data type *Integer* associates a value with each of these integers.

A.2.2.3 Sequences and arrays

The constructor ‘seq’ defines a type whose values consist of a sequence of values of some other type, for example,

$$CharString == \mathrm{seq}\ Char$$

defines values of type *CharString* to consist of sequences of values each of type *Char*. The constructor ‘seq’ allows sequences of length 0. An additional family of sequence constructors is provided by the following forms (for an arbitrary type *X* and non–negative integer *n*):.

$$\mathrm{seq}_n X == \{ f : \mathrm{seq}\ X \mid \#f \geq n \}$$
$$\mathrm{seq}_e X == \{ f : \mathrm{seq}\ X \mid \#f \bmod 2 = 0 \wedge \#f \geq 1 \}$$
$$\mathrm{seq}_o X == \{ f : \mathrm{seq}\ X \mid \#f \bmod 2 = 1 \wedge \#f \geq 1 \}$$

where the operator # delivers the length of the sequence.

Finally, a family of array constructors is defined with the following forms (for an arbitrary type *X* and size *i*):

$$\text{array}_i\, X == \{f : \text{seq}\, X \mid \#f = i\}$$

where *i* is a non–negative integer.

Specific sequences and arrays will also be denoted in the text as:

$<Element_1, Element_2, ..., Element_n>$

A.3 Object type definitions

Object types are described in term of their inheritance relationships and their operations. Type specifications are enclosed in *type schemas*:

```
┌══ Type Name ════════════════════════════════════

  SuperType1 redef (RedefinedOperation1, RedefinedOperation2)
  SuperType2

  short description of the type in natural language

 │ attribute1: AttributeType
 │ attribute2: AnotherType                          [Retrieve Only]
 ├────────────
 │ description of the attribute(s) in natural language

 ┌─ operation ─────────────────────────────────────
 │ parameter1_in: TypeOfParameter1
 │ parameter2_out: TypeOfParameter2
 │ parameter3_in: TypeOfParameter3                  [Shallow Copy]
 │ parameter4_in: TypeOfParameter4                     [Deep Copy]
 │ exceptions: {ExceptionValue1, ExceptionValue2}
 ├────────────
 │ behavioural description in natural language
 │
 │ Exceptions raised:
 │         ExceptionValue              Description.
 └─────────────────────────────────────────────────

└══ Type Name ════════════════════════════════════
```

The name of the type appears on the top and the bottom lines of the schema; these lines enclose the set of inheritance relationships and operation definitions which constitute the type specification. If the type is defined to be abstract (see 8.8), the keyword *abstract* appears as a subscript of the type name in the top enclosing line.

A type schema encloses the following units:

— *inheritance relationships*: this follows the top enclosing line, and is a list of PREMO types, one per line, specifying the set of immediate supertypes for the newly defined type. A simple listing of type names indicates that all operations defined in these supertypes are inherited by the new type without change; if the new type provides new implementations for some operations, these shall be listed between a pair of brackets, and this list shall be preceded by the keyword 'redef' (see the schema above).

— *short description of the type*: a short description, comment, etc., which applies to the type as whole, in English prose. This unit is optional.

— *attributes*: a separate sub–schema shall be used for attributes, which is functionally equivalent to declaring pairs of accessor and mutator operations, to retrieve the value of the attribute and to set the value of the attribute, respectively. Each attribute shall be defined with its type. Exceptions may also be present in this sub-schema, denoting exceptions raised when setting the corresponding attribute value. The type name in an attribute may be followed by the remark [Retrieve Only], which means that the operation to set the attribute, although conceptually present, does not change the value of the attribute. In other words, although the attribute value may be changed by the object itself depending on its state and semantics, the value cannot be set by other objects, or only through dedicated, and separate, operations which are to be defined explicitly.

— *operation schemas (one per operation)*: each operation schema contains the name of the operation on the top line of the schema, followed by the signature of the operation, the semantic description of the operation, and the list of exceptions the operation may raise.

- The signature consists of a list of parameters, each with its type (using the declaration notation, see clause A.3). The keywords *in* or *out*, appearing as a subscript to the parameter name, denote whether the parameter is an input or an output parameter, respectively. Note that the controlling argument (see 8.6) does *not* appear in this list.

  The type specification of an argument may be followed by the remark [Deep Copy] or [Shallow Copy], in the case the type is an object reference type. These remarks specify whether the object, referred to by the argument, shall be copied on the call or not and, if yes, whether the copy is deep or shallow (see 8.11). If no remark is present, the object is not copied, and the value of the object reference shall be used by the operation.

  The list of parameters is followed, if applicable, by a set of exceptions which can be raised by the operation.

- The semantics description of the operation, in English prose.

- The semantics of each exception which can be raised by the operation. Unless stated otherwise the data assigned to exceptions are empty; if this is not the case, the detailed information on the returned data is part of the semantics of the exception.

The request semantics of the operation (see 8.9) appears in the name of the operation in the operation schema: a character *a* or *s* in the subscript of the name denotes an asynchronous or sampled operation, respectively. If none of the two character appears as subscript, the operation is synchronous.

The name of the operations may be preceded by the symbol ‘Ξ’ which denotes that the operation is protected (see 8.10 for the definition of protected operations).

There are some other, more succinct ways of defining operations, too. These are further described in clause A.6.

A.4 Definition of finite state machines

The behaviour of objects in PREMO are sometimes defined in terms of states and state transitions of finite state machines. This clause defines the notational conventions used in the PREMO Standard for the specification of finite state machines.

State transitions are defined through state transition tables:

To: / *From:*	*State1*	*State2*	*State3*
State1	Y	I	Y
State2	N	N	I
State3	Y	Y	Y

The symbol in the i^{th} row and j^{th} column can be either ‘Y’, ‘N’, or ‘I’, and the meaning is as follows.

— ‘Y’ means that a transition from state *i* to state *j* is possible, and this transition can be requested by a client;

— ‘I’ means that a transition from state *i* to state *j* is possible, but this transition cannot be explicitly requested by a client, i.e., only the object may perform such a state transition internally;

— ‘N’ means that the state transition from state *i* to state *j* is not allowed.

The name of the initial state is underlined.

States are specified in the functional specification of the objects using constant with integer values (although type synonyms may be used to make the specification more readable). This makes extensions of finite state machines easy to describe.

A.5 Reference to operations and objects

In some cases, e.g., to make the behavioural descriptions or various examples more concise, it is necessary to have a clear notation to refer to an operation of an object type or an object instance. In PREMO, the following notation is used.

— If the symbol *A* refers to an object type, an object instance, or an object reference (depending on the context), and the symbol *f* refers to the name of an operation, the symbol *A.f* will be used to denote this operation on *A*. If necessary, input and output arguments of the operation may also be given by listing them in a pair of brackets, each name followed by the ‘in’ and ‘out’ subscripts to denote whether it is an input or output argument respectively, followed by a colon and the corresponding value. This convention is analogous to the one used in the type schemas. For example:

$$A.f(arg\_one_{in} : 1234, arg\_two_{in} : 3.456, arg\_three_{out} : output\_value)$$

denotes an operation with two input arguments (*arg_one* and *arg_two*) and one output argument (*arg_three*). The operation is invoked by assigning the values *1234* and *3.456* to the two input arguments, respectively, and the value of the output argument is fed into *output_value*. The names of the arguments appearing between brackets shall be present in the operation specification schema of the object: the order of the argument is not significant. It is not necessary to list all the arguments, only those which are relevant in the context. Finally, if no argument notation is required for the context, the pair of brackets can be missing altogether.

— If the symbol *A* refers to an object type, an object instance, or an object reference (depending on the context), and the symbol *a* refers to an attribute of the object, *A.a-set* and *A.a-retrieve* refer to the mutator and the accessor functions, respectively. The symbol *A.a* may also be used as a shorthand for *A.a-retrieve*. Because *A.a-set* may have only one argument with an obvious argument name, it is not necessary to add the name or the argument if the bracketed notation is used, i.e.,

$$A.a\text{-}set(6.789)$$

is acceptable, although the notation:

$$A.a\text{-}set(a_{in} : 6.789)$$

is also correct (though redundant).

In the behavioural description of objects, the following notational convention is also used. Types and instances of those types (i.e., the actual active objects) are distinguished as follows. If *Foo* is an object type or a generic type definition, the type is referred to as the “*Foo* object type” or “*Foo* type”, while a specific instance of a type is referred to as a “*Foo* object”.

A.6 Shorthands for operation specifications

Using the notations for the references to operations and for finite state machines it is possible to define alternative ways of specifying operations in type schemas; these are presented in this clause. These shorthands are merely shorthand for the operation specification schemas as described in clause A.3, and do not represent any new concepts. However, using these shorthands, the object type specifications become more succinct.

A.6.1 State transition operations of finite state machines

Objects implementing finite state machines may also have operations to perform state transitions. Although it is possible to describe such operations with operation schemas and a semantic description in English, the following notation is also possible:

Type Name

SuperType

$transitionOp == \sigma(TargetStateName_1) \oplus$
$\sigma(TargetStateName_2, StateName_1 \mid \ldots \mid StateName_n) \oplus$
$\sigma(TargetStateName_3 \mid \ldots \mid TargetStateName_m, StateName_3 \mid \ldots \mid StateName_n)$:

remarks on the transition operations in natural language (if necessary)

Type Name

This defines *transitionOp* to be an operation which may perform one of various possible state transitions (the possibilities are separated by the character '⊕'). The alternatives are taken from left to right; if a state transition is possible (i.e., is allowed by the state transition table), it is performed and the operation terminates. Each specification of a state transition operation may be of the form:

— $\sigma(TargetStateName_1)$: means a state transition of the object to state *TargetStateName;*

— $\sigma(TargetStateName_2, StateName_1 \mid \ldots \mid StateName_n)$: state transition of the object to state $TargetStateName_2$, provided the current state is one of the range $StateName_1, \ldots, StateName_n$.

— $\sigma(TargetStateName_3 \mid \ldots \mid TargetStateName_m, StateName_1 \mid \ldots \mid StateName_n)$: state transition to one of the states in the range $TargetStateName_3, \ldots, TargetStateName_m$. The choice among the various possibilities for the target state depends on the general behaviour of the object; the semantic details for this transition are described by the additional (English) text in the scheme.

If the operation is invoked but none of the state transitions are permitted, the operation raises the exception *WrongState*, defined in 9.3 (page 29) of ISO/IEC 14478-2.

A.6.2 Sequential composition of operations

The schema

TypeName

$SuperType_1$
$SuperType_2$

$operation == operation_1 \, ; \, operation_2$

remarks on the operation in natural language (if necessary).

TypeName

defines *operation* as a *sequential composition* of $operation_1$ and $operation_2$. Both $operation_1$ and $operation_2$ may be either operations defined elsewhere in the type specification schema of *TypeName* or may be, for example, of the form $SuperType_1.op$, i.e., referring to an inherited operation.

Sequential composition means that the operations are performed sequentially in left to right order, unless one of the operations is interrupted by raising an exception. In the latter case the whole operation is interrupted and raises the same exception. Only operations without arguments are specified this way in PREMO.

Although the definition above describes the sequential composition of two operations it is possible, by natural extension and, if necessary, through the usage of parentheses as delimiters, to extend the composition to an arbitrary number of operations. Also, the formalism can be used with one operation without real sequential composition, meaning simply an identification of operations.

A.7 Specification of components and profiles

Components and profiles are defined through *component schemas*. Each component shall contain one and only one component schema:

Name

Basic

provides service

type1, type2, type3
Profile Q1

provides type

type1, type2, type3
Profile Q2

requires service

Component R1 Profile P1

requires type

Component R2 Profile P2

Advanced

provides service

type1, type2, type3
Profile Q3

provides type

type1, type2, type3
Profile Q4

requires service

Component R3 Profile P3

requires type

Component R4 Profile P4
Profile Basic

Name

The top and bottom enclosing lines of the component schema contain a name for the component; this name shall be unique among all standard components of PREMO and is used as a reference to the component in other component schemas.

Each component schema consists of one or more *profile schemas*. Each profile shall have a name which is unique among all profiles of the same component; this name appears on the top line of the corresponding profile schema. The name ***Basic*** shall refer to the basic profile of the component (see clause 9).

Each profile schema consists of four sub–schemas:

c) Sub–schema ***provides service****:* Description of the types offering services through their operations to other components and profiles, or to PREMO applications. Other component profiles may have a service dependency on this profile when using the types listed in this sub–schema.

d) Sub–schema ***provides type****:* Description of the types usable for subtyping by other components and profiles, or by PREMO applications and extensions. Other component profiles may have a type dependency on this profile when using the types listed in this sub–schema.

e) Sub–schema ***requires service****:* Description of the service dependencies of the profile.

f) Sub–schema ***requires type****:* Description of the type dependencies of the profile.

Any of the four sub–schemas may be missing if, for example, the profile has no service dependencies, or does not offer any type for subtyping.

The types made available by a profile (either as services or as supertypes) are specified by:

— Listing the object types (which can include generic object types, too). Listing an object type means implicitly providing information on all non–object types appearing as part of the signature of any of the operations (either directly or indirectly, i.e., through structures or discriminated unions).

— Using the construct ***Profile Q***, where ***Q*** refers to another profile of the *same* component.

The dependencies of a profile (either service or type dependencies) are specified by:

— Using the construct ***Component R Profile Q***, which refers to the profile named ***Q*** of the component named ***R***.

— Using the construct ***Profile S***, which refers to the profile named ***S*** of the *same* component.

Language bindings to PREMO shall provide details on how these notions are mapped onto a particular programming environment. Note that in some cases the programming environment may not make it possible to enforce the difference between service and type dependencies.

Annex B
(normative)
Generic types

A *Generic Type* is a notational mechanism which gives a succinct way of defining families of PREMO object types. The notation is based on the conventions used for object type definitions, as described in clause A.3. It has to be emphasized that the notion of generic types does *not* modify the PREMO Object Model, nor does it add new notions to it; its usage is purely a convenience.

A generic type is defined by a *generic type schema*, which is analogous to a type schema:

GenericType [*A::B1*|*B2*|...|*Bn*,*C*]

operation

parameters
exceptions: {*ExceptionValue1*, *ExceptionValue2*}

behavioural description in natural language

Exceptions raised:
ExceptionValue: Description.

GenericType

The only difference between the generic type schema and the type schema is the appearance of a comma separated list of *formal types* in the type name. Formal types can refer to PREMO object types and non–object types. They can appear in the type name following two forms:

— Form *A::B1*|*B2*|…|*Bn*, where *B1, B2, …, Bn* shall be valid PREMO object type names: if only one PREMO object type is used, the format is *A::B*. The symbol *A* represents a PREMO object type, and may appear in the signature of the operations in the form *RefA*, as well as in the behavioural description of the operations.

— Form *C*: the symbol *C* represents a PREMO non–object type, and may appear in the signature of the operations as well as in the behavioural description of the operations.

B1, B2, …, Bn are referred to as the *type constraints* on the formal type *A*. If only one type is used as a constraint, the symbol '|' may be omitted.

A generic type does *not* define a new PREMO object type; instead, it defines a family of possible object types. Object types may be defined turning the formal types of a generic type schema into *actual types*; this step is referred to as the *actualization* of the generic type. This is done as follows (using the example above):

NewType

GenericType[*NA*,*NC*]

NewType

The rules for actual types are:

— If the type to be actualized in the generic type is of the form *A::B1|B2|…|Bn*, then *NA* shall be a valid PREMO object type name; furthermore, *NA* shall be a subtype of one of *B1, B2, …, Bn*. *NA* may refer to an abstract type.

— If the type to be actualized in the generic type is of the form *C*, then *NC* shall be a valid PREMO non–object type name.

Functionally, actualizing the formal types means:

— If the type to be actualized in the generic type is of the form *A::B1|B2|…|Bn*, all occurrences of *A* are replaced by *NA* and all occurrences of *RefA* are replaced by *RefNA*.

— If the type to be actualized in the generic type is of the form *C*, all occurrences of *C* are replaced by *NC*.

As a result of the actualization of the generic type schema, through the actualization of the formal types, the schema above implicitly creates a copy of the generic type schema, including all operation signatures and behavioural descriptions, but with the formal types replaced, thereby defining a valid PREMO object type.

The formalism used for type inheritance can be used for generic types, too; e.g., a new generic type could be defined by:

AnotherGenericType[*V1*]

GenericType[*V1,Char*]

operation

parameters
exceptions: {$ExceptionValue_1$, $ExceptionValue_2$}

behavioural description in natural language

Exceptions raised:

$ExceptionValue_1$	Description
$ExceptionValue_2$	Description

AnotherGenericType

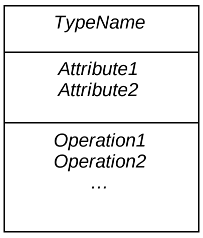
For generic types, inheritance means simply an implicit copy of all operation signatures, attributes, and formal types. By default, the behavioural descriptions are also copied, unless the keyword ‘redef’ is used, just as for type schemas. Note that in this last schema the generic type GenericType has been partially actualized (on its second formal type). Using the inheritance of generic type schemas, complex generic types can easily be defined.

NOTE — The notion similar to generic type is also known as “template” in some programming languages. Other languages may have powerful macro facilities which may be used to implement generic types.

Annex C
(informative)
Graphical conventions

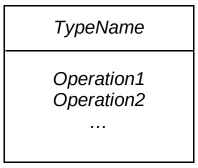
This annex describes the conventions used for the figures in various parts of ISO/IEC 14478.

Objects are depicted using the following diagram:

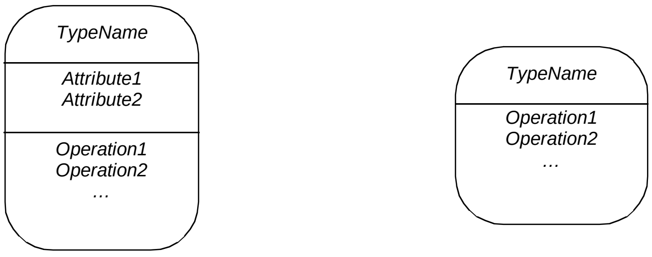


There is no pictorial distinction between object types and object instances in the figures; the difference shall be clear from the context of the figure.

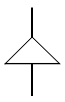
The list of attribute and operation names is not exhaustive for the objects; only those attributes and operations appear in the figures which are relevant for the context. The attribute part of the figure may be missing altogether, in which case a simplified version may be used:



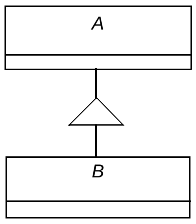
A rounded rectangle is used to depict abstract types:



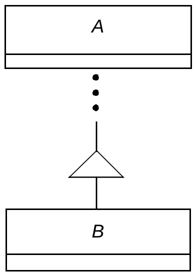
The relationship of (immediate) supertyping is depicted using the symbol



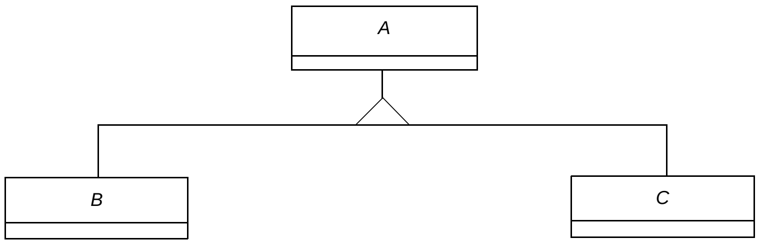
where the triangle is pointing toward the immediate supertype. For example, the figure below denotes the fact that the type *A* is and immediate supertype of *B:*



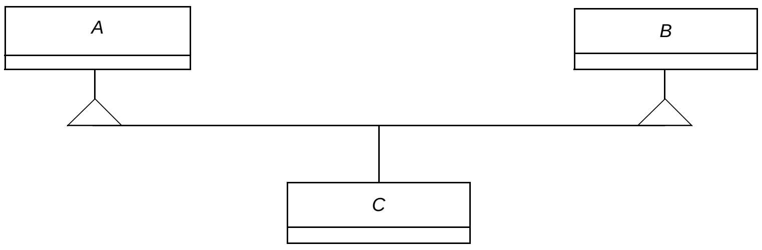
If subtyping, but not necessarily immediate subtyping, is intended, the following figure can be used:



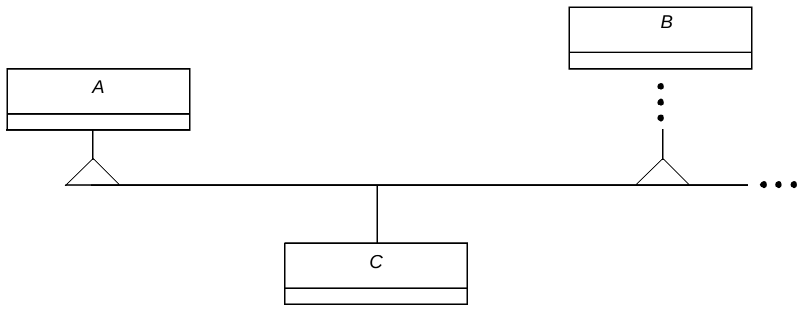
which denotes only the fact that *A* is a supertype of *B*, but it is not necessarily an immediate supertype. The lines and the triangles can also be “shared” in the figures to depict more complex subtyping relationships. For example, the following figure depicts the situation when both types *B* and *C* are immediate subtypes of *A*,



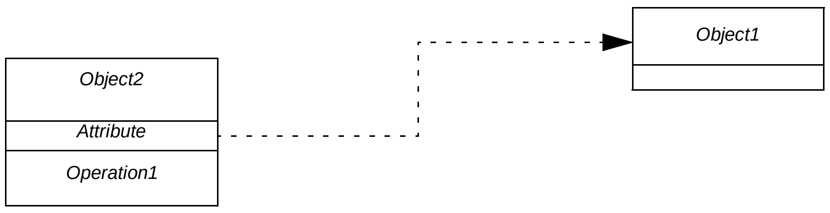
whereas the figure below describes the fact that *C* has two immediate supertypes, namely *A* and *B*.



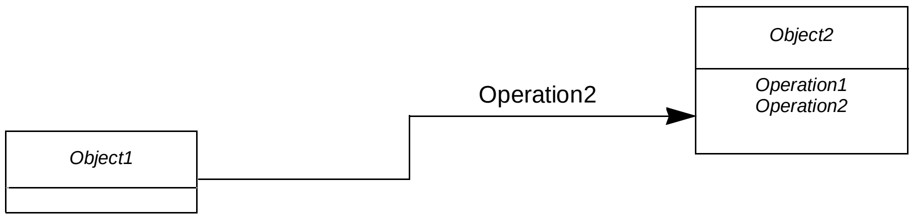
Similar notation as for the non–immediate subtyping can be used when only part of the subtyping relationships are depicted. For example, the figure below describes that fact that *A* is an immediate supertype of *C*, *B* is a supertype of *C* (but not necessarily immediate), and, finally, that *C* may have other supertypes, which do not appear on this figure.



Attributes in objects may be object references, referring to a particular object type (instance). This relationship can be expressed using the following convention (where *Attribute* is an object reference):



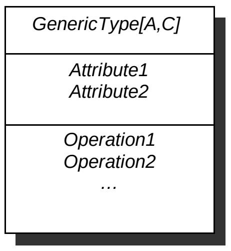
Finally, in some cases, it is also necessary to depict operation invocation in the figures. This is done as follows:



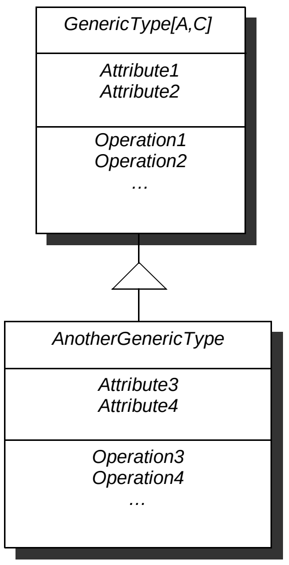
where the label on the arrow denotes the operation to be invoked.

C.1 Graphical conventions for generic types

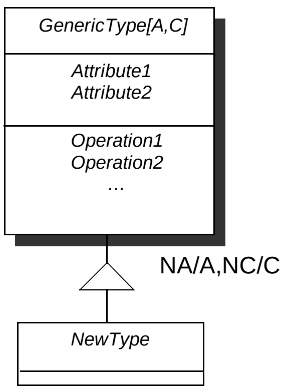
The graphical conventions used for generic types is a simple extension of the graphical conventions used for types. Generic types are denoted by:



i.e., the only difference between generic types and object types is that the enclosing rectangles have a shadow, and the formal names appear in the type names. Inheritance of generic types follows the same conventions, e.g.,



Denoting actualization is done through connecting a generic type with a PREMO type. If necessary, the data types used for the actualization may be used to label the inheritance figure, where the labels also denote which symbol is replaced by an actual type name:


GenericType[A,C]
Attribute1
Attribute2
Operation1
Operation2
…
NA/A,NC/C
NewType