INTERNATIONAL TELECOMMUNICATION UNION

ITU-T **X.903**

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(11/95)

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

OPEN DISTRIBUTED PROCESSING

INFORMATION TECHNOLOGY – OPEN DISTRIBUTED PROCESSING – REFERENCE MODEL: ARCHITECTURE

ITU-T Recommendation X.903

(Previously "CCITT Recommendation")

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC. The text of ITU-T Recommendation X.903 was approved on 21st of November 1995. The identical text is also published as ISO/IEC International Standard 10746-3.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized private operating agency.

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

(February 1994)

ORGANIZATION OF X-SERIES RECOMMENDATIONS

Subject area	Recommendation Series
PUBLIC DATA NETWORKS	
Services and Facilities	X.1-X.19
Interfaces	X.20-X.49
Transmission, Signalling and Switching	X.50-X.89
Network Aspects	X.90-X.149
Maintenance	X.150-X.179
Administrative Arrangements	X.180-X.199
OPEN SYSTEMS INTERCONNECTION	
Model and Notation	X.200-X.209
Service Definitions	X.210-X.219
Connection-mode Protocol Specifications	X.220-X.229
Connectionless-mode Protocol Specifications	X.230-X.239
PICS Proformas	X.240-X.259
Protocol Identification	X.260-X.269
Security Protocols	X.270-X.279
Layer Managed Objects	X.280-X.289
Conformance Testing	X.290-X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300-X.349
Mobile Data Transmission Systems	X.350-X.369
Management	X.370-X.399
MESSAGE HANDLING SYSTEMS	X.400-X.499
DIRECTORY	X.500-X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600-X.649
Naming, Addressing and Registration	X.650-X.679
Abstract Syntax Notation One (ASN.1)	X.680-X.699
OSI MANAGEMENT	X.700-X.799
SECURITY	X.800-X.849
OSI APPLICATIONS	
Commitment, Concurrency and Recovery	X.850-X.859
Transaction Processing	X.860-X.879
Remote Operations	X.880-X.899
OPEN DISTRIBUTED PROCESSING	X.900-X.999

CONTENTS

Summary

This Recommendation | International Standard contains the specification of the required characteristics that qualify distributed processing systems as open. These are the constraints to which ODP standards must comply. It uses the descriptive techniques from Recommendation X.902.

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for the standardization of Open Distributed Processing (ODP). This Reference Model provides such a framework. It creates an architecture within which support of distribution, interworking and portability can be integrated.

The Reference Model of Open Distributed Processing, ITU-T Recs. X.901 to X.904 | ISO/IEC 10746, is based on precise concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques for specification of the architecture.

The Reference Model consists of:

– ITU-T Rec. X.901 | ISO/IEC 10746-1: **Overview**: Contains a motivational overview of ODP giving scoping, justification and explanation of key concepts, and an outline of the ODP architecture. It contains explanatory material on how this Reference Model is to be interpreted and applied by its users, who may include standards writers and architects of ODP systems. It also contains a categorization of required areas of standardization expressed in terms of the reference points for conformance identified in this Recommendation | International Standard. This part is not normative.

– ITU-T Rec. X.902 | ISO/IEC 10746-2: **Foundations**: Contains the definition of the concepts and analytical framework for normalised description of (arbitrary) distributed processing systems. It introduces the principles of conformance to ODP standards and the way in which they are applied. This is only to a level of detail sufficient to support this Recommendation | International Standard and to establish requirements for new specification techniques. This part is normative.

– ITU-T Rec. X.903 | ISO/IEC 10746-3: **Architecture**: Contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards must conform. It uses the descriptive techniques from ITU-T Recommendation X.902 | ISO/IEC 10746-2. This part is normative.

– ITU-T Rec. X.904 | ISO/IEC 10746-4: **Architectural semantics**: Contains a formalization of the ODP modelling concepts defined in ITU-T Recommendation X.902 | ISO/IEC 10746-2, clauses 8 and 9. The formalization is achieved by interpreting each concept in terms of the constructs of the different standardized formal description techniques. This part is normative.

This Recommendation | International Standard contains one annex (this annex forms an integral part of the Reference Model).

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

INFORMATION TECHNOLOGY – OPEN DISTRIBUTED PROCESSING – REFERENCE MODEL: ARCHITECTURE

1 Scope

This ITU-T Recommendation | International Standard:

- defines how ODP systems are specified, making use of concepts in ITU-T Recommendation X.902 | ISO/IEC 10746-2;
- identifies the characteristics that qualify systems as ODP systems.

It establishes a framework for coordinating the development of existing and future standards for ODP systems and is provided for reference by those standards.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunications Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.200 (1994) | ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection – Basic Reference Model*: *The Basic Model.*
- ITU-T Recommendation X.810 (1995) | ISO/IEC 10181-1:1996, *Information technology – Open Systems Interconnection – Security frameworks for open systems: Overview.*
- ITU-T Recommendation X.811 (1995) | ISO/IEC 10181-2:1996, *Information technology – Open Systems Interconnection – Security frameworks for open systems: Authentication framework*.
- ITU-T Recommendation X.812 (1995) | ISO/IEC 10181-3:1996, *Information technology – Open Systems Interconnection – Security frameworks for open systems: Access control framework*.
- ITU-T Recommendation X.813[1)] | ISO/IEC 10181-4...[1)], *Information technology – Open Systems Interconnection – Security frameworks for open systems: Non-repudiation framework.*
- ITU-T Recommendation X.814 (1995) | ISO/IEC 10181-5:1996, *Information technology – Open Systems Interconnection – Security frameworks for open systems: Confidentiality framework*.
- ITU-T Recommendation X.815 (1995) | ISO/IEC 10181-6:1996, *Information technology – Open Systems Interconnection – Security frameworks for open systems: Integrity framework*.
- ITU-T Recommendation X.816 (1995) | ISO/IEC 10181-7:1996, *Information technology – Open Systems Interconnection – Security frameworks for open systems: Security audit framework.*
- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, *Information technology – Open distributed processing – Reference Model: Foundations*.

[1)] Presently at the stage of draft.

2.2 Paired Recommendations | International Standards equivalent in technical content

– CCITT Recommendation X.800 (1991), *Security architecture for Open Systems Interconnection for CCITT Applications.*

  ISO 7498-2:1989, *Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture.*

3 Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply.

3.1 Descriptive definitions

This Reference Model makes use of the following term defined in ITU-T Rec. X.200 | ISO/IEC 7498-1:

– transfer syntax.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.811 | ISO/IEC 10181-2:

– claimant;
– exchange authentication information;
– principal;
– trusted third party.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.812 | ISO/IEC 10181-3:

– access control information;
– access decision function;
– access enforcement function;
– initiator;
– target.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec X.813 | ISO/IEC 10181-4:

– evidence generator;
– evidence user;
– evidence verifier;
– (non-repudiable data) originator;
– (non-repudiable data) recipient;
– non-repudiation evidence;
– non-repudiation service requester;
– notary.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.814 | ISO/IEC 10181-5:

– confidentiality-protected data;
– hide;
– originator;
– recipient;
– reveal.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.815 | ISO/IEC 10181-6:

- integrity-protected data;
- originator;
- recipient;
- shield;
- validate.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.816 | ISO/IEC 10181-7:

- alarms collector function;
- alarm examiner function;
- audit trail examiner function;
- audit trail archiver function;
- audit recorder function;
- audit trail examiner function;
- audit trail collector function.

This Recommendation | International Standard makes use of the following terms defined in ISO/IEC 11170-1 Key Management Framework:

- key generation;
- key registration;
- key certification;
- key deregistration;
- key distribution;
- key storage;
- key archiving;
- key deletion.

This Reference Model makes use of the terms defined in ITU-T Rec. X.902 | ISO/IEC 10746-2 shown in Figure 1.

3.2 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ODP	Open Distributed Processing.
OSI	Open Systems Interconnection.

4 Framework

This Reference Model defines a framework comprising:

- five *viewpoints*, called enterprise, information, computational, engineering and technology which provide a basis for the specification of ODP systems;
- a *viewpoint language* for each viewpoint, defining concepts and rules for specifying ODP systems from the corresponding viewpoint;
- specifications of the *functions* required to support ODP systems;
- *transparency prescriptions* showing how to use the ODP functions to achieve distribution transparency.

The architecture for ODP systems and the composition of functions is determined by the combination of the computational language, the engineering language and the transparency prescriptions.

abstraction;	failure;	perceptual reference point;
action;	fault;	permission;
activity;	<x> group;	persistence;
architecture;	identifier;	policy;
atomicity;	information;	producer object;
behaviour;	initiating object;	programmatic reference point;
binding;	instance;	prohibition;
class;	instantiation;	Quality of Service;
client object;	interaction;	reference point;
communication;	interchange reference point;	refinement;
communications management;	interface;	role;
composition;	interface signature;	server object;
configuration;	interworking reference point;	spawn action;
conformance point;	introduction;	stability;
consumer object;	invariant;	state;
contract;	liaison;	subdomain;
creation;	location in space;	subtype;
data;	location in time;	supertype;
decomposition;	name;	system;
deletion;	naming context;	<x> template;
distributed processing;	naming domain;	term;
distribution transparency;	notification;	thread;
<x> domain;	object;	trading;
entity;	obligation;	type;
environment;	ODP standards;	viewpoint.
error;	ODP system;	
establishing behaviour;	open distributed processing;	

Figure 1 – Terms taken from ITU-T Rec. X.902 | ISO/IEC 10746-2

4.1 Viewpoints

4.1.1 Concepts

4.1.1.1 Enterprise viewpoint: A viewpoint on an ODP system and its environment that focuses on the purpose, scope and policies for that system.

4.1.1.2 Information viewpoint: A viewpoint on an ODP system and its environment that focuses on the semantics of information and information processing.

4.1.1.3 Computational viewpoint: A viewpoint on an ODP system and its environment which enables distribution through functional decomposition of the system into objects which interact at interfaces.

4.1.1.4 Engineering viewpoint: A viewpoint on an ODP system and its environment that focuses on the mechanisms and functions required to support distributed interaction between objects in the system.

4.1.1.5 Technology viewpoint: A viewpoint on an ODP system and its environment that focuses on the choice of technology in that system.

4.1.2 Using viewpoints

The enterprise, information, computational, engineering and technology viewpoints have been chosen as a necessary and sufficient set to meet the needs of ODP standards. Viewpoints can be applied, at an appropriate level of abstraction, to a complete ODP system, in which case the environment defines the context in which the ODP system operates. Viewpoints can also be applied to individual components of an ODP system, in which case the component's environment will include some abstraction of both the system's environment and other system components.

NOTE – The process of abstraction might be such that the system's environment and the other system components are composed into a single object.

4.2 ODP viewpoint languages

4.2.1 Concept

4.2.1.1 <Viewpoint> language: Definitions of concepts and rules for the specification of an ODP system from the <viewpoint> viewpoint; thus: **engineering language**: definitions of concepts and rules for the specification of an ODP system from the engineering viewpoint.

4.2.2 Using viewpoint languages

This Reference Model defines a set of five languages, each corresponding to one of the viewpoints defined in 4.1.1. Each language is used for the specification of an ODP system from the corresponding viewpoint. These languages are:

– the enterprise language (defined in clause 5);

– the information language (defined in clause 6);

– the computational language (defined in clause 7);

– the engineering language (defined in clause 8);

– the technology language (defined in clause 9).

Each language uses concepts taken from ITU-T Rec. X.902 | ISO/IEC 10746-2, and introduces refinements of those concepts, prescriptive rules and additional viewpoint-specific concepts relevant to the nature of the specifications concerned. These additional concepts are, in turn, defined using concepts from ITU-T Rec. X.902 | ISO/IEC 10746-2.

A system specification comprises one or more viewpoint specifications. These specifications must be mutually consistent. Rules for the consistent structuring of viewpoint specifications are given in clause 10. The specifier must demonstrate by other means that terms in the specifications are used consistently. A specification of a system using several viewpoint specifications will often restrict implementations more than a specification using fewer viewpoint specifications. Objects identified in one viewpoint can be specified using the viewpoint language associated with that viewpoint or using the viewpoint languages associated with other viewpoints. It is not necessary to specify an object fully from every viewpoint in order to achieve a mutually consistent set of viewpoint specifications.

NOTES

1 The list of terms taken from ITU-T Rec X.902 | ISO/IEC 10746-2 are listed in Figure 1.

2 The qualification of a term from ITU-T Rec X.902 | ISO/IEC 10746-2 by the name of a viewpoint (e.g. as in "**computational** object") is interpreted as using of the term from ITU-T Rec X.902 | ISO/IEC 10746-2, subject to whatever additional provisions are specified in the identified viewpoint language.

3 The unqualified use of a term from ITU-T Rec X.902 | ISO/IEC 10746-2 in a viewpoint specification (e.g. "interface") is interpreted as if the term had been qualified by the name of the viewpoint (i.e. "computational interface"), if the associated viewpoint language places additional constraints on the term.

4.3 ODP functions

4.3.1 ODP function: A function required to support Open Distributed Processing.

4.3.2 Using ODP functions

This Reference Model specifies, in clauses 11 to 15, the functions required to achieve Open Distributed Processing.

Each ODP function description contains:

– an explanation of the use of the function for open distributed processing;

– prescriptive statements, about the structure and behaviour of the function, sufficient to ensure the overall integrity of the Reference Model;

– a statement of other ODP functions upon which it depends.

4.4 ODP distribution transparencies

4.4.1 Concepts

4.4.1.1 Access transparency: A distribution transparency which masks differences in data representation and invocation mechanisms to enable interworking between objects.

4.4.1.2 Failure transparency: A distribution transparency which masks, from an object, the failure and possible recovery of other objects (or itself), to enable fault tolerance.

4.4.1.3 Location transparency: A distribution transparency which masks the use of information about location in space when identifying and binding to interfaces.

4.4.1.4 Migration transparency: A distribution transparency which masks, from an object, the ability of a system to change the location of that object. Migration is often used to achieve load balancing and reduce latency.

4.4.1.5 Relocation transparency: A distribution transparency which masks relocation of an interface from other interfaces bound to it.

4.4.1.6 Replication transparency: A distribution transparency which masks the use of a group of mutually behaviourally compatible objects to support an interface. Replication is often used to enhance performance and availability.

4.4.1.7 Persistence transparency: A distribution transparency which masks, from an object, the deactivation and reactivation of other objects (or itself). Deactivation and reactivation are often used to maintain the persistence of an object when a system is unable to provide it with processing, storage and communication functions continuously.

4.4.1.8 Transaction transparency: A distribution transparency which masks coordination of activities amongst a configuration of objects to achieve consistency.

4.4.2 Using distribution transparency

Distribution transparency is an important end-user requirement in distributed systems. This Reference Model defines a set of distribution transparencies which make it possible to implement ODP systems which are distribution transparent from the point of view of users of those systems. Distribution transparency is selective; the Reference Model includes rules for selecting and combining distribution transparencies in ODP systems.

This Reference Model contains, for each distribution transparency defined in 4.4.1.1 to 4.4.1.8, definitions of both:

- a schema for expressing requirements for the particular transparency;
- a refinement process for transforming a specification which contains requirements for the particular distribution transparency to a specification which explicitly realizes the masking implied by that transparency.

NOTES

1 In some cases (e.g. access transparency) the schema is null; in others (e.g. transaction transparency) the schema contains one or more parameters dictating the precise form of transparency required.

2 The refinement process typically involves introducing additional behaviour, including the use of one or more ODP functions into the specification.

The specifications of the refinement processes in clause 16 are prescriptive to the level required to ensure overall integrity of the Reference Model.

4.5 Standards derived from the framework

This Reference Model provides a framework for the definition of new standards and the use of existing standards as ODP standards.

ODP standards are any of:

- standards for components of ODP systems;
- standards for composing ODP system components;
- standards for modelling and specifying ODP systems.

ODP standards:

- use the enterprise language to specify policies;
- use the information language to specify consistent use and interpretation of information in, and between, standards;
- use the computational language to specify the configuration and behaviour of interfaces;
- use the engineering language to specify the infrastructures they require;
- use the technology language to specify conformance to international, private, or consensual specifications.

Standards for methodology, modelling, programming, implementation and testing of ODP systems use the framework as a whole.

ODP standards can be based on a subset of this Reference Model (e.g. by excluding some forms of interaction, particular functions or transparencies). Such standards can also extend upon this Reference Model, provided that the extensions they introduce do not change or contradict its provisions. Extensions will relate new terms to terms defined in this Reference Model: for example, by introducing new types and new type rules.

ODP standards comply with all the prescriptive statements in this Reference Model.

4.6 Conformance

The enterprise, information, computational and engineering languages are used to specify the conformance requirements for ODP systems. The technology language can be used to assert conformance to ODP standards in ODP systems. Each interface which is defined as a conformance point has an information specification to enable interpretation of interactions of that interface. The rules for identifying conformance points are given in the computational and engineering languages.

An ODP system conforms to an ODP standard if it satisfies the conformance requirements of that standard.

5 Enterprise language

The enterprise language comprises concepts, rules and structures for the specification of an ODP system from the enterprise viewpoint.

An enterprise specification defines the purpose, scope and policies of an ODP system.

In this Reference Model, prescription in the enterprise viewpoint is restricted to a small basic set of concepts and rules addressing the scope and nature of enterprise specifications.

5.1 Concepts

The enterprise language contains the concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and those defined here, subject to the rules of 5.2.

5.1.1 **Community**: A configuration of objects formed to meet an objective. The objective is expressed as a contract which specifies how the objective can be met.

5.1.2 **<X> federation**: A community of <x> domains.

5.2 Structuring rules

An enterprise specification defines, and the enterprise language is able to express, the purpose, scope and policies of an ODP system in terms of each of the following items:

- roles played by the system;
- activities undertaken by the system;
- policy statements about the system, including those relating to environment contracts.

In an enterprise specification, an ODP system and the environment in which it operates are represented as a community. At some level of description the ODP system is represented as an enterprise object in the community. The objectives and scope of the ODP system are defined in terms of the roles it fulfils within the community of which it is part, and policy statements about those roles. A community is defined in terms of each of the following elements:

- the enterprise objects comprising the community;
- the roles fulfilled by each of those objects;
- policies governing interactions between enterprise objects fulfilling roles;
- policies governing the creation, usage and deletion of resources by enterprise objects fulfilling roles;
- policies governing the configuration of enterprise objects and assignment of roles to enterprise objects;
- policies relating to environment contracts governing the system.

A role is defined in terms of the permissions, obligations, prohibitions and behaviour of the enterprise object fulfilling the role. An enterprise object can fulfil one or more roles in a community, and the roles which it can fulfil are determined by the contract on which the community is based. While it is part of one community the enterprise object can continue to fulfil roles in other communities, subject to the provisions in the contracts of the communities involved. The enterprise object can fulfil different roles in different communities. Interactions between enterprise objects fulfilling appropriate roles within different communities can be considered as interactions between those communities.

NOTE 1 – Examples of roles include policy administrator, president, service provider, owner, manager, shareholder, consumer.

NOTE 2 – Examples of environment contracts in enterprise specifications include safety requirements, legislative requirements and codes of practice.

NOTE 3 – In an enterprise specification the term "<x> object", where <x> is a role, is interpreted as meaning "an enterprise object fulfilling an <x> role"; where an enterprise object fulfils multiple roles, the names can be concatenated, e.g. "owner driver object".

When fulfilling a role, an object becomes subject to permissions, obligations and prohibitions by delegation or transfer. In some roles, objects are permitted to change policy. There are five fundamental types of actions with respect to contractual matters:

– an object incurs an obligation to another object (it must currently be permitted to incur the obligation);
– an object fulfils an obligation to another object;
– an object waives an obligation to another object;
– an object acquires permission from another object to perform some action it was previously forbidden to perform;
– an object is forbidden to perform an action it was previously permitted to perform.

NOTE 4 – An important special case of acquisition is where the permitted action is performative, i.e. when an object in a subordinate role is enabled to issue further permissions or obligations on behalf of an object fulfilling a superior role. This leads to the notion of agency or delegation.

Obligations include accounting and charging for the use of resources. Billing and payment is modelled as the reassignment of resources between objects in accordance with the roles they fulfil.

A resource is either consumable or non-consumable. A consumable resource deletes itself after some amount of use. In <x> federation, the objective defines the resources each <x> domain in the federation shares with the other members of the federation. The objective can leave each domain with a defined degree of autonomy in the use of its own resources. The establishing behaviour for an <x> federation can allow autonomy for each participating <x> domain in deciding whether or not to become part of the federation.

5.3 Conformance and reference points

Conformance statements in the enterprise language require that the behaviour of an ODP system is conformant to a particular set of objectives and policies.

An implementor claiming conformance must identify the engineering reference points which give access to the system and the engineering, computational and information specifications which apply at them. By this act the identified reference points become conformance points. The interactions at these conformance points can then be interpreted in enterprise language terms to check that the enterprise specification is not violated.

Enterprise specifications can be applied to all four classes of reference point (programmatic, perceptual, interworking and interchange reference points) identified in ITU-T Rec. X.902 | ISO/IEC 10746-2.

6 Information language

The information language comprises concepts, rules and structures for the specification of an ODP system from the information viewpoint.

An information specification defines the semantics of information and the semantics of information processing in an ODP system.

In this Reference Model, prescription in the information viewpoint is restricted to a small basic set of concepts and rules addressing the scope and nature of information specifications.

6.1 Concepts

The information language contains the concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and those defined here, subject to the rules of 6.2.

6.1.1 Invariant schema: A set of predicates on one or more information objects which must always be true. The predicates constrain the possible states and state changes of the objects to which they apply.

NOTE – Thus, an invariant schema is the specification of the types of one or more information objects that will always be satisfied by whatever behaviour the objects might exhibit.

6.1.2 Static schema: A specification of the state of one or more information objects, at some point in time, subject to the constraints of any invariant schemata.

NOTE – Thus, a static schema is the specification of the types of one or more information objects at some particular point in time. These types are subtypes of the types specified in the invariant schema.

6.1.3 Dynamic schema: A specification of the allowable state changes of one or more information objects, subject to the constraints of any invariant schemata.

NOTES

1 Behaviour in an information system can be modelled as transitions from one static schema to another, i.e. reclassification of instances from one type to another.

2 In the information language, a state change involving a set of objects can be regarded as an interaction between those objects. Not all of the objects involved in the interaction need change state; some of the objects may be involved in a read-only manner.

6.2 Structuring rules

An information specification defines the semantics of information and the semantics of information processing in an ODP system in terms of a configuration of information objects, the behaviour of those objects and environment contracts for the system.

An information object template references static, invariant and dynamic schemata. The relationships between information objects can be modelled as part of the state of those information objects. Information objects are either atomic or are represented as a composition of component information objects. The state of the composite object is represented by the combined state of its component information objects. An atomic information object template represents a concept for which there is no model at a particular level of abstraction. A composite information object represents a derived concept expressed in terms of other concepts. Since object composition includes encapsulation, an information object that is a component of one composite object cannot be a component of another. Therefore, information objects resulting from the instantiation of a composite information object template only exist as part of the instantiated composite object and have no meaning outside it.

Allowable state changes specified by a dynamic schema can include the creation of new information objects and the deletion of information objects involved in the dynamic schema. Allowable state changes can be subject to ordering and temporal constraints.

NOTE 1 – The result of accessing the state of one or more information objects can be modelled as the creation of a new information object.

In an information specification, the configuration of information objects and the behaviour of those objects need not be suitable for distribution (e.g. there need not be any concept of failure or location for information interactions).

NOTE 2 – If an information notation uses the concept of interface, any interfaces defined cannot themselves be reference points; thus there is no commitment to the interfaces appearing in an implementation.

6.3 Conformance and reference points

Conformance statements in information specifications require that the behaviour of an ODP system is conformant to a particular set of invariant, static and dynamic schemata.

An implementor claiming conformance must list the relevant engineering reference points which give access to the system and the engineering and computational specifications which apply to them. By this act the identified reference points become conformance points. The interactions at these conformance points can then be interpreted in information language terms to check that they are consistent with the invariant, static and dynamic schemata.

Information specifications can be applied to all four classes of reference point (programmatic, perceptual, interworking, and interchange reference points) identified in ITU-T Rec. X.902 | ISO/IEC 10746-2.

7 Computational language

The computational language comprises concepts, rules and structures for the specification of an ODP system from the computational viewpoint.

A computational specification defines the functional decomposition of an ODP system into objects which interact at interfaces.

In the computational viewpoint, applications and ODP functions consist of configurations of interacting computational objects.

7.1 Concepts

The computation language contains the concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and those defined here, subject to the structuring rules of 7.2.

7.1.1 **Signal**: An atomic shared action resulting in one-way communication from an initiating object to a responding object.

NOTE – A signal is an interaction.

7.1.2 **Operation**: An interaction between a client object and a server object which is either an interrogation or an announcement.

7.1.3 **Announcement**: An interaction – the **invocation** – initiated by a client object resulting in the conveyance of information from that client object to a server object, requesting a function to be performed by that server object.

7.1.4 **Interrogation**: An interaction consisting of:

– one interaction – the **invocation** – initiated by a client object, resulting in the conveyance of information from that client object to a server object, requesting a function to be performed by the server object, followed by

– a second interaction – the **termination** – initiated by the server object, resulting in the conveyance of information from the server object to the client object in response to the invocation.

NOTE – In interrogations, invocations and terminations are always paired. Announcements do not have terminations. Thus there is no possibility of an operation consisting of an invocation followed by a sequence of associated terminations.

7.1.5 **Flow**: An abstraction of a sequence of interactions, resulting in conveyance of information from a producer object to a consumer object.

NOTE – A flow may be used to abstract over, for example, the exact structure of a sequence of interactions, or over a continuous interaction including the special case of an analogue information flow.

7.1.6 **Signal interface**: An interface in which all the interactions are signals.

7.1.7 **Operation interface**: An interface in which all the interactions are operations.

7.1.8 **Stream interface**: An interface in which all the interactions are flows.

7.1.9 **Computational object template**: An object template which comprises a set of computational interface templates which the object can instantiate, a behaviour specification and an environment contract specification.

7.1.10 **Computational interface template**: An interface template for either a signal interface, a stream interface or an operation interface. A computational interface template comprises a signal, a stream or an operation interface signature as appropriate, a behaviour specification and an environment contract specification.

7.1.11 **Signal interface signature**: An interface signature for a signal interface. A signal interface signature comprises a finite set of action templates, one for each signal type in the interface. Each action template comprises the name for the signal, the number, names and types of its parameters and an indication of causality (initiating or responding, but not both) with respect to the object which instantiates the template.

7.1.12 **Operation interface signature**: An interface signature for an operation interface. An operation interface signature comprises a set of announcement and interrogation signatures as appropriate, one for each operation type in the interface, together with an indication of causality (client or server, but not both) for the interface as a whole, with respect to the object which instantiates the template.

Each **announcement signature** is an action template containing both the name of the invocation and the number, names and types of its parameters.

Each **interrogation signature** comprises an action template with the following elements:

- the name of the invocation;
- the number, names and types of its parameters;
- a finite, non-empty set of action templates, one for each possible termination type of the invocation, each containing both the name of the termination and the number, names and types of its parameters.

7.1.13 Stream interface signature: An interface signature for a stream interface. A stream interface comprises a finite set of action templates, one for each flow type in the stream interface. Each action template for a flow contains the name of the flow, the information type of the flow, and an indication of causality for the flow (i.e. producer or consumer but not both) with respect to the object which instantiates the template.

NOTES

1 The phrase "complementary interface signature to *X*", where *X* is itself an interface signature describes an interface signature identical to *X* in all respects except causality, which is opposite to that in *X*.

2 Many Interface Definition Languages (IDLs) capture only the action templates of a signature and depend upon the context in which the IDL is used to determine the causality that is to be applied.

7.1.14 Binding object: A computational object which supports a binding between a set of other computational objects.

NOTE – Binding objects are subject to special provisions (see 7.2.3).

7.2 Structuring rules

A computational specification describes the functional decomposition of an ODP system, in distribution transparent terms, as:

- a configuration of computational objects (including binding objects);
- the internal actions of those objects;
- the interactions that occur among those objects;
- environment contracts for those objects and their interfaces.

A computational specification is constrained by the rules of the computational language. These comprise:

- interaction rules (see 7.2.2), binding rules (see 7.2.3) and type rules (see 7.2.4) that provide distribution transparent interworking;
- template rules (see 7.2.5) that apply to all computational objects;
- failure rules (see 7.2.6) that apply to all computational objects and identify the potential points of failure in computational activities.

Portability rules (see 7.2.7) are provided to give guidance to developers of ODP Portability standards.

A computational specification defines an initial set of computational objects and their behaviour. The configuration will change as the computational objects:

- instantiate further computational objects;
- instantiate further computational interfaces;
- perform binding actions;
- effect control functions upon binding objects;
- delete computational interfaces;
- delete computational objects.

7.2.1 Naming rules

Each kind of name defined in the computational language has an associated context, as follows:

- a signal name in a signal interface signature is an identifier in the context of that signature;
- a flow name in a stream interface signature is an identifier in the context of that signature;
- an invocation name in an operation interface signature is an identifier in the context of that signature;
- a termination name in an operation interface signature is an identifier in the context of the operation template in which it appears;
- the name of a parameter in a signal template is an identifier in the context of that template;

- the name of a parameter in an invocation template in an operation interface signature is an identifier in the context of that template;
- the name of a parameter in a termination template in an operation interface signature is an identifier in the context of that template;
- the name of a parameter in a signal template in a signal interface signature is an identifier in the context of that template.

NOTE 1 – Thus signal names are distinct in any signal interface signature but signals in different signatures can have the same name, and so forth.

A computational interface identifier is unambiguous within its context (i.e. cannot be associated with more than one computational interface in that context). The choice of contexts for computational interface identifiers is a matter of language design and therefore beyond the scope of this Reference Model. Hence the Reference Model places no constraints on the extent of contexts for computational interface identifiers. Thus no reliance can be placed on:

- the extent of naming contexts for computational interface identifiers (e.g. assumptions about them being related to engineering language structures such as nodes or communications domains);
- the uniqueness of computational interface identifiers (i.e. synonyms are permitted);
- a computational interface identifier denoting the same computational interface in all places where the identifier appears (i.e. names need not be "global").

NOTE 2 – A particular computational notation may not have explicit terms denoting computational identifiers; therefore in that notation computational interface identifiers are implicit; however they remain subject to the rules given above.

7.2.2 Interaction rules

Each interaction of a computational object occurs at one of its computational interfaces. The computational language imposes constraints on the behaviour permitted at a computational interface. Interaction at an unbound interface fails. The binding rules (see 7.2.3) impose constraints on how interfaces can be bound.

The interaction part of the computational language supports three models of interaction, each of which has an associated kind of computational interface:

- signals and signal interfaces;
- flows and stream interfaces;
- operations and operation interfaces.

In addition to the different kinds of interaction supported, the interaction models differ in their failure properties. The participants in a flow or operation can have an inconsistent view of an interaction at different times, especially when failures have occurred. In contrast to streams and operations, there is no concept of partial failure of a signal – a signal either succeeds or fails identically for both participants in the interaction.

7.2.2.1 Signal interaction rules

A computational object offering a signal interface of a given signal interface type:

- initiates signals that have initiating causality in the interface's signature;
- responds to signals that have responding causality in the interfaces's signature.

7.2.2.2 Stream interaction rules

A computational object offering a stream interface:

- generates flows that have producer causality in the interface's signature;
- receives flows that have consumer causality in the interface's signature.

7.2.2.3 Operation interaction rules

A client object using an operation interface invokes the operations named in the interface's signature. A server object offering an operation interface expects any of the operations named in the interface's signature. In the case of an interrogation, the server responds to the invocation by initiating any one of the terminations named for the operation in the server interface signature. The client expects any of the terminations named for the operation in the client interface signature. The duration of the operation is arbitrary unless required otherwise by environment contracts applicable to the objects and interfaces involved.

NOTE – If a client thread invokes a chain of interrogations, the two way "handshake" of invocation and termination ensures that operations will be responded to by the server in the same order as the client initiated them. If the client invokes a chain of announcements (or a chain containing both announcements and invocations) there is no handshake to guarantee the order in which the server responds to the announcement, unless this is implicit in the environment contracts applicable to the interaction. There are no ordering guarantees for either interrogations or announcements that are in different descendant activities of a previous dividing action.

7.2.2.4 Parameter rules

The parameters for signals, invocations and terminations can include identifiers for computational interfaces and computational interface signature types.

NOTE 1 – The possibility of computational interface signature parameters makes the computational signature type systems higher order. Explicit representation of signature types is required, for example in trading, where the parameters of import and export operations include signature types:

trader.import (T: Type, ...) : (service: T) -> failed (reason: String)
trader.export (T: Type, service: T) : (....) -> failed (reason: String)

This introduces a need for dynamic signature subtype checking (see 7.2.5.1).

A formal parameter that is an identifier for a computational interface is qualified by a computational interface signature type. The corresponding actual parameter must reference an interface with that interface signature type (or one of its subtypes). The actual parameter can only be used as if it referenced a computational interface with the same signature type as the formal parameter (or one of the formal parameter’s supertypes). After an interaction, both initiator and responder can reference the identified interface, although possibly by different computational interface identifiers.

NOTE 2 – This rule prevents the user of the interface referenced by the actual parameter being able to perform additional interactions beyond those in the formal interface signature type, even if the interface referenced by the actual parameter is a subtype of the interface signature type associated with the formal parameter.

7.2.2.5 Flows, operations and signals

Flows and operations can be defined in terms of signals. This enables signal interfaces to be used as a basis for explaining multi-party binding; end-to-end Quality of Service characteristics and compound bindings between different kinds of interface (e.g. stream to operation interface bindings).

A definition of flows using signals depends upon the details of the interactions abstracted in the specification of the stream interface concerned and therefore is beyond the scope of this Reference Model.

Operations can be modelled by signals by introducing corresponding signal interfaces to both the client operation interface and server operation interface involved:

– in a signal interface corresponding to a client operation interface there is a signal (**invocation submit**) corresponding to each invocation with the same parameters, and, in the case of an interface containing interrogations, a signal (**termination deliver**) corresponding to each possible termination with the same parameters as that termination;

– in the signal interface corresponding to a server operation interface there is a signal (**invocation deliver**) corresponding to each invocation with the same parameters, and, in the case of an interface containing interrogations, a signal (**termination submit**) corresponding to each possible termination with the same parameters as that termination.

This creates an equivalence between the resulting set of signals and the set of invocations and terminations in the operation interfaces being described.

7.2.3 Binding rules

In this Reference Model binding is defined with reference to binding actions. Use of such actions is called **explicit binding**. There are two kinds of binding actions: **primitive binding actions** and **compound binding actions**.

A primitive binding action binds two computational object directly. A compound binding action can be expressed in terms of primitive binding actions linking two or more computational objects via a binding object. The presence of a binding object in a computational binding gives the means to express configuration and Quality of Service control (see 7.2.3.3).

In notations which have no terms for expressing binding actions, binding is implicit. **Implicit binding** for other than server operation interfaces is not defined in the Reference Model because in other cases it is not self-evident where the initiative in binding should be placed relative to subsequent interaction. The additional information needed can be supplied in an explicit binding action.

7.2.3.1 Implicit binding rules for server operation interfaces

If an invocation by a client object references a server operation interface to which the client is not bound, implicit binding is required. Setting up an implicit binding involves the following procedure if a suitable client operation interface bound to the server does not exist:

– create a client operation interface of complementary signature type to the server interface;

– bind the client operation interface to the server operation interface;

- invoke the server object using the client operation interface;
- (optionally) when the operation completes, delete the client interface.

7.2.3.2 Primitive binding rules

Primitive binding actions enable binding of an interface of the object which initiates the action to another interface (of another object, or itself). The binding action is parameterized by two identifiers, one for each interface involved. The pre-conditions for a primitive binding action are that both interfaces involved are of the same kind (viz. signal, stream or operation), are of complementary causality and their signature types are complementary.

Primitive binding either establishes a binding between the two interfaces concerned, or fails.

Deleting an interface that has been bound to another interface using a primitive binding action deletes the binding as well as the interface.

7.2.3.3 Compound binding rules

Compound binding actions enable a set of interfaces to be bound, using a binding object to support the binding. Except for the provisions of this clause, a binding object is an ordinary computational object. In a binding object template, the behaviour specification is expressed in terms of a set of formal role parameters each of which is associated with an interface template.

Compound binding actions are parameterized by a binding object template and a set of interfaces to be bound for interaction.

The pre-conditions for compound binding are that, for each formal role in the binding object template:

- the corresponding interface parameter must be of the same kind (viz. signal, stream or operation) as the interface template associated with the formal role in the binding object template;
- the corresponding interface parameter must be of complementary causality to the interface template associated with the formal role in the binding object template;
- the corresponding interface parameter must be a subtype of the signature type of the interface template associated with the formal role in the binding object template.

A compound binding action comprises the following steps:

- a binding object is instantiated from the binding object template;
- each interface template within the binding object associated with a formal role parameter in the binding object template is instantiated;
- the binding object uses primitive binding actions to bind each such interface to the interface referenced in the corresponding actual parameter;
- a set of control interfaces are instantiated and identifiers for these interfaces returned as results of the binding action (i.e. becoming part of the state of the object which performed the action – this object can subsequently pass on the identifier by interaction with other computational objects).

The control interfaces of a binding object provide some or all of the following functions:

- monitoring the use of the binding;
- monitoring changes to the binding;
- authorizing changes to the binding;
- changing the membership of the binding;
- changing the pattern of communication enabled by the binding;
- changing the Quality of Service of the binding;
- deleting the binding as a whole.

The effect of deleting a binding to a binding object is determined by the behaviour of the binding object.

7.2.4 Type rules

This Reference Model specifies signature type rules for computational interfaces. Signature subtyping rules define minimum requirements for one interface to substitute for another. The rules are based on the interaction semantics of computational interfaces (viz. signal, stream and operation interfaces). They are sufficient to ensure that a substituted interface can consistently interpret the structure of any interactions that occur.

Signature subtyping rules for interfaces with alternative interaction semantics can be defined in terms of signals; such definitions can be introduced by an ODP standard.

7.2.4.1 Signature subtyping rules for signal interfaces

The definition of signal interface signature subtyping is given in Annex A. For signal interface types that are not defined recursively, the rules are summarized below.

Signal interface signature type *X* is a subtype of signal interface signature type *Y* if the conditions below are met:

- for every initiating signal signature in *Y* there is a corresponding initiating signal signature in *X* with the same name, with the same number and names of parameters, and that each parameter type in *X* is a subtype of the corresponding parameter type in *Y*;
- for every responding signal signature in *X* there is a corresponding responding signal signature in *Y* with the same name, with the same number and names of parameters, and that each parameter type in *Y* is a subtype of the corresponding parameter type in *X*.

7.2.4.2 Signature subtyping rules for stream interfaces

Stream signature subtyping rules depend upon the details of the interactions abstracted in the definition of the stream interfaces concerned. In particular these details will clarify whether or not the subtyping rules will permit incomplete correspondences between the set of flows in the two interfaces. Therefore complete signature subtyping rules are beyond the scope of this Reference Model. The constraints on stream signature subtyping are given in Annex A. For stream interface types that are not defined recursively, the constraints are summarized below.

Stream interface *X* is a signature subtype of stream interface *Y* if the conditions below are met for all flows which have identical names:

- if the causality is producer, the information type in *X* is a subtype of the information type in *Y*;
- if the causality is consumer, the information type in *Y* is a subtype of the information type in *X*.

7.2.4.3 Signature subtyping rules for operation interfaces

The definition of operation interface signature subtyping is given in Annex A. For interface types that are not defined recursively, the rules are summarized below.

Operation interface *X* is a signature subtype of interface *Y* if the conditions below are met:

- for every interrogation in *Y*, there is an interrogation signature in *X* (the corresponding signature in *X*) which defines an interrogation with the same name;
- for each interrogation signature in *Y*, the corresponding interrogation signature in *X* has the same number and names of parameters;
- for each interrogation signature in *Y*, every parameter type is a subtype of the corresponding parameter type of the corresponding interrogation signature in *X*.
- the set of termination names of an interrogation signature in *Y* contains the set of termination names of the corresponding interrogation signature in *X*;
- for each interrogation signature in *Y*, a given termination in the corresponding interrogation signature in *X* has the same number and names of result parameters in the termination of the same name in the interrogation signature in *Y*;
- for each interrogation signature in *Y*, every result type associated with a given termination in the corresponding interrogation signature in *X* is a subtype of the result type (with the same name) in the termination with the same name in *Y*;
- for every announcement in *Y*, there is an announcement signature in *X* (the corresponding signature in *X*) which defines an announcement with the same name;

- for each announcement signature in *Y*, the corresponding announcement signature in *X* has the same number and names of parameters;
- for each announcement signature in *Y*, every parameter type is a subtype of the corresponding parameter type n the corresponding announcement signature in *X*.

7.2.5 Template rules

7.2.5.1 Computational object template rules

A computational object (including the special case of a binding object) can:

- initiate or respond to signals;
- produce or consume flows;
- initiate operation invocations;
- respond to operation invocations;
- initiate operation terminations;
- respond to operation terminations;
- instantiate interface templates;
- instantiate object templates;
- bind interfaces;
- access and modify its state;
- delete one or more of its interfaces;
- delete itself;
- spawn, fork and join activities;
- obtain a computational interface identifier for an instance of the trading function;
- test if one computational interface signature is a subtype of another.

Any of these actions can fail.

7.2.5.2 Computational interface instantiation

Computational interface instantiation establishes one or more computational interface identifiers for the new interface in the object performing the instantiation.

7.2.5.3 Computational object template instantiation

The behaviour expression in a computational object template includes a description of the behaviour to occur when the template is instantiated (the **instantiated behaviour**). The environment contract specification describes the contract to be established between the instantiated object and its environment when the template is instantiated. Where the instantiation behaviour includes interface instantiation, the instantiation establishes identifiers for these interfaces in the object which initiated the instantiation.

7.2.6 Failure rules

The failure modes visible to an object are determined by its behaviour and environment contract specifications.

Any of the computational actions in 7.2.5.1 can fail and that failure can be observed by the object performing the action. Interaction can be disrupted by failure of the objects involved, or the binding between them, or both. In the case of signals the failure is identical for, and visible to, all participants in the interaction. In the case of flows and operations, failure need not occur in all the participants, and may occur at different times and with different parameters for each failing participant.

NOTE – Examples of interaction failure include security failure, communication failure and resource failure.

For operations, the failure of the server computational to respond to invocations or to initiate terminations can be observed by the client computational object involved.

The instantiation of an object template or a computational interface template fails if the environment contract cannot be satisfied. A binding action fails if any of the environment contracts in the interfaces being bound cannot be satisfied.

7.2.7 Portability rules

Standards for portability in ODP systems specify action templates for the actions described in 7.2.5.1. The specification of such templates is a matter of language design and is therefore beyond the scope of this Reference Model. In addition to syntactic concerns, a portability standard must address specific semantic issues including:

- composition rules for action templates, including templates for forking and joining actions to enable concurrency and synchronization;
- the terms available for specification of object and interface templates, together with rules for their composition;
- ordering and delivery guarantees for announcements.

A portability standard may represent the permitted actions directly (e.g. as library functions) or indirectly through syntactic structures. There can be alternative portability standards both in terms of style (e.g. an event based processing model versus a threads based model) and content (e.g. the number of computational actions supported). This Reference Model identifies two.

A **basic portability standard** is one which includes at least:

- interrogations;
- implicit binding;
- computational object instantiation;
- computational interface instantiation;
- access to and modification of state;
- support for threads with spawn, fork and join actions;
- obtaining an identifier for a computational interface at which the trading function is provided (to enable subsequent binding to and use of the function);
- interface signature subtype testing.

An **extended portability standard** is one which includes all the actions described in 7.2.5.1.

7.3 Conformance and reference points

In the computational language there exists a reference point at any interface of any object. Each reference point can become either a programmatic conformance point, a perceptual conformance point, an interworking conformance point or an interchange conformance point, depending upon the requirements set when the reference point is designated as a conformance point by a specific standard or in the system specification.

In the computational language these requirements are specified in terms of interface and object templates which determine the interface of the conforming object.

An implementor claiming conformance to a computational specification must list the engineering reference points which correspond to the required computational reference point and state which transparency and engineering structures apply to them. By this act the identified reference points become conformance points. The set of interactions at these conformance points can then be interpreted in computational language terms to determine that the computational specification is not violated.

Conformance of an object at a programmatic conformance point can be tested in terms of a standardized interface specification language and a language binding which satisfies the portability rules. Conformance of an object at an interworking conformance point can be tested in terms of interactions visible in communications protocols.

8 Engineering language

The engineering language comprises concepts, rules and structures for the specification of an ODP system from the engineering viewpoint.

An engineering specification defines the mechanisms and functions required to support distributed interaction between objects in an ODP system.

8.1 Concepts

The engineering language contains the concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and those defined here, subject to the rules of 8.2.

8.1.1 Basic engineering object: An engineering object that requires the support of a distributed infrastructure.

8.1.2 Cluster: A configuration of basic engineering objects forming a single unit for the purposes of deactivation, checkpointing, reactivation, recovery and migration.

NOTE – A segment of virtual memory containing objects is an example of a cluster.

8.1.3 Cluster manager: An engineering object which manages the basic engineering objects in a cluster.

8.1.4 Capsule: A configuration of engineering objects forming a single unit for the purpose of encapsulation of processing and storage.

NOTE – A virtual machine (e.g. a process) is an example of a capsule.

8.1.5 Capsule manager: An engineering object which manages the engineering objects in a capsule.

8.1.6 Nucleus: An engineering object which coordinates processing, storage and communications functions for use by other engineering objects within the node to which it belongs.

NOTE – An operating system (kernel) is an example of a nucleus.

8.1.7 Node: A configuration of engineering objects forming a single unit for the purpose of location in space, and which embodies a set of processing, storage and communication functions.

NOTES

1 A computer and the software it supports (operating system and applications) is an example of a node.

2 A node may have internal structure which is not of concern in an engineering specification. Thus a node can be, for example, a parallel computer under the control of a single operating system.

8.1.8 Channel: A configuration of stubs, binders, protocol objects and interceptors providing a binding between a set of interfaces to basic engineering objects, through which interaction can occur.

NOTE – Bindings that require channels are referred to as **distributed bindings** in the engineering language; bindings between engineering objects that do not require channels (e.g. between engineering objects in the same cluster) are referred to as **local bindings**.

8.1.9 Stub: An engineering object in a channel, which interprets the interactions conveyed by the channel, and performs any necessary transformation or monitoring based on this interpretation.

NOTE – For example, a stub can perform marshalling of parameters into communications buffers, or the recording of activity for audit purposes.

8.1.10 Binder: An engineering object in a channel, which maintains a distributed binding between interacting basic engineering objects.

8.1.11 <x> Interceptor: An engineering object in a channel, placed at a boundary between <x> domains. An <x> interceptor:

– performs checks to enforce or monitor policies on permitted interactions between basic engineering objects in different domains;

– performs transformations to mask differences in interpretation of data by basic engineering objects in different domains.

NOTE – An inter-subnetwork relay is an example of an interceptor.

8.1.12 Protocol object: An engineering object in a channel, which communicates with other protocol objects in the same channel to achieve interaction between basic engineering objects (possibly in different clusters, possibly in different capsules, possibly in different nodes).

8.1.13 Communications domain: A set of protocol objects capable of interworking.

8.1.14 Communication interface: An interface of a protocol object that can be bound to an interface of either an interceptor object or another protocol object at an interworking reference point.

8.1.15 Binding endpoint identifier: An identifier, in the naming context of a capsule, used by a basic engineering object to select one of the bindings in which it is involved, for the purpose of interaction.

NOTES

1 A memory address (for a data structure representing an engineering interface) is an example of a binding endpoint identifier.

2 The same form of binding endpoint identifier can be used, whether the binding involved is either local or distributed.

8.1.16 Engineering interface reference: An identifier, in the context of an engineering interface reference management domain, for an engineering object interface that is available for distributed binding.

NOTE – An engineering interface reference is necessary to establish distributed bindings, and is distinct from the binding endpoint identifiers used by a basic engineering object for the purposes of interaction.

8.1.17 Engineering interface reference management domain: A set of nodes forming a naming domain for the purpose of assigning engineering interface references.

8.1.18 Engineering interface reference management policy: A set of permissions and prohibitions that govern the federation of engineering interface reference management domains.

8.1.19 Cluster template: An object template for a configuration of objects and any activity required to instantiate those objects and establish initial bindings.

8.1.20 Checkpoint: An object template derived from the state and structure of an engineering object that can be used to instantiate another engineering object, consistent with the state of the original object at the time of checkpointing.

8.1.21 Checkpointing: Creating a checkpoint. Checkpoints can only be created when the engineering object involved satisfies a pre-condition stated in a checkpointing policy.

8.1.22 Cluster checkpoint: A cluster template containing checkpoints of the basic engineering objects in a cluster.

8.1.23 Deactivation: Checkpointing a cluster, followed by deletion of the cluster.

8.1.24 Cloning: Instantiating a cluster from a cluster checkpoint.

8.1.25 Recovery: Cloning a cluster after cluster failure or deletion.

8.1.26 Reactivation: Cloning a cluster following its deactivation.

8.1.27 Migration: Moving a cluster to a different capsule.

8.2 Structuring rules

An engineering specification defines the infrastructure required to support functional distribution of an ODP system, by:

– identifying the ODP functions required to manage physical distribution, communication, processing and storage;

– identifying the roles of different engineering objects supporting the ODP functions (for example the nucleus).

An engineering specification is expressed in terms of:

– a configuration of engineering objects, structured as clusters, capsule and nodes;

– the activities that occur within those engineering objects;

– the interactions of those engineering objects.

An engineering specification is constrained by the rules of the engineering language. These comprise:

– channel rules (see 8.2.1), interface reference rules (see 8.2.2), distributed binding rules (see 8.2.3) and relocation rules (see 8.2.4) for the provision of distribution transparent interaction among engineering objects;

– cluster rules (see 8.2.5), capsule rules (see 8.2.6) and node rules (see 8.2.7) governing the configuration of engineering objects;

– failure rules (see 8.2.9).

8.2.1 Channel rules

A channel supports distribution transparent interaction between engineering objects. This includes, for example:

- operation execution between a client object and a server object;
- a group of objects multicasting to another group of objects;
- stream interactions involving multiple producer objects and multiple consumer objects.

Interaction between engineering objects entails the transfer of some or all of:

- engineering interface references;
- cluster templates;
- data.

A channel is a configuration of stubs, binders, protocol objects and interceptors interconnecting a set of engineering objects. The configuration is an acyclic graph with stub objects at the outermost vertices as illustrated in Figures 2 and 3. Each path through the graph between stub objects consists of, in sequence, either

- a binder, a protocol object, a protocol object and a binder; or
- a binder, a protocol object, an interceptor, a protocol object and a binder.

The behaviour of a channel with respect to channel configuration or Quality of Service management is controlled via control interfaces of stubs, binders, protocol objects and interceptors. Such control interfaces are optional.

NOTES

1 Stubs, binders, protocol objects and interceptors in a channel can have (local or distributed) bindings to other engineering objects outside the channel providing, for example, relocation or coordination functions.

2 Depending upon the kind of transformation involved, an interceptor may be decomposed into stubs, binders, protocol objects and basic engineering objects, mirroring the channel structure.

Objects in a channel can themselves be basic engineering objects supported by channels.

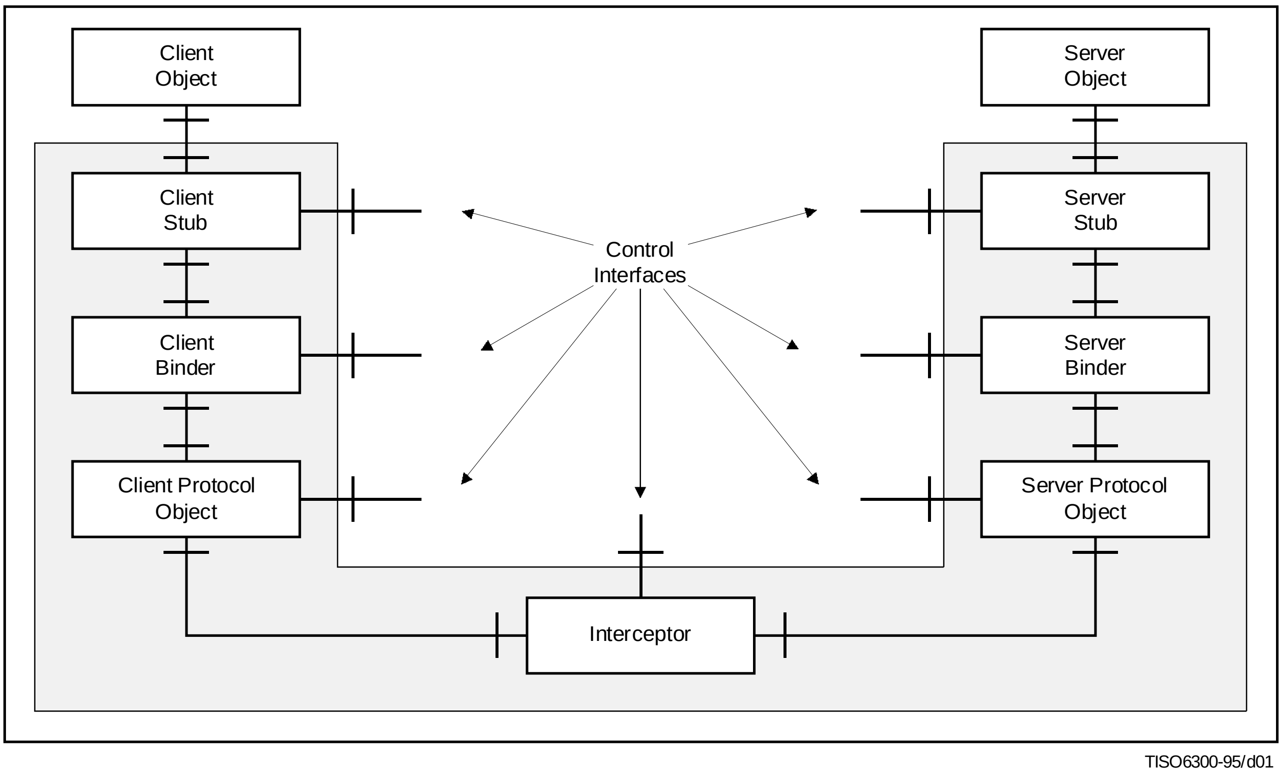


Figure 2 – An example of a basic client/server channel

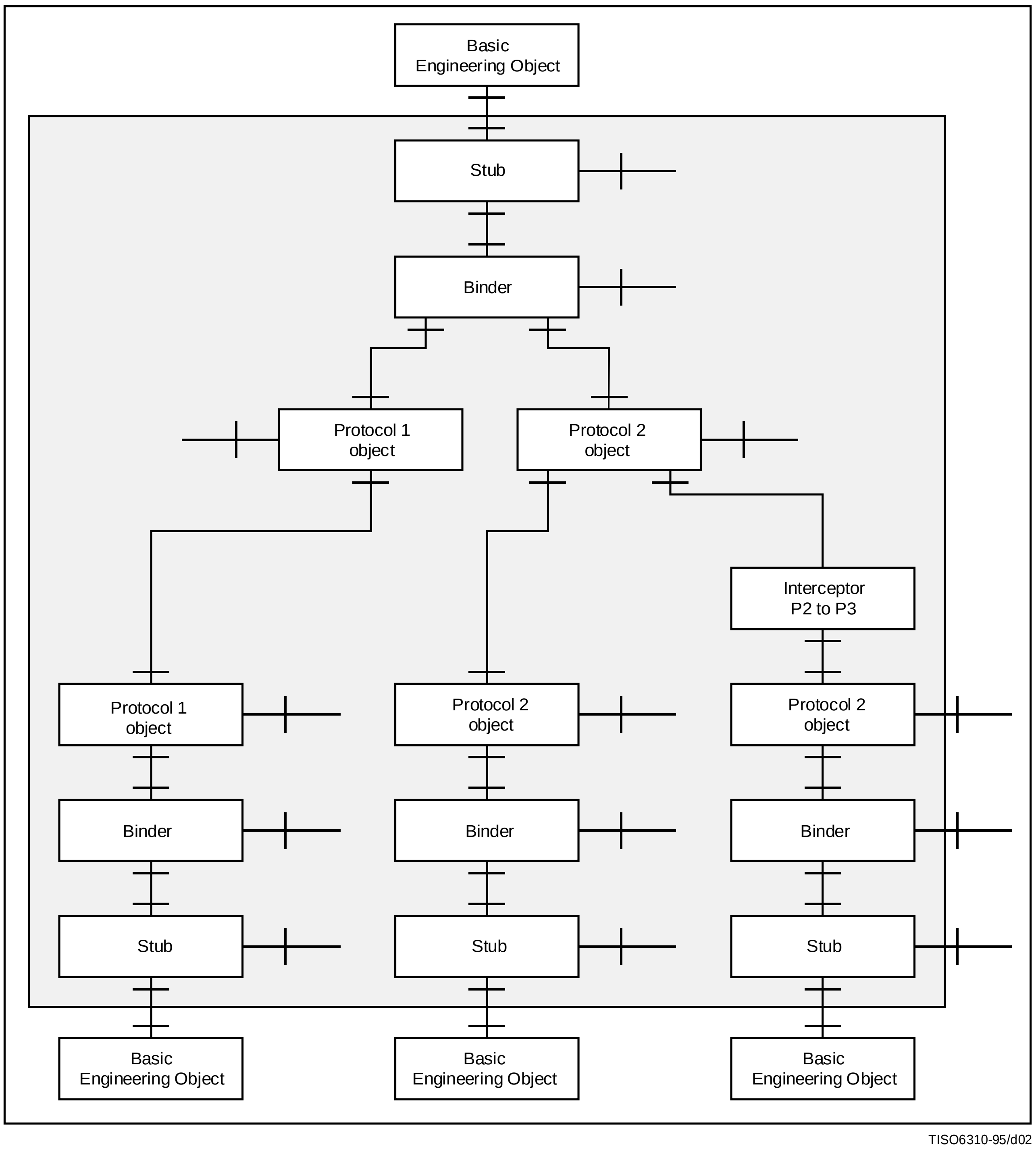


Figure 3 – An example of a multi-endpoint channel

8.2.1.1 Stubs

Basic engineering objects which interact via channels are locally bound to stubs. In a channel, stubs provide the conversion of data carried by interactions. Stubs can apply controls and keep records (e.g. for security and accounting). Stubs can interact with engineering objects outside the channel (e.g. a security function) if required. A stub in a channel has an interface for use by the basic engineering object it supports and an interface for interaction with a binder. It can also have a control interface.

When interconnected stubs use different transfer syntaxes, there must be an interceptor in the path between them which can transform data from one syntax to the other.

A stub can be of one of the following forms:

a) specific to the basic engineering object *interface instance* to which it is bound;

b) specific to the *interface type* of the basic engineering object interface to which it is bound (therefore the stub may be shared between a number of channels of the same type);

c) generic (i.e. not specific to any single interface type); such stubs may be shared between a number of channels of different types.

NOTES

1 In a) interactions between the engineering object and the stub only convey interaction data (e.g. operation names and parameters in the case of an invocation). The stub acts as a local proxy for other basic engineering objects bound to the channel.

2 In b) interactions must additionally include an identifier for the channel to be used.

3 In c) interactions must additionally include an identifier and type for the channel to be used, to enable the stub to ensure the interaction data are compatible with the channel type.

8.2.1.2 Binders

The binders in a channel manage the end-to-end integrity of that channel. When required, binders provide relocation transparency by monitoring communication failures and repairing broken distributed bindings. Binders in a channel can interact with engineering objects outside the channel to obtain additional data needed to perform their function (e.g. a relocator to obtain location data). A binder in a channel has at least an interface for interaction with a stub and one or more interfaces for interaction with protocol objects. A binder can have a control interface. If present, the control interface enables changes in the configuration of the channel and deletion of the channel.

8.2.1.3 Protocol objects

Protocol objects provide communication functions. Protocol objects can interact with engineering objects outside a channel (e.g. directory functions) to obtain the information they need. A protocol object has an interface for interaction with a binder and at least one communications interface for interaction with other protocol objects (via interceptors if required). A protocol object can have a control interface. When protocol objects in a channel are of different type, they require an interceptor which provides protocol conversion. All protocol objects in a communication domain can communicate directly using facilities of the communications domain (which fall outside the scope of this Reference Model).

At any given location in time, a protocol object is identified by its location in space, but different protocol objects can occupy the same location in space at different times (i.e. network addresses can be recycled). When protocol objects in a channel are of the same type, but in disjoint communications domains, naming conflicts are possible (e.g. names for communications interfaces might be ambiguous). In this circumstance, an interceptor is required to transform names exchanged in the process of establishing and maintaining the integrity of the channel.

8.2.1.4 Interceptors

An <x> interceptor in a channel stands at the boundary between <x> domains and provides checks and transformations on interactions that cross <x> domain boundaries. Depending upon the boundary crossed, interceptors require different information to perform their task. Some interceptors will need to know the signature types of the basic engineering object interfaces bound to the channel in which the interceptor is located so that they can interpret interactions supported by the channel. An interceptor has at least two communication interfaces. It can have a control interface.

8.2.2 Interface reference rules

For the purpose of distributed binding, engineering interfaces are located in space and time by engineering interface references. Engineering interface references are defined relative to an engineering interface reference management domain which determines policy for the content, allocation, tracking and validation of engineering interface references within that domain. An engineering interface reference management domain consists of a set of nodes. Engineering interface reference domains can be federated if their engineering interface reference management policies do not conflict.

An engineering interface reference contains information which enables bindings to be established to engineering object interfaces. This information both enables nucleus objects to create channels and also enables binders within channels to maintain a distributed binding between interconnected engineering objects. The information within an engineering interface reference can take the form of:

- data;
- identifiers for interfaces giving access to such data;
- a combination of data and identifiers.

The data necessary for binding can include any or all of the following items:

- the interface type of the referenced interface;
- a channel template describing the interceptors, protocol objects, binders and stubs that can be selected when configuring a channel to support the distributed binding;
- the location in space and time (e.g. a network address) of the communication interfaces at which the binding process can be initiated;
- information to enable the detection and repair of distributed bindings invalidated by engineering object relocation.

An interface reference management domain can be divided into subdomains. In this case interface references within the domain are organized as a set of alternative sets of information, one for each subdomain in which binding is to be enabled.

NOTE 1 – If the nucleus supporting the engineering interface supports different protocols, binding processes and transfer syntaxes, the engineering interface reference will indicate the valid combinations that can be selected in any particular distributed binding; different bindings might make different selections.

NOTE 2 – This Reference Model does not prescribe the method by which a channel template and the location in space and time of related interfaces are derived from an engineering interface reference.

Engineering interface references are allocated by nuclei, at interfaces supporting the node management function (see 12.1.3). Engineering interface references are tracked by the engineering interface reference tracking function (see 13.9) for the purposes of detecting engineering interfaces which are no longer referenced. Policy for rebinding to engineering object interfaces which have been relocated, and updated engineering interface references for engineering interfaces which have been relocated, are recorded by the relocation function (see 14.3). These three functions (node management, engineering interface reference tracking and relocation) may be coordinated by use of a shared information organization function (see 14.2).

Engineering interface references are unambiguous in the naming context of an engineering interface reference management domain. To achieve this unambiguity, the nodes within the engineering interface reference management domain must allocate engineering interface references in a coordinated way. Engineering interfaces must be allocated in a manner which prevents an engineering interface reference from referencing the wrong interface, even in the presence of failures and interface relocation. At worst an interface reference will refer to an interface which does not exist (e.g. immediately after failure of the engineering object supporting the interface).

NOTE 3 – In ODP systems in which most interfaces do not change location, the management of interface references can be optimized: the nucleus can allocate engineering interface references autonomously; the channel type and communications interface identifier associated with the interface can be stored and transmitted within the engineering interface reference; the relocation function can be used to validate and update engineering interface references for interfaces that have been relocated.

Before issuing an engineering interface reference, the nucleus constructs a channel template which defines a configuration of stub, binder and protocol objects suitable to support interactions in the interface. In addition the nucleus establishes sufficient local structure to make binding to the interface possible, and associates the template and the local structure with a communications interface. The engineering interface reference makes this information available.

An interceptor that stands on a boundary between engineering interface reference management domains maintains mappings between engineering interface references in those domains. When an engineering interface reference or a cluster template containing engineering interface references crosses an engineering interface reference domain boundary, the engineering interface references involved must be transformed so as to be valid in the new domain.

Media interchange of engineering interface references from one interface reference management domains to another is only possible when there is a defined procedure for mapping the references to avoid ambiguity.

8.2.3 Distributed binding rules

Establishing a channel requires the creation of appropriate stubs, binders, protocol objects and interceptors. Channel establishment can be initiated by any engineering object. It is provided by each nucleus as a function of its node management interface. Distributed binding involves interaction with the nuclei of the nodes at which the interfaces to be bound are located. Channel establishment is parameterized by a channel template and a set of interface references each assigned to a particular role in the channel template. The channel template must be compatible with the channel types nominated by the engineering interface references for the interfaces to be bound. The nucleus for each object to be bound creates a configuration of stubs, binders and protocol objects at its node to support the interfaces of that object being bound. This includes configuration of their control interfaces. The protocol objects that support the channel are connected (possibly via interceptors) at their communication interfaces. The selection and configuration of stubs, binders, protocol objects and interceptors is determined by the channel template and channel types of the interface references involved. Each basic engineering object bound by the channel is assigned a binding endpoint identifier for each interface it has to the channel. Basic engineering objects use binding endpoint identifiers to nominate at which of their interfaces a distributed interaction is to occur.

NOTES

1 Channels can be established by any engineering object irrespective of whether or not the object has an interface that is to be bound by the channel.

2 A basic engineering object initiating a distributed binding requires a set of interface references. These may be obtained in any of the following ways:

(a) on initialization of the object;

(b) by interaction between the initiating object and the nucleus as part of the instantiation of the initiating object's interfaces;

(c) through some chain of interactions with the other objects concerned (e.g. by parameter passing or trading).

3 A channel template can contain alternative configurations to be applied in selected circumstances. For example if communication paths are insecure, encrypting stubs might be required.

8.2.4 Relocation rules

Engineering objects can be relocated as a result of:

– reactivation and deactivation;

– checkpointing and recovery;

– migration;

– communications domain management functions (e.g. changing a communication interface identifier).

NOTES

1 A communications interface may have its identifier changed as a consequence of changing the network address of a node.

2 When channels are re-established, stubs, binders and protocol objects might be used which are different to those used prior to relocation. Thus, an identifier for a communications interface is not necessarily sufficient to identify an engineering object interface.

Relocation can cause channels to fail and can invalidate engineering interface references. Failed channels can be repaired if the activity which changes the location of the engineering object interface notifies the relocation function appropriately (see 14.3).

The binders in a channel detect when relocation has invalidated the channel. Either the binders collaborate to correct the mapping between engineering interface references and the channel structure (i.e. relocation transparency is required – see 16.6), or the channel fails. When relocation transparency is required the information available through the engineering interface reference enables the binders to use the relocation function to determine the new locations of the basic engineering objects involved.

8.2.5 Cluster rules

A cluster contains a set of basic engineering objects, associated with a cluster manager. Each member of a cluster can have an interface supporting the object management function. Each such object management interface is bound to the cluster's cluster manager. A basic engineering object in a cluster is always bound to its nucleus, at an interface providing the node management function and to its cluster manager. In addition, a basic engineering object in a cluster can be bound to other basic engineering objects in the same cluster, or in other clusters. Each cluster manager in a capsule is bound to the capsule manager for that capsule. This structure is illustrated in Figure 4.

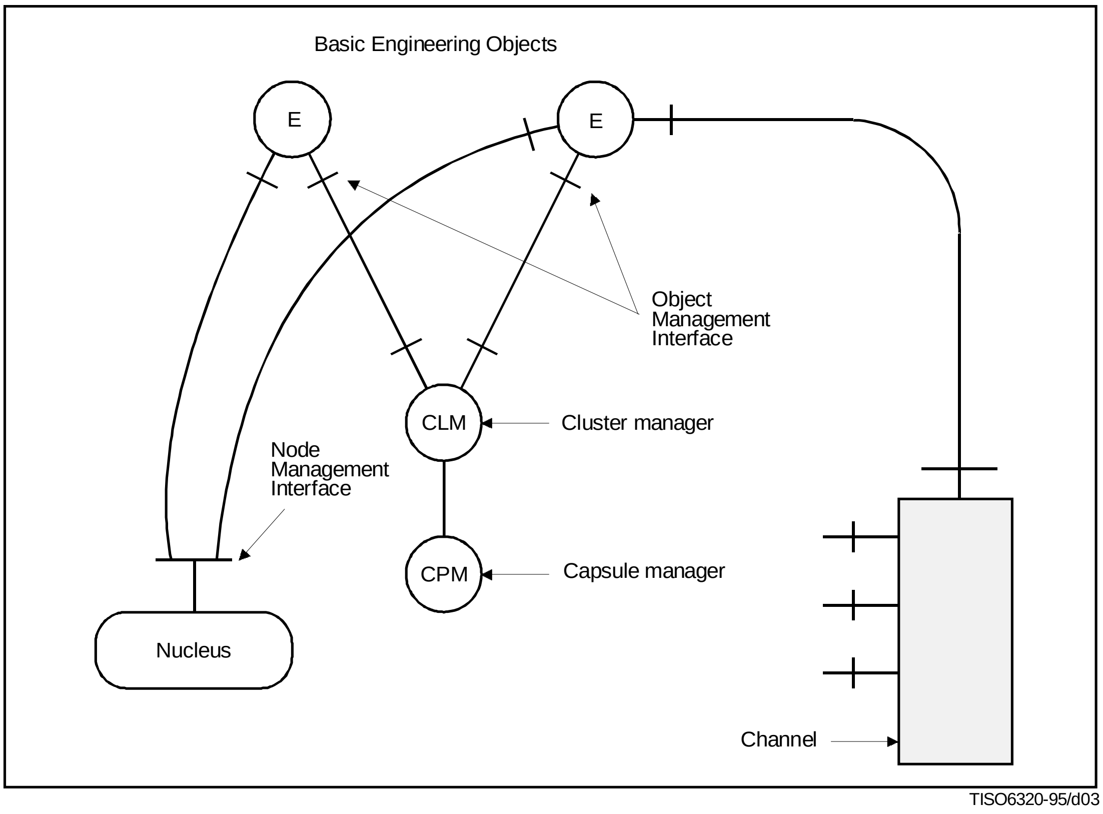

Basic Engineering Objects
E
E
Object Management Interface
CLM
Cluster manager
Node Management Interface
CPM
Capsule manager
Nucleus
Channel
TISO6320-95/d03

Figure 4 – Example structure supporting a basic engineering object

A cluster is always contained in a single capsule. A cluster is responsible for its own security, but can be assisted by security functions. Any assisting security function must either be provided by an object in the same capsule as that cluster, or be accessed by secure interactions if it is outside the capsule. Engineering objects in the same cluster can interact using a local binding within the cluster, or using a distributed binding supported by a channel. Engineering objects in different clusters interact using distributed bindings supported by channels.

NOTES

1 Interactions that are at perceptual or interchange reference points are not precluded.

2 Although no channel is needed to support local bindings between objects in the same cluster, the identifier used to perform invocations is of the same kind as is used between engineering objects in different clusters, and is still called a binding endpoint identifier.

Cluster instantiation (including cloning as a special case) is performed by a capsule manager.

If the template is a cluster checkpoint, the instantiation (i.e. cloning) enables the new cluster to act as a substitute for the original cluster from which the cluster template was derived. When required for distribution transparency reasons, the cloning process includes re-establishing any distributed bindings that were held by the original cluster.

A cluster has an associated cluster manager. The cluster manager provides the cluster management function. The cluster manager embodies the management policy for the engineering objects in its cluster. The cluster management policy might lead the cluster manager to interact with other ODP functions to complete cluster management activities.

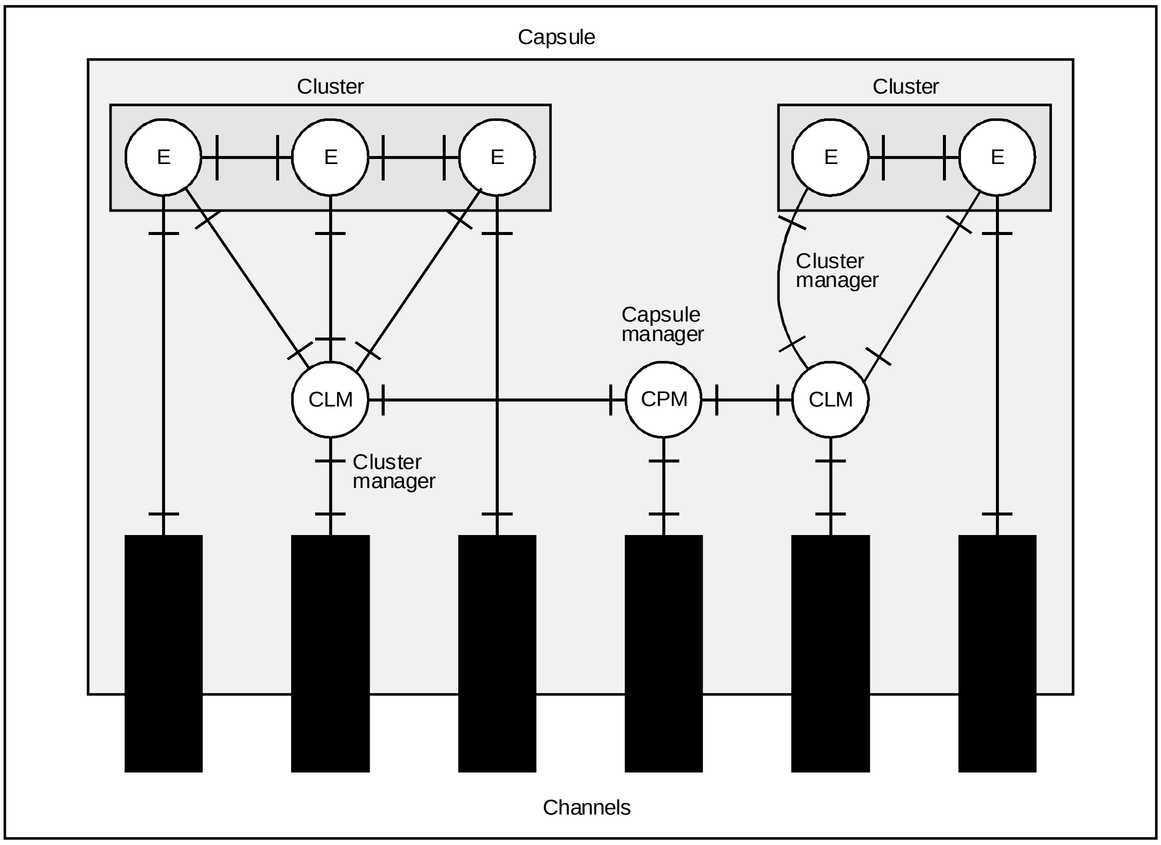
A cluster manager assists its capsule manager in the management of the engineering interface references of objects. This may require access to the engineering interface reference tracking function.

8.2.6 Capsule rules

A capsule consists of

- cluster(s);
- cluster managers, one for each cluster in the capsule;
- a capsule manager to which each of the cluster managers in the capsule is bound.

Stubs, binders and protocol objects for a channel bound to an interface of a basic engineering object within a cluster in a capsule can be included within that capsule. All the engineering objects in a capsule are bound to the same node management interface. Engineering objects in other capsules are bound to different node management interfaces. A capsule is contained within a node. A capsule has a capsule manager. The capsule manager is bound, at an interface providing the cluster management function, to each cluster manager in the capsule. This structure is illustrated in Figure 5 (the figure abstracts out details of the nucleus).



TISO6330-95/d04

Figure 5 – Example structure of a capsule

Capsule instantiation is performed by the nucleus using a capsule template which specifies the initial configuration of engineering objects in the capsule, including clusters, cluster managers, stubs, binders, protocols and a capsule manager.

A capsule is a naming context for binding endpoint identifiers. The Reference Model does not require that such identifiers be valid in any larger context. Engineering interface references are used to communicate knowledge of engineering object interfaces between capsules (for the purposes of binding).

The capsule manager embodies the management policy for the clusters in its capsule. The capsule management policy might lead the capsule manager to interact with other functions to complete capsule management activities. The capsule manager has an interface providing the capsule management function. The structures supporting interaction between cluster managers, capsule managers and the nucleus within a node are a matter of implementation detail outside the scope of this Reference Model.

8.2.7 Node rules

A node consists of one nucleus and a set of capsules. All of the engineering objects in a node share common processing, storage and communication functions.

A node is a member of one or more engineering interface reference management domains.

The nucleus provides a set of node management interfaces, one to each capsule within the node.

The structure of a node is illustrated in Figure 6.

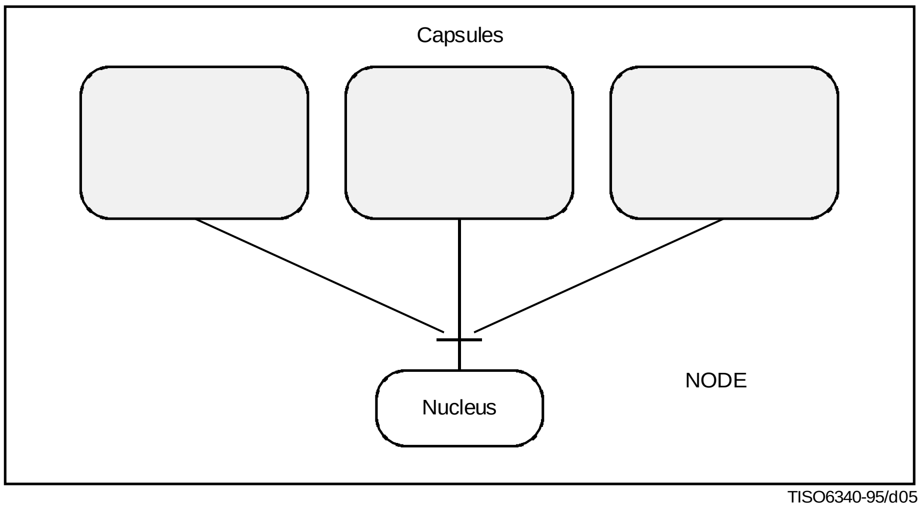


Figure 6 – Example structure of a node

The procedure for node instantiation is outside the scope of this Reference Model: it must result in:

- introduction of the node’s nucleus and its associated processing, storage and communications functions, including introduction of node management functions to enable distributed binding of engineering interface references;
- introduction of any trading function needed by the instantiation process;
- instantiation of any channels required as part of the initial configuration of the node (e.g. to supporting engineering objects such as a relocator).

The set of protocol objects introduced during node instantiation determine the initial set of communication domains to which the node belongs. The nucleus provides the node management function and embodies a node management policy. This policy might lead the nucleus to interact with other ODP functions to complete node management activities. A capsule is the basic unit for the application of node management policy; although an individual object can use the node management function, it is subject to the policies applicable to its capsule. Distinct capsules can be subject to different node management policies.

8.2.8 Application management rules

The management of the lifecycle (creation, migration, deactivation, reactivation, checkpointing, failure, recovery and deletion) of coordinated sets of clusters is driven by application-specific management policies. Application management policies can apply to individual clusters or to coordinated sets of clusters. A set of managed clusters form an **application management domain**. Applications management policy can be implemented by applications-specific management functions which effect changes using the mechanisms provided by the coordination and management functions defined in this reference model, e.g. the cluster management function.

Where appropriate (i.e. depending upon which distribution transparencies apply), application-specific management functions can receive notifications of significant events affecting the clusters they manage and takes action in response to those notifications. For example binding failure reports can lead to reactivation of a cluster, or excessive workload reports can lead to migration of clusters. Requests and notifications relating to one cluster in an application management domain can lead to application-specific management functions initiating lifecycle actions on other clusters in the domain.

The specifics of application domain management are beyond the scope of this Reference Model.

8.2.9 Failure rules

Failures can be categorized as involving clusters, capsules, nodes or communications domains. The analysis of failures is based on the fact that:

- a failure localized to a cluster can be detected by its cluster manager;
- a failure localized to a capsule can be detected by its capsule manager;
- the failure of a node may be detected by protocol objects at other nodes to which it is interconnected.
- the failure of a communications domain may be detected by protocol objects in other communication domains to which it is interconnected.

NOTE – There is an inherent ambiguity in communication systems which may prevent a protocol object from distinguishing between communication failure and remote node failure.

8.3 Conformance and reference points

There is a programmatic reference point at an interaction point between a cluster manager and a basic engineering object.

There is a programmatic reference point at an interaction point between an engineering object and a nucleus.

There is a programmatic reference point at an interaction point between basic engineering objects.

There can be a perceptual or an interchange reference point at an interface of a basic engineering object.

Where interaction between basic engineering objects is provided by a channel, then, for each basic engineering object involved, there is a programmatic reference point at the interaction points between those objects and the corresponding stubs in the channel.

Within the configuration of objects that comprise a channel, there is a programmatic reference point at each of the following interaction points within the channel:

- between stubs (abstracting the binders, protocol objects and interceptors in the channel between the stubs);
- between stubs and binders;
- between binders (abstracting the protocol objects and interceptors in the channel between the binders);
- between binders and protocol objects;
- between protocol objects and other protocol objects within the same node (abstracting the interceptors, if any, between the protocol objects),

and there is an interworking reference point at any interaction point between protocol objects and other protocol objects or interceptors in different nodes.

Control interfaces of stubs, binders, protocol objects and interceptors are programmatic reference points.

Where engineering objects within a channel interact with other engineering objects (either within that channel or outside that channel) via interfaces which are not within that channel, then the reference points applicable to such interfaces are determined by the recursive application of these rules.

By defining conformance at the interworking reference points, interworking of systems is made possible.

By defining conformance at the programmatic reference points, portability of engineering objects between systems is made possible.

Conformance of individual engineering objects at the programmatic reference points does not itself guarantee that the engineering object will be portable to all systems or will interwork with matching engineering objects bound to other nuclei.

9 Technology language

A technology specification defines the choice of technology for an ODP system.

9.1 Concepts

The technology language consists of the concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and the concepts defined here subject to the structuring rules in 9.2.

9.1.1 **Implementable standard**: A template for a technology object.

9.1.2 **Implementation**: A process of instantiation whose validity can be subject to test.

9.1.3 **IXIT**: Implementation eXtra Information for Testing.

9.2 Structuring rules

A technology specification defines the choice of technology for an ODP system in terms of:

- a configuration of technology objects; and
- the interfaces between them.

A technology specification:

- expresses how the specifications for an ODP system are implemented;
- identifies specifications for technology relevant to the construction of ODP systems;
- provides a taxonomy for such specifications;
- specifies the information required from implementors to support testing.

When applying a specification written in another viewpoint language, a technology specification is constructed to give an interpretation of the atomic terms in the other viewpoint specification.

The technology specification for an ODP function can reference the specifications of other ODP functions.

A technology specification consists of statements that technology objects are instances of named implementable standards.

The technology specification gives a proforma for the conformance IXIT by listing the required set of templates and the descriptive names for all necessary reference points.

All implementable standards are introduced by reference to other specifications. The technology language does not define any other rules constraining the behaviour of technology objects or for constructing implementable standards.

9.3 Conformance and reference points

The technology language is used to assert that technology objects are instances of implementable standards; these standards will, in general, contain conformance statements.

10 Consistency rules

A set of specifications of an ODP system written in different viewpoint languages should not make mutually contradictory statements (see 4.2.2), i.e. they should be mutually consistent. Thus, a complete specification of a system includes statements of correspondences between terms and language constructs relating one viewpoint specifications to another viewpoint specification, showing that the consistency requirement is met. The minimum requirements for consistency in a set of specifications for an ODP system is that they should exhibit the correspondences defined in this Reference Model and those defined within the set of specifications itself. This Reference Model does not declare generic correspondences between every pair of viewpoint languages. This clause is restricted to the specification of correspondences between a computational specification and an information specification, and between a computational specification and an engineering specification. In each case, the correspondences are expressed as interpretation relationships linking terms in one viewpoint language to terms in the other viewpoint language. A set of specifications based on this Reference Model will, in general, need to relate all the viewpoint specifications.

The key to consistency is the idea of correspondences between specifications, i.e. a statement that some terms or structures in one specification correspond to other terms and specifications in a second specification. Correspondences can be established between two different specifications in a single language or in two different languages. Statements of correspondences between two languages imply equivalent correspondences between any pair of specifications expressed in those languages.

Analysis of consistency depends on the application of a specific consistency techniques. Most of these are based on checks for particular kinds of inconsistency, and thus cannot prove absolute consistency. One form of consistency involves a set of correspondence rules to steer a transformation from one language to another. Thus, given a specification S_1 in viewpoint language L_1 and specification S_2 in viewpoint language L_2, where S_1 and S_2 both specify the same system, a transformation T can be applied to S_1 resulting in a new specification $T(S_1)$ in viewpoint language L_2 which can be compared directly to S_2 to check, for example, for behavioural compatibility between allegedly equivalent objects or configurations of objects.

10.1 Computational and information specification correspondences

This Reference Model does not prescribe exact correspondences between information objects and computational objects. In particular, not all states of a computational specification need correspond to states of an information specification. There may exist transitional computational states within pieces of computational behaviour which are abstracted as atomic transitions in the information specification.

Where an information object corresponds to a set of computational objects, static and invariant schemata of an information object correspond to possible states of the computational objects. Every change in state of an information object corresponds either to some set of interactions between computational objects or to an internal action of a computational object. The invariant and dynamic schemata of the information object correspond to the behaviour and environment contract of the computational objects.

NOTE – If the concept of information interface is used in an information specification, there is no necessary correspondence between an information interface and any computational interface.

10.2 Engineering and computational specification correspondences

Each computational object which is not a binding object corresponds to a set of one or more basic engineering objects (and any channels which connect them). All the basic engineering objects in the set correspond only to that computational object.

Except where transparencies which replicate objects are involved, each computational interface corresponds exactly to one engineering interface, and that engineering interface corresponds only to that computational interface.

NOTE 1 – The engineering interface is supported by one of the basic engineering objects which corresponds to the computational object supporting the computational interface.

Where transparencies which replicate objects are involved, each computational interface of the objects being replicated correspond to a set of engineering interfaces, one for each of the basic engineering objects resulting from the replication. These engineering interfaces each correspond only to the original computational interface.

Each computational interface is identified by any member of a set of one or more computational interface identifiers. Each engineering interface is identified by any member of a set of one or more engineering interface references. Thus, since a computational interface corresponds to an engineering interface, an identifier for a computational interface can be represented unambiguously by an engineering interface reference from the corresponding set.

Each computational binding (either primitive bindings or compound bindings with associated binding objects) corresponds to either an engineering local binding or an engineering channel. This engineering local binding or channel corresponds only to that computational binding. If the computational binding supports operations, the engineering local binding or channel must support the interchange of at least:

- computational signature names;
- computational operation names;
- computational termination names;
- invocation and termination parameters (including computational interface identifiers and computational interface signatures).

Except where transparencies which replicate objects are involved, each computational binding object control interface has a corresponding engineering interface and there exists a chain of engineering interactions linking that interface to any stubs, binders, protocol objects or interceptors to be controlled in support of the computational binding.

NOTE 2 – The set of control interfaces involved depends on the type of the binding object.

Each computational interaction corresponds to some chain of engineering interactions, starting and ending with an interaction involving one or more of the basic engineering objects corresponding to the interacting computational objects.

Each computational signal corresponds either to an interaction at an engineering local binding or to a chain of engineering interactions which provides the necessary consistent view of the computational interaction.

The transparency prescriptions in clause 16 specify additional correspondences.

NOTE 3 – Basic engineering objects corresponding to different computational objects can be members of the same cluster.

NOTE 4 – In an entirely object-based computational language, data are represented as abstract data types (i.e. interfaces to computational objects).

NOTE 5 – Computational interface parameters (including those for abstract data types) can be passed by reference, such parameters correspond to engineering interface references.

NOTE 6 – Computational interface parameters (including those for abstract data types) can be passed by migrating or replicating the object supporting the interface. In the case of migration such parameters correspond to cluster templates.

NOTE 7 – If the abstract state of a computational object supporting an interface parameter is invariant, the object can be cloned rather than migrated.

NOTE 8 – Cluster templates can be represented as abstract data types, thus strict correspondences between computational parameters and engineering interface references are sufficient. The use of cluster templates or data are important engineering optimizations and therefore not excluded.

11 ODP functions

The ODP functions defined in this Recommendation | International Standard are those that are either fundamental or widely applicable to the construction of ODP systems.

The specifications for individual ODP functions can be combined to form specifications for components of ODP systems. The identification of such components is a matter for the standardization process and is not prescribed by this Reference Model: it contains outline function descriptions written using concepts from ITU-T Rec. X.902 | ISO/IEC 10746-2.

Some function descriptions in this Reference Model introduce objects as a simplifying modelling construct. Except where explicit constraints are given on the distribution of these objects, they do not define necessary structure in an implementation.

The functions defined in this Reference Model are listed below. Those forming an integral part of the computational language are tagged with a "*", those forming an integral part of the engineering language are tagged with a "+":

a) *Management functions:*
   1) node management function^{+};
   2) object management function^{+};
   3) cluster management function^{+};
   4) capsule management function^{+};

b) *Coordination functions:*
   1) event notification function;
   2) checkpointing and recovery function;
   3) deactivation and reactivation function;
   4) group function;
   5) replication function;
   6) migration function;
   7) engineering interface reference tracking function^{+};
   8) transaction function.

c) *Repository functions:*
   1) storage function;
   2) information organization function;
   3) relocation function;
   4) type repository function;
   5) trading function^{+*}.

d) *Security functions:*
   1) access control function;
   2) security audit function;
   3) authentication function;
   4) integrity function;
   5) confidentiality function;
   6) non-repudiation function;
   7) key management function.

12 Management functions

12.1 Node management function

The node management function controls processing, storage and communication functions within a node.

The node management function is provided by each nucleus at one or more node management interfaces.

Each capsule uses a node management interface distinct from the node management interfaces used by other capsules in the same node.

The node management function:

– manages threads;

– accesses clocks and manages timers;

– creates channels and locates interfaces.

Within the architecture defined by this Reference Model, the node management function is used by all other functions.

12.1.1 Thread management

Node management interfaces provides functions to spawn and fork threads within a capsule and to join, delay and synchronize threads within a capsule.

12.1.2 Clock access and timer management

Node management interfaces provides functions to determine the current time in a specified clock management domain and to start, monitor and cancel timers.

12.1.3 Channel creation and interface location

Node management interfaces provides functions to:

a) enable binding between an engineering object within a capsule and an instance of the trading function;

b) make an engineering interface available for binding to objects in other capsules;

c) establish a binding between an engineering object within a capsule and a set of other engineering objects (identified by engineering interface references);

d) from a specified engineering interface reference, determine the channel type and communications interface it implies.

NOTES

1 The use of interactions b) and c) is explained in 8.2.3.

2 Interaction d) exposes the information content of the interface reference, enabling it to be stored or transformed for use in an different interface reference management domain.

Making an interface available for binding to objects in other capsules – interaction b) above – consists of the following actions:

– assigning an engineering interface reference within a nominated engineering interface reference management domain;

– assigning a communications interface through which bindings to the interface can be established;

– assigning a channel type to the interface.

12.1.4 Capsule template instantiation and capsule deletion

Node management interfaces provide functions to instantiate capsule templates and to delete capsules.

Capsule template instantiation consists of the following steps:

– allocation of processing, storage and communication functions for a new capsule in the same node as the nucleus providing the node management interface;

– creating a capsule manager for the new capsule;

– creating a capsule *management* interface in the new capsule manager;

– producing an identifier for the capsule management interface;

– creating a capsule *control* interface for the new capsule in the nucleus;

– producing an identifier for the capsule control interface.

The capsule control interface produced by capsule template instantiation enables deletion of the capsule (e.g. when its manager has failed).

Deletion of a capsule deletes all objects in the capsule.

12.2 Object management function

The object management function checkpoints and deletes objects.

When an object belongs to a cluster that can be deactivated, checkpointed or migrated, the object must have an object management interface in which it provides one or more of the following functions:

– checkpointing the object;

– deleting the object.

NOTES

1 Thus different object management interfaces can have different interface types depending upon which functions they support.

2 Checkpointing an object produces information suitable for incorporation into a cluster checkpoint.

3 When an object is deleted, stubs, binders, protocol objects and interceptors supporting bindings to the object can be deleted.

The object management function is used by the cluster management function.

12.3 Cluster management function

The cluster management function checkpoints, recovers, migrates, deactivates or deletes clusters and is provided by each cluster manager at a cluster management interface, comprising one or more of the following functions with respect to the managed cluster:

– modifying cluster management policy (e.g. for the location of checkpoints of its cluster, for the use of the relocation function to trigger reactivation or recovery of the cluster);

– deactivating the cluster;

– checkpointing the cluster;

- replacing the cluster with a new one instantiated from a cluster checkpoint (i.e. deletion followed by recovery);
- migrating the cluster to another capsule (using the migration function);
- deleting cluster.

NOTE – Thus different cluster management interfaces can have different interface types depending upon which functions they support.

The behaviour of a cluster manager is constrained by the management policy for its cluster. Cluster checkpointing and deactivation is only possible if all objects in the cluster have object management interfaces supporting the object checkpointing function. Deactivation and cluster deletion both require that the objects in the cluster support object deletion.

Within the architecture defined by this Reference Model:

- the cluster management function is used by the capsule management function, the deactivation and reactivation function, the checkpoint and recovery function, the migration function and the engineering interface reference management function;
- the cluster management function uses the storage function for keeping checkpoints.

12.3.1 Cluster checkpoint

A cluster checkpoint contains information needed to re-establish a cluster and includes the following elements:

a) checkpoints of the objects in the cluster;

b) the configuration of the objects in the cluster;

c) sufficient information to re-establish distributed bindings that involve objects in the cluster.

NOTE – Engineering interface references are an essential part of the information required to establish bindings. A cluster checkpoint will contain all of the engineering references arising from a) and c) above.

12.3.2 Cluster deletion, deactivation and failure

Cluster deletion deletes all the objects in the cluster, the cluster manager and any objects solely supporting the cluster or its cluster manager (e.g. stubs and binders). Cluster deactivation is coordinated by the deactivation and reactivation function; it produces a checkpoint of the cluster, then deletes the cluster and its supporting structure. Cluster failure causes deletion of all objects in the cluster and, in some cases, leads to deleting the cluster’s supporting structure.

12.3.3 Cluster reactivation and recovery

A deactivated cluster can be reactivated from one of its checkpoints. Cluster reactivation is necessarily a capsule management function because the cluster manager is deleted as part of cluster deactivation. A cluster can be recovered from one of its checkpoints. If the associated cluster manager has not been deleted, it can initiate recovery; otherwise recovery is a capsule management function and includes creation of a new cluster manager.

12.3.4 Cluster migration

Cluster migration consists of the cloning of a source cluster in a target capsule followed by deletion of the source cluster. It is coordinated by the migration function and is parameterized by a capsule management interface for the target capsule for the migrated cluster.

12.4 Capsule management function

The capsule management function instantiates clusters (including recovery and reactivation), checkpoints all the clusters in a capsule, deactivates all the clusters in a capsule and deletes capsules. It is provided by each capsule manager at a capsule management interface comprising one or more of the following functions with respect to the managed capsule:

- Instantiation (within the capsule) of a cluster template.

  NOTE 1 – This includes reactivation and recovery.

- Deactivation of the capsule by deactivating all the clusters within it (using the cluster management function).
- Checkpointing the capsule by checkpointing all the clusters in the capsule (using the cluster management function).
- Deleting the capsule, by deleting all the clusters within it, followed by deletion of the capsule manager for the capsule.

NOTE 2 – Thus different capsule management interfaces can have different types, depending upon which functions they support.

The behaviour of a capsule manager is constrained by the management policy for its capsule.

Within the architecture defined by this Reference Model, the capsule management function is used by the deactivation and reactivation function, the checkpoint and recovery function and the migration function.

12.4.1 Cluster template instantiation

Cluster template instantiation is parameterized by a cluster template and consists of the following steps:

- instantiating a cluster from a cluster template;
- introducing a cluster manager for the new cluster;
- producing an identifier for a cluster management interface in the new cluster manager;
- binding the new cluster to other objects according to the engineering language rules and the binding information in the cluster template.

NOTE – Cluster reactivation and cluster recovery are special cases of cluster instantiation, in which case the cluster template is a cluster checkpoint.

A cluster template can contain information specific to a domain. If the template is to be instantiated in another domain, then this information must be transformed. In particular, engineering interface references contained in the cluster must be transformed if the cluster is being instantiated in a different engineering interface reference management domain.

12.4.2 Capsule deletion

Capsule deletion entails deletion of the capsule manager and can lead to deletion of stubs, binders, protocol objects or interceptors that supported objects in the capsule or its manager.

13 Coordination functions

13.1 Event notification function

The event notification function records and makes available event histories.

13.1.1 Concepts

13.1.1.1 Event history: An object representing significant actions.

13.1.2 Rules

Event producers interact with the event notification function to create event histories. The event notification function notifies event consumer objects of the availability of event histories.

The event notification function supports one or more event history types and has an event notification policy which determines the behaviour of the function, in particular:

- which objects can create event histories;
- which objects are notified of the creation of a new event history;
- the means by which such notifications occur;
- persistence and stability requirements for event histories;
- ordering relationships between interactions with event producer and event consumer objects.

An event consumer interacts with the event notification function to register for notification of new event histories. Depending upon the event notification policy, the interaction can:

- establish bindings to currently available event histories;
- enable communication about event histories created subsequent to the interaction.

NOTE – Event histories with stringent persistence and stability requirements can be supported using the transaction and replication functions. Ordering and multi-casting notifications can be supported using the group function.

13.2 Checkpoint and recovery function

The checkpoint and recovery function coordinates the checkpointing and recovery of failed clusters.

The checkpoint and recovery function embodies policies governing:

- when clusters should be checkpointed;
- when clusters should be recovered;
- where clusters should be recovered;
- where checkpoints should be stored;
- which checkpoint is recovered.

The checkpointing and recovery of clusters is subject to any security policy associated with those clusters, in particular, where the checkpoint is stored and where the checkpoint is recovered.

Within the architecture defined by this Reference Model, the checkpoint and recovery function uses the cluster management function and the capsule management function.

13.2.1 Checkpointing

Checkpointing is the responsibility of the object management function and the cluster management function. The checkpointing of a cluster is coordinated by its cluster manager: first the cluster manager uses the object management function to obtain a checkpoint of each object in the cluster; from these object checkpoints the cluster manager constructs a cluster checkpoint which it then makes persistent, using the storage function.

Depending upon the checkpointing policy, checkpointing a cluster can lead to checkpointing of other clusters which participate in common activities with the checkpointed cluster, subject to the following consistency rules:

- The initial cluster must be in a consistent state before it is checkpointed.
- There must be consistency between all of the cluster checkpoints of the various clusters being jointly checkpointed (e.g. all of the checkpoints reflect the same set of interactions having occurred between the clusters).
- When a cluster is checkpointed, all other clusters having checkpoint constraints with the cluster must be checkpointed. This rule must be applied recursively to obtain a closed set of clusters which can be checkpointed consistently.

13.2.2 Recovery

A cluster can be recovered either:

- in the capsule from which it was previously checkpointed; or
- in another capsule (e.g. when the capsule in which the checkpoint occurred has subsequently failed).

Recovery of a cluster is the responsibility of the cluster manager for that cluster, or in the absence of that cluster manager, a capsule manager. The checkpoint and recovery function interacts with the cluster manager or the capsule manager as appropriate to instantiate the cluster checkpoint. Before recovering a cluster, the checkpoint and recovery function must ensure that the cluster has been removed (e.g. as a consequence of failure). The objects bound to recovered clusters must be able to detect that the cluster has recovered from a checkpoint (e.g. so that they may re-do interactions that occurred after the checkpoint was taken).

The recovery of one cluster can lead to the recovery of other clusters, for example those forming a joint checkpoint with the cluster that is recovered.

13.3 Deactivation and reactivation function

The deactivation and reactivation function coordinates the deactivation and reactivation of clusters. It embodies policies governing:

- when clusters should be deactivated;
- where the checkpoint associated with a deactivation should be stored;
- when clusters should be reactivated;
- which checkpoint should be reactivated (e.g. the most recent);
- where clusters should be reactivated.

The deactivation and reactivation of clusters is subject to any security policy associated with those clusters, in particular within the architecture defined by this Reference Model, the deactivation and reactivation function uses the object management function, the cluster management function and the capsule management function. The deactivation and reactivation function is used by the migration function.

13.3.1 Deactivation

Deactivation of a cluster is a cluster management function and comprises the following steps:

- the cluster manager for the cluster involved interacts with each object in its cluster to obtain a checkpoint that can be used to make a cluster checkpoint;
- the cluster manager makes the cluster checkpoint persistent, using the storage function;
- the cluster manager deletes the cluster (and can itself be deleted).

13.3.2 Reactivation

The deactivation and reactivation function reactivates a cluster by using the capsule management function to instantiate a checkpoint of the cluster in the target capsule (and includes creation of a cluster manager for the cluster). The target capsule can either be the capsule from which it was previously deactivated or another capsule (e.g. to balance infrastructure load across several nodes).

13.4 Group function

The group function provides the necessary mechanisms to coordinate the interactions of objects in multi-party binding.

13.4.1 Concepts

13.4.1.1 Interaction group: A subset of the objects participating in a binding managed by the group function.

13.4.2 Rules

For each set of objects that is bound together in an interaction group, the group function manages:

- **interaction**: deciding which members of the group participate in which interactions, according to an interaction policy;
- **collation**: derivation of a consistent view of interactions (including failed interactions), according to a collation policy;
- **ordering**: ensuring that interactions between group members are correctly ordered with respect to an ordering policy;
- **membership**: dealing with member failure and recovery, and addition and removal of members according to a membership policy.

NOTE – The behaviour of the binding object linking members of the group determines how interaction is to be effected.

13.5 Replication function

The replication function is a special case of the group function in which the members of a group are behaviourally compatible (e.g. because they are replicas from the same object template). The replication function ensures the group appears to other objects as if it were a single object by ensuring that all members participate in all interactions and that all members participate in all interactions in the same order.

The membership policy for a replica group can allow for the number of members in a replica group to be increased or decreased. Increasing the size of a replica group achieves the same effect as if a member of the group had been cloned and then added to the group in a single atomic action.

For the replication function to be applied to a cluster, the objects comprising the cluster are replicated and configured into a set of identical clusters. The corresponding objects in each such replicated cluster form replica groups. Thus, a replicated cluster is a coordinated set of replica groups.

The replication function is used by the migration function.

13.6 Migration function

The migration function coordinates the migration of a cluster from one capsule to another. It uses the cluster management function and the capsule management function and embodies policies governing when clusters should be migrated and where they can be located.

Two possible means of migration are:

– replication; or

– deactivation in one capsule followed by reactivation in another.

13.6.1 Replication

Migration of a cluster by use of the replication function comprises the following sequence of actions:

– the old cluster is treated as a cluster replica group of size one;

– a copy of the original cluster is created in the destination capsule, together with a cluster manager;

– the objects in both the two clusters are formed into replica groups (of size two);

– the objects in the old cluster are removed from the object groups (leaving groups of size one);

– the old cluster (and its manager) is deleted.

13.6.2 Deactivation and reactivation

Migration of a cluster by deactivation and reactivation is coordinated by the cluster’s manager, and comprises deactivating the cluster at its old location, followed by reactivating the cluster at its new location.

13.7 Transaction function

13.7.1 Concepts

13.7.1.1 Transaction: An activity which leads to a set of object state changes consistent with a dynamic schema (and its constraining invariant schema).

13.7.1.2 Action of interest: An action in a transaction which leads to a state change of significance to the transaction.

13.7.1.3 Visibility: The degree to which a transaction can access object state concurrently with other transactions.

13.7.1.4 Recoverability: The degree to which object state changes resulting from failed transactions are cancelled.

13.7.1.5 Permanence: The degree to which failures can affect object state changes due to completed transactions.

13.7.2 Rules

The transaction function coordinates and controls a set of transactions to achieve a specified level of visibility, recoverability and permanence.

The transaction function:

– interacts with objects to monitor the occurrence of actions of interest, cancellation of the effects of actions of interest and the causality of actions of interest;

– decides if actions of interest are in conflict;

– interacts with objects to schedule the occurrence of actions of interest to prevent conflicts;

– interacts with objects to cancel the effects of actions of interest that have occurred, in order to resolve conflicts.

It is subject to policies determining:

- which actions are actions of interest;
- which actions of interest conflict;
- which actions of interest are to be cancelled to resolve conflicts.

13.8 ACID transaction function

The ACID transaction function is a special case of the general transaction function in which:

- visibility is specified as isolation of transactions from one another;
- recoverability is specified as the requirement that transactions be atomic;
- permanence is specified as the requirement that state changes arising from transactions be durable (i.e. stable); and
- consistency is achieved by a correct execution of transactions with the atomicity, isolation and durability properties in conformance with the associated dynamic and invariant schemata.

The actions of interest for the ACID transaction function are:

- start transaction;
- commit transaction;
- abort transaction;
- access state;
- modify state;
- cancellation of any of these.

Transaction policies are expressed as serializability rules for transactions.

13.9 Engineering interface reference tracking function

The engineering interface reference tracking function monitors the transfer of engineering interface references between engineering objects in different clusters to determine when the infrastructure associated with engineering interfaces is no longer required (i.e. when no object in another cluster is in a position to bind to the referenced interface).

The engineering interface reference tracking function maintains, within its scope:

- information on the possession of engineering interface references;
- information on the existence of interfaces.

The engineering interface reference tracking function:

- is notified by stubs when an engineering interface reference is passed between clusters;
- is notified by a cluster manager when all copies of an engineering interface reference are deleted in its cluster;
- detects when no copies of an engineering interface reference are held outside the cluster supporting the engineering interface referred to and notifies the corresponding cluster manager;
- notifies the managers of capsules in which there are holders of engineering interface references to an interface which has failed, or is deleted.

NOTE – The notification of engineering interface reference tracking events can be achieved using the event notification function.

The engineering interface reference management function is constrained by the engineering interface reference management policies of the engineering interface reference management domains to which it applies.

14 Repository functions

14.1 Storage function

The storage function stores data.

14.1.1 Concepts

14.1.1.1 Data repository: An object providing the storage function.

14.1.1.2 Container interface: An interface of a data repository allowing access to data.

14.1.2 Rules

A data repository stores sets of data. Each set of data is associated with a container interface created when the data are stored. A container interface provides functions to:

- obtain a copy of the data stored by the interface;
- modify the data associated with the interface;
- delete the container interface and its associated data.

NOTE – Objects embody both actions and state (i.e. process and data); they are inherently persistent. The checkpoint and recovery function or replication function can be used in an infrastructure of the storage function to provide stability. The functions can use other instances of the storage function as a repository for cluster checkpoints.

14.2 Information organization function

The information organization function manages a repository of information described by an information schema and includes some or all of the following elements:

- modifying and updating the information schema;
- querying the repository, using a query language;
- modifying and updating the repository.

The form and nature of queries and responses to queries depend upon the query language. The information organization function does not permit modifications or updates to the information repository that are inconsistent with its schema.

The information organization function can be modelled as a repository of objects (with computational interfaces) standing in correspondence to entities and relationships in an ODP system. These objects support operations to:

- define attributes, properties and relationships for objects in the repository;
- add and delete objects in the repository;
- assert and delete attributes, properties and relations for selected objects in the repository;
- select objects which satisfy a predicate (i.e. a type) specified in terms of attributes, properties and relations.

NOTES

1 The information organization function might derive additional relationships to those which are instantiated directly.

2 The information organization function can be used to maintain the relationship between a cluster and its checkpoint (i.e. a storage interface at which the checkpoint can be created or accessed).

3 In order to allow recovery in response to interactions with a failed cluster, the information organization function can be used to maintain a relationship between a cluster and the interface at which recovery of the cluster can be requested.

4 In order to allow reactivation in response to interactions with a deactivated cluster, the information organization function can be used to maintain a relationship between a deactivated cluster and the interface at which reactivation of the cluster can be requested.

5 The information needed by a relocator to validate or update an interface reference can be maintained by the information organization function.

14.3 Relocation function

The relocation function manages a repository of locations for interfaces, including locations of management functions for the cluster supporting those interfaces.

14.3.1 Concepts

14.3.1.1 Relocator: An object providing the relocation function.

14.3.2 Rules

A relocator has a directory of locations for interfaces that have had their locations changed as a consequence of either communications domain management (e.g. changing a node's network address) or cluster management activities such as deactivation, migration, replication or recovery of a cluster checkpoint.

An interface can be associated with a relocator; relocators can be specific to a single interface or common to several interfaces. When the scope of a relocator spans more than one engineering interface reference management domain, the means by which the relocator is accessed must make clear the interface reference management domain that applies.

When a relocator is associated with an interface, activities which change the location of interfaces must inform that relocator of the new location, in particular a cluster manager must notify the relocator for each interface of each object in the cluster when the cluster is relocated. Only interfaces of objects which have had their locations changed need be recorded. When an engineering interface reference is to remain valid even though the object supporting it is in a deactivated cluster or failed but previously checkpointed cluster, the relocator must embody a policy for use of the coordination functions to restore the cluster, by reactivation or recovery as appropriate, when another object attempts to validate the reference.

A relocator supports interactions that:

– record a change in the location of an interface identified by an engineering interface reference;

– validate the location of an interface identified by an engineering interface reference (including, if needed, restoration of the cluster containing the object supporting the interface);

– set policy for interacting with coordination functions (e.g. the deactivation and reactivation function) when validating the location of an interface identified by an engineering interface reference.

NOTE – An object performing engineering interface reference validation can retain the results of the validation to optimize future use of the reference.

14.4 Type repository function

The type repository function manages a repository of type specifications and type relationships. It has an interface for each type specification it stores.

14.4.1 Rules

The type repository function can be informed of type relationships in addition to those it can derive from comparison of type specifications. A type repository will not permit inconsistent relationships to be established.

Type specifications are immutable.

The type repository function includes creating types and their associated type interfaces.

A type repository interface for a specific type provides functions to:

– query the type's specification;

– assert relationships between the type and other types;

– query relationships between the type and other types.

NOTE – Subtyping relationships for computational signature types are defined by the signature subtyping rules in 7.2.4. There is no requirement for a type repository to enable signature subtypes relationships to be computed. Where signature types are included in a type repository, the repository is not permitted to impose additional signature typing rules or assertions that contradict the provisions of 7.2.4.

14.5 Trading function

The trading function mediates advertisement and discovery of interfaces.

14.5.1 Concepts

14.5.1.1 Service offer: Information about an interface including both an identifier for the interface and its computational interface signature type.

NOTES

1 The identifier enables binding to the interface.

2 The computational signature enables a trader to ensure service import selects service offers which will interact in the way expected by the importing object.

3 Additional information in the service offer can be used to provide greater discrimination than that embodied in interface signatures.

14.5.1.2 Service export: An interaction with the trading function in which a service offer is advertised, by adding the service offer to an identified set of service offers.

14.5.1.3 Service import: An interaction with the trading function which searches an identified set of service offers to discover interfaces at which a service satisfying a specified type is available.

14.5.2 Rules

The trading function provides service import and service export and its behaviour is governed by a trading policy which states rules on how the sets identified in service export are related to the sets in service import. On service import, a trading function must select only offers that satisfy the policy of the trading function itself, the policy of the exporter of the service offer and the policy of the importer of the service offer. Service import involves computational interface signature sub/supertype checking. It can, in addition, involve further levels of checks, including checks on behavioural capability and environmental constraints.

15 Security functions

15.1 Concepts

The following concepts are common to all of the security functions.

15.1.1 Security policy: A set of rules that constrains one or more sets of activities of one or more sets of objects.

15.1.2 Security authority: The administrator responsible for the implementation of a security policy.

15.1.3 Security domain: A domain in which the members are obliged to follow a security policy established and administered by a security authority.

NOTE – The security authority is the controlling object for the security domain.

15.1.4 Security interaction policy: Those aspects of the security policies of different security domains that are necessary in order for interactions to take place between those domains.

15.2 Access control function

The access control function prevents unauthorized interactions with an object. It includes both an **access control decision function** and an **access control enforcement function**. Within the context of access control, objects fulfil the roles of either **target** or **initiator**. The function requires **access control information** about the target, the initiator and the interaction.

The initiator requests an interaction with the target from the access control function. The action control decision function decides whether access is permitted or denied on the basis of the access control information and the decision is enforced by the access control enforcement function.

NOTE – The access control decision function and the access control enforcement function can be provided by the object which has the role of target, or by other objects.

15.3 Security audit function

The security audit function provides monitoring and collection of information about security-related actions, and subsequent analysis of the information to review security policies, controls and procedures.

The security audit function includes each of the following elements:

- **alarm collector function**;
- **alarm examiner function**;
- **audit trail analyser function**;
- **audit trail archiver function**;
- **audit recorder function**;
- **audit trail examiner function**;
- **audit trail collector function**.

15.4 Authentication function

The authentication function provides assurance of the claimed identity of an object. In the context of authentication, objects fulfil one or more of the following roles:

- **principal**;
- **claimant**;
- **trusted third party**.

Authentication requires use of **exchange authentication information**.

NOTES

1 Any identifiable object in an ODP system can be the principal for authentication, including both objects that model people and those that model computer systems.

2 The object initiating an authentication is not necessarily the claimant.

There are two forms of authentication:

- **peer entity authentication**, providing corroboration of the identity of a principal within the context of a communication relationship;
- **data origin authentication**, providing corroboration of the identity of the principal responsible for a specific data unit.

NOTE – Authentication mechanisms are categorized in ITU-T Rec. X.811 | ISO/IEC 10181-2.

In an authentication involving two objects, either or both objects can have the role of claimant. Where both objects have the role of claimant the style of authentication is known as **mutual authentication**. Exchange authentication information is passed from the initiating object to the responding object and further exchange authentication information may then be passed in the reverse direction. Additional exchanges may also take place: different authentication mechanisms require different numbers of exchanges. Peer entity authentication always involves interaction with the claimant. Data origin authentication need not involve interaction with the claimant.

A claimant supports operations to acquire information needed for an instance of authentication and to generate exchange authentication information. A verifier supports operations to acquire information needed for an instance of authentication, and to verify received exchange authentication information and/or to generate it. Information may be exchanged with an authentication server and either the claimant or the verifier (or both) either prior to or during authentication exchanges.

The authentication function may use the key management function.

15.5 Integrity function

The integrity function detects and/or prevents the unauthorized creation, alteration or deletion of data.

The integrity function includes the all following functions:

- **shield**;
- **validate**;
- **unshield**.

In the context of integrity, objects fulfil one or more of the following roles:

- **integrity-protected data originator**;
- **integrity-protected data recipient**.

Integrity-protected data is passed from originator to recipient. An integrity-protected data originator supports an interface providing the shield function. An integrity-protected data recipient supports an interface providing the validate or unshield functions.

The integrity function may use the key management function.

15.6 Confidentiality function

The confidentiality function prevents the unauthorized disclosure of information.

The confidentiality function includes the functions **hide** and **reveal**.

In the context of confidentiality, objects fulfil either or both of the following roles:

- **confidentiality-protected information originator**;
- **confidentiality-protected information recipient**.

Confidentiality-protected information is passed from originator to recipient. A confidentiality-protected information originator supports an interface providing the hide function. A confidentiality-protected information recipient supports an interface providing the reveal function.

The confidentiality function may use the key management function.

15.7 Non-repudiation function

The non-repudiation function prevents the denial by one object involved in an interaction of having participated in all or part of the interaction.

In the context of non-repudiation, objects fulfil one or more of the following roles:

- **(non-repudiable data) originator**;
- **(non-repudiable data) recipient**;
- **evidence generator**;
- **evidence user**;
- **evidence verifier**;
- **non-repudiation service requester**;
- **notary**;
- **adjudicator**.

The non-repudiation function makes use of non-repudiation evidence. In non-repudiation with proof of origin, the originator has the role of non-repudiation evidence generator for the origination interaction and includes this evidence in an acknowledgement of participation in the interaction. The recipient has the role of evidence user and uses the services of an evidence verifier (which may be itself) to gain confidence in the adequacy of the evidence. In non-repudiation with proof of delivery, the recipient has the role of non-repudiation evidence generator for the delivery interaction and includes this evidence in an acknowledgement of participation in the interaction. The originator has the role of evidence user and uses the services of an evidence verifier (which may be itself) to gain confidence in the adequacy of the evidence.

A notary provides functions required by the originator and/or recipient. These may include notarization, time stamping, monitoring, certification, certificate generation, signature generation, signature verification and delivery as identified in ITU-T Rec. X.813 | ISO/IEC 10181-4.

In the event of a dispute, an adjudicator collects information and evidence from the disputing parties (and optionally from notaries) and applies a resolution function as described in ITU-T Rec. X.813 | ISO/IEC 10181-4.

The non-repudiation function may use the key management function.

15.8 Key management function

The key management function provides facilities for the management of cryptographic keys and includes all of the following elements:

- **key generation**;
- **key registration**;
- **key certification**;
- **key deregistration**;
- **key distribution**;
- **key storage**;
- **key archiving**;
- **key deletion**.

Within the context of key management, objects can have one or more of the following roles:

- **certification authority**;
- **key distribution centre**;
- **key translation centre**.

A certification authority is a trusted third party which creates and assigns certificates as defined in ISO/IEC 11770-1. A key distribution centre provides means to establish key management information securely between objects authorized to obtain it. A key translation centre is a specific form of key distribution centre which establishes key management information between objects in different security domains.

16 ODP distribution transparency

Distribution transparency is selective in ODP systems. This Reference Model describes how to achieve the following distribution transparencies:

- access transparency;
- failure transparency;
- location transparency;
- migration transparency;
- persistence transparency;
- relocation transparency;
- replication transparency;
- transaction transparency.

ODP standards can define both:

- refinements of the descriptions in this Reference Model; and
- additional distribution transparencies required for those standards.

Transparencies are defined as constraints on the mapping from a computational specification containing a transparency schema to a specification that uses specific ODP functions and engineering structures to provide the required form of masking.

The behaviour of stubs, binders, protocol objects and interceptors in channels is determined by the combination of distribution transparencies (e.g. access transparency, relocation transparency) that apply to the channel.

In some implementable standards requiring more than one distribution transparency, composition rules for objects supporting individual transparencies might be specified. In other implementable standards requiring more than one distribution transparency, a single object might provide the combined distribution transparency.

The transparency descriptions in this Reference Model support at least the following combinations of distribution transparencies:

a) access and location transparency;
b) (a) and relocation transparency;
c) (b) and migration transparency;
d) (b) and resource transparency;
e) (b) and failure transparency;
f) (a) and transaction transparency;
g) (b) and transaction transparency.

16.1 Access transparency

Access transparency masks differences in data representation and invocation mechanisms to enable interworking between objects.

Access transparency is provided by the selection of a suitable channel structure (e.g. in which stubs provide appropriate conversions, such as marshalling into a canonical data representation).

16.2 Failure transparency

Failure transparency masks, from an object, the failure and possible recovery of other objects or itself, to enable fault tolerance.

16.2.1 Concepts

16.2.1.1 Stability schema: A specification of failure modes which an object will not exhibit.

16.2.2 Rules

The failure transparency refinement can satisfy a stability schema by one of the following methods:

– locating the object at a node with an infrastructure that excludes the specified failures;
– using the checkpoint and recovery function to make the object stable;
– using the replication function to make the object stable.

16.2.2.1 Replication

In the case of replication, the failure transparency refinement includes each of the following steps:

– defining an object management interface supporting object checkpointing and deletion for the computational object;
– introducing a replication function;
– setting a replication policy for the clusters containing the object;
– associating a relocator which supports replication with each of the object’s interfaces.

In the architecture defined by this Reference Model, failure transparency based on replication of a cluster requires relocation transparency.

16.2.2.2 Checkpoint and recovery

In the case of checkpoint and recovery, the failure transparency refinement includes each of the following steps:

– defining an object management interface supporting object checkpointing and deletion for the computational object;
– introducing a checkpoint and recovery function;

- setting a checkpoint and recovery policy for the clusters containing the object;
- associating a relocator which supports recovery with each of the object's interfaces.

Failure transparency based on checkpointing and recovery of a cluster requires relocation transparency.

16.3 Location transparency

Location transparency masks the use of information about location in space when identifying and binding to interfaces. This allows objects to access interfaces without using location information.

16.4 Migration transparency

Migration transparency masks, from an object, the ability of a system to change the location of that object.

16.4.1 Concepts

16.4.1.1 Mobility schema: A specification putting constraints on the mobility of an object.

A mobility schema includes:

- latency constraints on interactions with the object;
- performance constraints on the object's threads;
- security constraints on the location of the object.

16.4.2 Rules

Migration transparency is provided by use of the migration function to coordinate the location of an object to satisfy a mobility schema. The migration transparency refinement includes the following steps:

- defining an object management interface supporting object checkpointing and deletion for the computational object;
- introducing a migration function;
- setting a migration policy for the clusters containing the object;
- associating a relocator with each of the object's interfaces.

In the architecture defined by this Reference Model, migration transparency requires relocation transparency for all channels bound to the cluster.

16.5 Persistence transparency

Persistence transparency masks, from an object, the deactivation and reactivation of other objects (or itself).

16.5.1 Concepts

16.5.1.1 Persistence schema: A specification of constraints on the use of specific processing, storage and communication functions.

16.5.2 Rules

Persistence transparency is provided by use of the reactivation and deactivation function to coordinate the deactivation and reactivation of clusters to satisfy a persistence schema. The persistence transparency refinement includes the following steps:

- defining an object management interface supporting object checkpointing and deletion for the computational object;
- introducing a deactivation and reactivation function;
- setting a deactivation and reactivation policy for the clusters containing the object;
- associating a relocator which supports reactivation with each of the object's interfaces.

In the architecture defined by this Reference Model, persistence transparency requires relocation transparency for all channels bound to the cluster.

16.6 Relocation transparency

Relocation transparency masks relocation of an interface from other interfaces bound to it.

Relocation transparency requires:

- A relocator to be associated with each interface in a cluster.
- Propagation of information about changes in object location to relocators (e.g. by cluster managers).
- Binders to exchange additional data in interactions through a channel to confirm the validity of the binding supported by the channel. This data is derived from the location in space and time associated with engineering interface references of the interfaces bound to the channel.

In the event of a binder detecting that a channel has been invalidated by relocation of an object (e.g. by observing communication failure), the binder must validate the engineering interface references for the interfaces to which the channel was bound, and if necessary, re-establishing the channel. The validation is performed by the relocation function.

NOTE – If the channel is invalidated by deactivation of an object or by failure of a previously checkpointed object, the relocator will embody a procedure for restoring the cluster containing the object as part of the validation.

16.7 Replication transparency

Replication transparency masks the use of a group of mutually behaviourally compatible objects to support an interface.

16.7.1 Concepts

16.7.1.1 Replication schema: A specification of constraints on the replication of an object including both constraints on the availability of the object and constraints on the performance of the object.

16.7.2 Rules

Replication transparency is provided by the use of the replication function to coordinate the replication of an object to satisfy a replication schema. The replication transparency refinement includes the following steps:

- defining an object management interface supporting object checkpointing and deletion for the computational object;
- introducing a replication function;
- setting replication policy for the clusters containing the object;
- associating a relocator with each of the object’s interfaces.

In the architecture defined by this Reference Model, replication transparency requires relocation transparency for the cluster.

16.8 Transaction transparency

Transaction transparency masks coordination of activities among a configuration of objects, to achieve consistency.

16.8.1 Concept

16.8.1.1 Transaction schema: A dynamic schema and an invariant schema defining transactions and their dependencies.

16.8.2 Rules

Transaction transparency is provided by use of the transaction function to coordinate the behaviour of an object to satisfy a transaction schema. The transaction transparency refinement includes the following steps:

- deriving transaction function policies from the transactional schema;
- adding checkpoint and recovery operations for the state of the object;
- replacing bindings among the objects by a transaction function;
- extending the interfaces of the computational objects.

The extensions to the object’s interfaces include functions for both notifying the occurrence of actions of interest and recovering object state after cancellation.

Annex A

Formal computational supertype/subtype rules

(This annex forms an integral part of this Recommendation | International Standard)

This annex defines formal subtyping rules for computational interface signatures. The types of computational interface signatures can be higher order as implied by 7.2.2.4 on parameter rules. This annex only formalizes a first-order subset of the subtyping rules – formalizing higher order features of computational interface signature types is left for further study.

This annex defines a first order type system that consists of a simple type language together with type equality rules and signature subtyping rules. It also describes a sound and complete type checking algorithm for the type system. Signal signature interface types, operation interface signature types and stream interface signature types are defined using the type language. Since stream interface signature subtyping is only partially defined in 7.2.4.2, this annex only formalizes the subtyping rule that applies between corresponding flows.

A.1 Notations and conventions

The following notations are used:

- α, β, γ, etc., denote types;
- *t, s,* etc., denote identifiers for types (i.e. type variables) and ground types (i.e. type constants); the set of type variables (and type constants) is called T_{var};
- *a, b, c,* a_1, a_2, a_n, etc., denote identifiers or *labels* for elements of structures in the type language; the set of labels is called Λ;
- $\alpha[\beta/t]$ denotes the substitution of β for *t* in α;
- *Nil* denotes a predefined type constant.

A.2 Type system

The type system contains type constants, functions, cartesian products, records, tagged unions recursive definitions. The type language, *Type*, is given by the grammar in Figure A.1.

$$
\begin{array}{lll}
♋ & ::= & t \\
 & | & \bot \\
 & | & \top \\
 & | & \alpha \rightarrow \beta \\
 & | & \alpha_1 \times \ldots \times \alpha_n \\
 & | & \langle ♋_1 : \alpha_1, \ldots, ♋_n : \alpha_n \rangle \\
 & | & [c_1 : \gamma_1, \ldots, c_n : \gamma_n] \\
 & | & \mu t.\alpha
\end{array}
$$

Figure A.1 – Abstract syntax for type declarations

The ground types are called $\top$ (top) and $\bot$ (bottom). They play the roles of greatest and least elements in the subtype relation, respectively. Functions are denoted thus: $\alpha \rightarrow \beta$. Cartesian products are denoted thus: $\alpha_1 \times \ldots \times \alpha_n$. Unions are denoted thus: $[c_1 : \gamma_1 \ldots, c_n : \gamma_n]$. Records are denoted thus: $\langle a_1 : \alpha_1, \ldots, a_n : \alpha_n \rangle$.

μ is a variable binding operator. Recursive types can be constructed by binding types to identifiers and referencing an identifier for one type in another.

Parentheses are used to determine precedence where necessary. In their absence, $\rightarrow$ associates to the right, and the scoping of μ extends to the right as far as possible.

The set of free variables occuring in α is denoted thus: $FV(\alpha)$.

A.2.1 Typing rules

This clause gives type equality rules and subtyping rules for the language given.

A type α is *contractive* in the type variable t, denoted $\alpha \downarrow t$, if either t does not occur free in α, or α can be rewritten via unfolding as a type of one of the following forms:

- $\alpha_1 \rightarrow \alpha_2$;
- $\langle \text{♋}_1 : \alpha_1, \ldots, \text{♋}_m : \alpha_m \rangle$;
- $[c_1 : \gamma_1, \ldots, c_n : \gamma_n]$;
- $\alpha_1 \times \ldots \times \alpha_n$.

The type equality rules are given in Figure A.2. Type equality is denoted by =.

(E.1)	$\alpha = \alpha$
(E.2)	$\alpha = \beta \Rightarrow \beta = \alpha$
(E.3)	$\alpha = \beta, \beta = \gamma \Rightarrow \alpha \Rightarrow \gamma$
(E.4)	$\alpha_1 = \alpha_2, \beta_1 = \beta_2 \Rightarrow \alpha_1 \rightarrow \beta_1 = \alpha_2 \rightarrow \beta_1$
(E.5)	$\alpha = \beta \Rightarrow \mu t.\alpha = \mu t.\beta$
(E.6)	$\forall i \, \varepsilon \, \{1, \ldots, n\}, \alpha_i = \beta_i \Rightarrow \alpha_1 \times \ldots \times \alpha_n = \beta_1 \times \ldots \times \beta_n$
(E.7)	$\forall i \, \varepsilon \, \{1, \ldots, n\}, \alpha_i = \beta_i \Rightarrow \langle \text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n \rangle = \langle \text{♋}_1 : \beta_1, \ldots, \text{♋}_n : \beta_n \rangle$
(E.8)	$\forall i \, \varepsilon \, \{1, \ldots, n\}, \alpha_i = \beta_i \Rightarrow [\text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n] = [\text{♋}_1 : \beta_1, \ldots, \text{♋}_n : \beta_n]$
(E.9)	$\mu t.t = \perp$
(E.10)	$\alpha[\mu t.\alpha/t] = \mu t.\alpha$
(E.11)	$\alpha[\beta/t] = \beta_1, \alpha[B_2/t] = \beta_2 \; \alpha \downarrow t \Rightarrow \beta_1 = \beta_2$

Figure A.2 – Type equality rules

The subtyping rules are given in the form of inference rules on judgments that resemble a Prolog program. Judgements are of the form: $\Gamma \vdash \alpha \leq \beta$, where Γ is a set of subtyping assumptions on type variables of the form: $\{t_1 \leq s_1, \ldots, t_n \leq s_n\}$. A typical rule may take the following form:

$$\Gamma \vdash \alpha_1 \leq \beta_1, \; \Gamma \vdash \alpha_2 \leq \beta_2 \; \Rightarrow \; \Gamma \vdash \alpha \leq \beta$$

Informally, this means that in order to determine whether $\Gamma \vdash \alpha \leq \beta$ holds, one must first try to determine whether $\Gamma \vdash \alpha_1 \leq \beta_1$ and $\Gamma \vdash \alpha_2 \leq \beta_2$. If these subgoals are reached, then one may conclude that α is a subtype of β.

The subtyping rules are given in Figure A.3. It can be said that α is a subtype of β if $\varnothing \vdash \alpha \leq ß$ can be derived using the subtyping rules and the equality rules.

(S.1)	$\alpha = \beta \Rightarrow \Gamma \vdash \alpha \leq \beta$
(S.2)	$\Gamma \vdash \alpha \leq \beta, \Gamma \vdash \beta \leq \gamma \Rightarrow \Gamma \vdash \alpha \leq \gamma$
(S.3)	$t \leq s \in \Gamma \Rightarrow \Gamma \vdash t \leq s$
(S.4)	$\Gamma \vdash \perp \leq \alpha$
(S.5)	$\Gamma \vdash \alpha \leq \top$
(S.6)	$\Gamma \vdash \alpha_2 \leq \alpha_1, \Gamma \vdash \beta_1 \leq \beta_2 \Rightarrow \Gamma \vdash \alpha_1 \rightarrow \beta_1 \leq \alpha_2 \rightarrow \beta_2 \leq$
(S.7)	$\forall \iota \, \varepsilon \, \{1, \ldots, n\}, \Gamma \vdash \alpha_i \leq \beta_i \Rightarrow \Gamma \vdash \langle \text{♋}_1 : \alpha_1, \ldots, \text{♋}_m : \alpha_m \rangle \leq \langle \text{♋}_1 : \beta_1, \ldots, \text{♋}_n : \beta_n \rangle$ *with $n \leq m$*
(S.8)	$\forall \iota \, \varepsilon \, \{1, \ldots, n\}, \Gamma \vdash \alpha_i \leq \beta_i \Rightarrow \Gamma \vdash [\text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n] \leq [\text{♋}_1 : \beta_1, \ldots, \text{♋}_n : \beta_n]$ *with $n \leq m$*
(S.9)	$\forall \iota \, \varepsilon \, \{1, \ldots, n\}, \Gamma \vdash \alpha_i \leq \beta_i \Rightarrow \Gamma \vdash \alpha_1 \times \ldots \times \alpha_n \leq \beta_1 \times \ldots \times \beta_n$
(S.10)	$\Gamma \cup \{t \leq s\} \vdash \alpha \leq \beta \Rightarrow \Gamma \vdash \mu t.\alpha \leq \mu s.\beta$ *with t only in α; s only in β, t, s not in Γ*

Figure A.3 – Subtyping rules

A.2.2 Type definitions

Elements of *Type* are defined by sets of mutually dependent equations, modelled as well-formed environments. An environment is a finite mapping between type variables and types belonging to *T*, where *T* is the non-recursive subset of *Type*. A well-formed environment Υ is an environment such that the free variables of a type α associated with a variable *t* in the domain of Υ all belong to the domain of Υ. Intuitively, each type variable in an environment represents a type. The associations between type variables and elements of *T* in an environment can be understood as mutually dependent defining equations for the corresponding types.

Formally, let Υ_{wf} be the set of well-formed environments and define $f : A \rightarrow_f B$ as a partial function from *A* to *B* with finite domain; $FV(\alpha)$ denotes the set of free variables occurring in α):

$$\Upsilon_{wf} =_{def} \{\Upsilon : T_{var} \rightarrow_{\mathrm{f}} Type \mid \forall\, t, t' \in dom\,(\Upsilon), t' \in FV\,(\Upsilon\,(t)) \Rightarrow t' \in dom\,(\Upsilon)\}$$

Let $\Upsilon = \{t \mapsto \alpha, t1 \mapsto \alpha_1, \ldots, t_q \mapsto \alpha_q\}$. $\Upsilon \backslash t$ denotes the following environment:

$$\Upsilon \backslash t =_{def} \{t_1 \mapsto \alpha_1, \ldots, t_q \mapsto \alpha_q\}$$

The type associated with a type variable *t* in the context of a well-formed environment Υ ($t \in dom\,(\Upsilon)$) is defined to be *Val (*t, Υ), where *Val* is the function on types and environments defined recursively in Figure A.4. Thus, any element of *Type* can be defined as *Val (t,* Υ), where Υ is a well-formed environment and $t \in dom\,(\Upsilon)$.

(IT.1)	$Val\,(\bot, \Upsilon) = \bot$
(IT.2)	$Val\,(\top, \Upsilon) = \top$
(IT.3)	$Val\,(Nil, \Upsilon) = Nil$
(IT.4)	$Val\,(\alpha \rightarrow \beta, \Upsilon) = Val\,(\alpha, \Upsilon) \rightarrow Val\,(\beta, \Upsilon)$
(IT.5)	$Val\,(\langle \text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n \rangle, \Upsilon) = \langle \text{♋}_1 : Val\,(\alpha_1, \Upsilon), \ldots, \text{♋}_n : Val\,(\alpha_n, \Upsilon)\rangle$
(IT.6)	$Val\,([\text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n], \Upsilon) = [\text{♋}_1 : Val\,(\alpha_1, \Upsilon), \ldots, \text{♋}_n : Val\,(\alpha_1, \Upsilon)]$
(IT.7)	$Val\,(\alpha_1 \times \ldots \times \alpha_n, \Upsilon) = Val\,(\alpha_1, \Upsilon) \times Val\,(\alpha_n, \Upsilon)$
(IT.8)	*if* t /,$\in$ *dom* (Υ) *then* $Val\,(t, \Upsilon) = t$
(IT.10)	*if* $t \in$ *dom* (Υ) *then* $Val\,(t, \Upsilon) = \mu t.Val\,(\Upsilon\,(t), \Upsilon \backslash t)$

Figure A.4 – Semantics of interface type definitions

A.2.3 An algorithm for type checking

This subclause defines an algorithm for type checking which is sound and complete with respect to the type equality and subtyping rules above. The algorithm involves two well-formed environments ε_1 and ε_2 such that $dom(\varepsilon_1) \cap dom(\varepsilon_2) = \emptyset$ (the types to be compared are associated with two variables, one in ε_1 and the other in ε_2). It is described as a set of inference rules involving $\varepsilon =_{def} \varepsilon_1 \cup \varepsilon_2$ and a set Σ of the form $\{t_1 \leq s_1, \ldots, t_n \leq s_n\}$ which records inclusion of variables discovered during execution of the algorithm. An inference rule corresponds to a logical implication of *judgements* of the form $\Sigma, \varepsilon \vdash \alpha \leq \beta$. A judgement intuitively captures the assertion $\alpha \leq \beta$ holds in the context of Σ and ε. Initial judgements $\{t_1 \leq s_1, \ldots, t_n \leq s_n\}$ must be such that $\{t_1, s_1, \ldots, t_n, s_n\} \cap dom\,(\varepsilon) = \emptyset$.

The inference rules are given in Figure A.5. In Figure A.5 $\alpha, \beta \in Type$, *t*, *s* denote arbitrary variables, *u* denotes variables not in $dom(\varepsilon)$.

Given an initial goal $\Sigma, \varepsilon \vdash \alpha \leq \beta$, the algorithm consists in applying the inference rules backwards, generating subgoals in cases (rec), (fun), (rcd), (pro) and (uni). A tree of goals built in this way is called an execution tree. An execution tree is always finite: if $t \leq s$ is an assumption that is added to Σ, then *t* and *s* are type variables in *dom* (ε); also, the rules (fun), (pro), (uni) and (rcd) reduce the size of the current goal by replacing it with subexpressions of the goal, and each application of (rec) enlarges Σ.

(assmp)	$t \le s \in \Sigma \Rightarrow \Sigma, \varepsilon \vdash t \le s$
(bot)	$\Sigma, \varepsilon \vdash \bot \le \beta$
(top)	$\Sigma, \varepsilon \vdash \text{♋} \le \top$
(var)	$\Sigma, \varepsilon \vdash u \le u$
(fun)	$\Sigma, \varepsilon \vdash \alpha_2 \le \alpha_1, \Sigma, \varepsilon \vdash \beta_1 \le \beta_2 \Rightarrow \Sigma, \varepsilon \vdash \alpha_1 \rightarrow \beta_1 \le \alpha_2 \rightarrow \beta_2 \le$
(rcd)	$\forall \iota \in \{1, \ldots, n\}, \Sigma, \varepsilon \vdash \alpha_i \le \beta_i \Rightarrow \Sigma, \varepsilon \vdash \langle \text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n \rangle \le \langle \text{♋}_1 : \beta_1, \ldots, \text{♋}_n : \beta_n \rangle$ *with* $n \le m$
(uni)	$\forall \iota \in \{1, \ldots, n\}, \Sigma, \varepsilon \vdash \alpha_i \le \beta_i \Rightarrow \Sigma, \varepsilon \vdash [\text{♋}_1 : \alpha_1, \ldots, \text{♋}_n : \alpha_n] \le \{\text{♋}_1 : \beta_1, \ldots, \text{♋}_m : \beta_m]$ *with* $n \le m$
(pro)	$\forall \iota \in \{1, \ldots, n\}, \Sigma, \varepsilon \vdash \alpha_i \le \beta_i \Rightarrow \Sigma, \varepsilon \vdash \alpha_1 \times \ldots \times \alpha_n \le \beta_1 \times \ldots \times \beta_n$
(rec)	$\Sigma \cup \{t \le s\}, \varepsilon \vdash \varepsilon(t) \le \varepsilon(s) \Rightarrow \Sigma, \varepsilon \vdash t \le s$

Figure A.5 – Type-checking inference rules

An execution tree *succeeds* if all the leaves correspond to an application of one of the rules (assmp), (bot), (top) or (var). It *fails* if at least one leaf is an unfulfilled goal (i.e. if no rule can be applied to it). If the execution tree corresponding to the goal $\varnothing\varepsilon \vdash \alpha \le \beta$ succeeds, this is noted $\vdash_A \alpha \le \beta$.

Given recursive types α and β, such that $\alpha = Val(t_1, \varepsilon_1)$ and $\beta = Val(t_2, \varepsilon_2)$ (ε_1 and ε_2 as above) a subtyping relation, $\le_A$, is induced by the algorithm by the following definition:

$$\alpha \le_A \beta \Leftrightarrow \vdash_A t_1 \le t_2$$

This new subtyping relation coincides with the previous one, i.e. the algorithm is sound and complete with respect to the type equality and type subtyping rules:

– given α, β in T, if $\alpha \le_A \beta$ then $\alpha \le_R \beta$;

– given α, β in T, if $\alpha \le_R \beta$ then $\alpha \le_A \beta$.

A.3 Signal interface signature types

Signal interface signature types are formalized by interpreting them in the *Type* language. The set of signal interface signature types is denotenoted $Type_{sig}$. Elements of $Type_{sig}$ are defined abstractly through the use of two functions: $intype : Type_{sig} \rightarrow Type$ and $outype : Type_{sig} \rightarrow Type$. In a given signal interface signature type, *intype* describes the set of initiating signals, and *outype* describes the set of responding signals.

Elements of *Type* associated with a signal interface signature type through the *intype* and *outype* functions are defined by well-formed environments with codomain the subset of *Type* defined by the grammar in Figure A.6, where labels a_i, $i \in \{1, \ldots, q\}$, are supposed to be distinct. In effect, Figure A.6 provides an abstract syntax for signal interface signatures. Labels a_i correspond to signal names. *Arg* productions correspond to signal parameters. *Sigsig* productions correspond to individual signal signatures. The functional form adopted for individual signal signature highlights the analogy with announcement signatures.

The subtyping relation on signal interface types, $\le_s$, is defined by:

$$\forall \iota_1, \iota_2 \in Type_{sig}, \iota_1 \le \iota_2 \equiv \iota_1.intype \le \iota_2.intype \wedge \iota_2.outype \le \iota_1.outype$$

α	::=	$\langle \text{♋}_i\ Sigsig, \ldots, \text{♋}_q : Sigsig \rangle$
Sigsig	::=	$Arg \rightarrow Nil$
Arg	::=	$Nil \mid t_1 \times \ldots \times t_p$

Figure A.6 – Abstract syntax for signal interface signature types

A.4 Operation interface signature types

Operational *server* interface signature types are formalized by interpreting them in the *Type* language (operation client interface signature types can be derived immediately by complementation). The set of operational server interface types is noted $Type_o$ (S). Elements of $Type_o$ (S) are defined abstractly through the use of the function *optype*: $Type_o$ $(S) \to$ *Type*.

Elements of *Typ*e associated with an operation interface signature type through the *optype* function are defined by well-formed environments with codomain the subset of *Type* defined by the grammar in Figure A.7, where labels a_i, $i \in \{1, \ldots, q\}$, are supposed to be distinct, and where labels c_i, $i \in \{1, \ldots, q\}$ are supposed to be distinct in the context of an *Opsig* production.

α	::=	$\langle \mathfrak{a}_i : Opsig, \ldots, \mathfrak{a}_q : Opsig \rangle$
Opsig	::=	$Arg \to Term \mid Arg \to Nil$
Term	::=	$[c_1 : Arg, \ldots, c_q : Arg]$
Arg	::=	$Nil \mid t_1 \times \ldots \times t_p$

Figure A.7 – Abstract syntax for operation interface signature types

In effect, Figure A.7 provides an abstract syntax for operation interface signatures. *Opsig* productions correspond to individual operation signatures. Specifically, *Opsig* productions of the form $Arg \to Term$ in Figure A.1 correspond to interrogations. *Opsig* productions of the form $Arg \to Nil$ correspond to announcements. *Arg* productions correspond to invocation parameters. *Term* productions correspond to terminations. *Nil* on the left hand side of an *Opsig* production means that the given invocation does not have any parameter. *Nil* on the right hand side of an *Opsig* production (i.e. in an announcement signature) means that no termination is expected. Labels a_i correspond to operation names. Labels c_1 correspond to termination names.

The subtyping relation on server operational interface types, $\leq_o$, is defined by:

$$\forall\, \iota_1, \iota_2 \in Type_o\,(S),\ \iota_1 \leq \iota_2 \equiv \iota_1.optype \leq \iota_2.optype$$

A.5 Stream interface types

Defining complete signature subtyping rules for stream interfaces is beyond the scope of this Reference Model (see 7.2.4.2). Note, however, that an individual flow signature type can be formalized by interpreting it in the *Typ*e language. Elements of *Type* associated with a flow signature can be defined by well-formed environments with codomain the subset of *Type* defined by the grammar in Figure A.8, where label a_i corresponds to the flow name.

The subtyping rule in 7.2.4.2 associated with corresponding flows (assuming they have the same causality) just corresponds to the subtype relation, ≤, in this case.

α	::=	$\langle \mathfrak{a}_i : Flowsig \rangle$
Flowsig	::=	$Arg \to Nil$
Arg	::=	$Nil \mid t$

Figure A.8 – Abstract syntax for stream interface signature types

A.6 Example

Consider the following server operation interface signature type definitions (i.e. the well-formed environment):

$$\Upsilon = \{t \mapsto \alpha, f_t \mapsto \beta\}$$

where

$$\alpha =_{def} \langle op : t \rightarrow [ok : Nil, nok : Nil], factory : Nil \rightarrow [ok : f_t] \rangle$$

$$\beta =_{def} \langle new : Nil \rightarrow [ok : t] \rangle$$

and where t, $f_t \in Tvar$; op, $factory$, new, ok and $nok \in \Lambda$ (*op*, *factory* and *new* are operation names; *ok* and *nok* are termination names).

Intuitively, the environment Υ corresponds to the definition of two types, t and f_t. f_t is equipped with only one operation *new*, that takes no argument and returns a reference to an instance of type t. One may imagine, for instance, that f_t is the type of a factory object that creates objects with an interface of type t on demand, i.e. on each invocation of operation new. t is equipped with two operations: *op* and *factory*. Operation *op* takes a reference to an instance of type t as argument: this is a first instance of a recursive definition. Operation *factory* takes no argument and returns a reference to an instance of type f_t. One may imagine for example that, for management purposes, each object with an interface of type t is able, on request (i.e. upon invocation of operation *factory*), to return a reference to the factory that created it. This is a second instance of a recursive definition, since the definition of f_t refers to t.

Applying the definition of *Val* given above, defining $\Upsilon_1 =_{def} \{f_t \mapsto \beta\}$, overloading the = sign and using type equivalence rule E.10 the following is obtained:

$$\begin{aligned}
Val\,(t, \Upsilon) &= \mu\, t.\, Val\,(\alpha, \{ f_t \mapsto \beta\} \\
&= \mu\, t.\, \langle op : Val(t, \Upsilon_1) \rightarrow [ok: Nil, nok : Nil] \\
&\quad factory: Nil \rightarrow [ok : Val\,(f_t, \Upsilon_1)]\rangle \\
&= \mu\, t.\, \langle op : t \rightarrow [ok: Nil, nok : Nil] \\
&\quad factory: Nil \rightarrow [ok : \mu\, f_t.Val\,(\beta, \emptyset)]\rangle \\
&= \mu\, t.\, \langle op : t \rightarrow [ok: Nil, nok : Nil] \\
&\quad factory: Nil \rightarrow [ok : \mu \langle f_t \langle new : Nil \rightarrow [ok : t]\rangle]\rangle \\
&= \mu\, t.\, \langle op : t \rightarrow [ok: Nil, nok : Nil] \\
&\quad factory: Nil \rightarrow [ok : \langle new : Nil \rightarrow [ok : \mathrm{t}]\rangle]\rangle
\end{aligned}$$