Information technology – Open distributed processing – Use of UML for ODP system specifications
<table>
<thead>
<tr>
<th>Category</th>
<th>X.1 – X.199</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUBLIC DATA NETWORKS</strong></td>
<td></td>
</tr>
<tr>
<td>Services and facilities</td>
<td>X.1 – X.19</td>
</tr>
<tr>
<td>Interfaces</td>
<td>X.20 – X.49</td>
</tr>
<tr>
<td>Transmission, signalling and switching</td>
<td>X.50 – X.89</td>
</tr>
<tr>
<td>Network aspects</td>
<td>X.90 – X.149</td>
</tr>
<tr>
<td>Maintenance</td>
<td>X.150 – X.179</td>
</tr>
<tr>
<td>Administrative arrangements</td>
<td>X.180 – X.199</td>
</tr>
<tr>
<td><strong>OPEN SYSTEMS INTERCONNECTION</strong></td>
<td></td>
</tr>
<tr>
<td>Model and notation</td>
<td>X.200 – X.209</td>
</tr>
<tr>
<td>Service definitions</td>
<td>X.210 – X.219</td>
</tr>
<tr>
<td>Connection-mode protocol specifications</td>
<td>X.220 – X.229</td>
</tr>
<tr>
<td>Connectionless-mode protocol specifications</td>
<td>X.230 – X.239</td>
</tr>
<tr>
<td>PICS proforms</td>
<td>X.240 – X.259</td>
</tr>
<tr>
<td>Protocol Identification</td>
<td>X.260 – X.269</td>
</tr>
<tr>
<td>Security Protocols</td>
<td>X.270 – X.279</td>
</tr>
<tr>
<td>Layer Managed Objects</td>
<td>X.280 – X.289</td>
</tr>
<tr>
<td>Conformance testing</td>
<td>X.290 – X.299</td>
</tr>
<tr>
<td><strong>INTERWORKING BETWEEN NETWORKS</strong></td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>X.300 – X.349</td>
</tr>
<tr>
<td>Satellite data transmission systems</td>
<td>X.350 – X.369</td>
</tr>
<tr>
<td>IP-based networks</td>
<td>X.370 – X.379</td>
</tr>
<tr>
<td><strong>MESSAGE HANDLING SYSTEMS</strong></td>
<td></td>
</tr>
<tr>
<td>X.400 – X.499</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECTORY</strong></td>
<td></td>
</tr>
<tr>
<td>X.500 – X.599</td>
<td></td>
</tr>
<tr>
<td><strong>OSI NETWORKING AND SYSTEM ASPECTS</strong></td>
<td></td>
</tr>
<tr>
<td>Networking</td>
<td>X.600 – X.629</td>
</tr>
<tr>
<td>Efficiency</td>
<td>X.630 – X.639</td>
</tr>
<tr>
<td>Quality of service</td>
<td>X.640 – X.649</td>
</tr>
<tr>
<td>Naming, Addressing and Registration</td>
<td>X.650 – X.679</td>
</tr>
<tr>
<td>Abstract Syntax Notation One (ASN.1)</td>
<td>X.680 – X.699</td>
</tr>
<tr>
<td><strong>OSI MANAGEMENT</strong></td>
<td></td>
</tr>
<tr>
<td>Systems management framework and architecture</td>
<td>X.700 – X.709</td>
</tr>
<tr>
<td>Management communication service and protocol</td>
<td>X.710 – X.719</td>
</tr>
<tr>
<td>Structure of management information</td>
<td>X.720 – X.729</td>
</tr>
<tr>
<td>Management functions and ODMA functions</td>
<td>X.730 – X.799</td>
</tr>
<tr>
<td><strong>SECURITY</strong></td>
<td></td>
</tr>
<tr>
<td>X.800 – X.849</td>
<td></td>
</tr>
<tr>
<td><strong>OSI APPLICATIONS</strong></td>
<td></td>
</tr>
<tr>
<td>Commitment, concurrency and recovery</td>
<td>X.850 – X.859</td>
</tr>
<tr>
<td>Transaction processing</td>
<td>X.860 – X.879</td>
</tr>
<tr>
<td>Remote operations</td>
<td>X.880 – X.889</td>
</tr>
<tr>
<td>Generic applications of ASN.1</td>
<td>X.890 – X.899</td>
</tr>
<tr>
<td><strong>OPEN DISTRIBUTED PROCESSING</strong></td>
<td>X.900 – X.999</td>
</tr>
<tr>
<td>INFORMATION AND NETWORK SECURITY</td>
<td>X.1000 – X.1099</td>
</tr>
<tr>
<td>SECURE APPLICATIONS AND SERVICES</td>
<td>X.1100 – X.1199</td>
</tr>
<tr>
<td>CYBERSPACE SECURITY</td>
<td>X.1200 – X.1299</td>
</tr>
<tr>
<td>SECURE APPLICATIONS AND SERVICES</td>
<td>X.1300 – X.1399</td>
</tr>
<tr>
<td>CYBERSECURITY INFORMATION EXCHANGE</td>
<td>X.1500 – X.1599</td>
</tr>
<tr>
<td>CLOUD COMPUTING SECURITY</td>
<td>X.1600 – X.1699</td>
</tr>
</tbody>
</table>

For further details, please refer to the list of ITU-T Recommendations.
Summary

Rec. ITU-T X.906 | ISO/IEC 19793 defines use of the Unified Modelling Language (UML 2.4.1 Superstructure Specification, ISO/IEC 19505-2, for expressing system specifications in terms of the viewpoint specifications defined by the reference model of open distributed processing (RM-ODP, Rec. ITU-T X.901 to X.904 | ISO/IEC 10746 Parts 1 to 4) and the Enterprise Language (Rec. ITU-T X.911 | ISO/IEC 15414). It covers:

a) the expression of a system specification in terms of RM-ODP viewpoint specifications using defined Unified Modeling Language (UML) concepts and extensions (e.g., structuring rules, technology mappings, etc.);

b) relationships between the resultant RM-ODP viewpoint specifications.

This Recommendation | International Standard refines and extends the definition of how open distributed processing (ODP) systems are specified by defining the use of the Unified Modelling Language for the expression of ODP system specifications.

History

<table>
<thead>
<tr>
<th>Edition</th>
<th>Recommendation</th>
<th>Approval</th>
<th>Study Group</th>
<th>Unique ID*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>ITU-T X.906</td>
<td>2007-11-13</td>
<td>17</td>
<td>11.1002/1000/9261</td>
</tr>
<tr>
<td>1.1</td>
<td>ITU-T X.906 (2007) Cor.1</td>
<td>2009-10-29</td>
<td>17</td>
<td>11.1002/1000/10244</td>
</tr>
<tr>
<td>2.0</td>
<td>ITU-T X.906</td>
<td>2014-10-07</td>
<td>17</td>
<td>11.1002/1000/12152</td>
</tr>
</tbody>
</table>

* To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.
FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

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.

NOTE

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

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

© ITU 2015

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.
13.2 UML profile .......................................................................................................................... 67
14 Conformance and compliance to this Recommendation | International Standard ................ 68
14.1 Conformance ....................................................................................................................... 68
14.2 Compliance ......................................................................................................................... 68
Annex A – An example of ODP specifications using UML......................................................... 69
A.1 The Templeman Library system ........................................................................................... 69
A.2 Enterprise specification in UML ......................................................................................... 70
A.3 Information specification in UML ....................................................................................... 83
A.4 Computational specification in UML ................................................................................ 91
A.5 Engineering specification in UML ...................................................................................... 96
A.6 Technology specification in UML ..................................................................................... 107
Annex B – An example of the representation of deontic concepts ........................................ 111
B.1 The scenario ....................................................................................................................... 111
B.2 Expressing the deontic constraints ................................................................................... 112
INDEX ......................................................................................................................................... 117
Introduction

The rapid growth of distributed processing has led to the adoption of the reference model of open distributed processing (RM-ODP), which provides a coordinating framework for the standardization of open distributed processing (ODP). It creates an architecture within which support of distribution, interworking and portability can be integrated. This architecture provides a framework for the specification of ODP systems.

The reference model of open distributed processing 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. It does not recommend any notation.

The Unified Modeling Language™ (UML®) was developed by the Object Management Group™ (OMG™). It provides a notation for modelling in support of information system design and is widely used throughout the IT industry as the language and notation of choice.

This Recommendation | International Standard refines and extends the definition of how ODP systems are specified by defining the use of the unified modelling language for the expression of ODP system specifications.

0.1 RM-ODP

The RM-ODP consists of:
- Part 1 [Rec. ITU-T X.901 | ISO/IEC 10746-1]: Overview, which 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 the RM-ODP 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 Rec. ITU-T X.903 | ISO/IEC 10746-3. This part is informative.
- Part 2 [Rec. ITU-T X.902 | ISO/IEC 10746-2]: Foundations, which 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 Rec. ITU-T X.903 | ISO/IEC 10746-3 and to establish requirements for new specification techniques. This part is normative.
- Part 3 [Rec. ITU-T X.903 | ISO/IEC 10746-3]: Architecture, which contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards shall conform. It uses the descriptive techniques from Rec. ITU-T X.902 | ISO/IEC 10746-2. This part is normative.
- Part 4 [Rec. ITU-T X.904 | ISO/IEC 10746-4]: Architectural semantics, which contains a formalization of the ODP modelling concepts defined in Rec. ITU-T X.902 | ISO/IEC 10746-2 clauses 8 and 9. The formalization is achieved by interpreting each concept in terms of the constructs of one or more of the different standardized formal description techniques. This part is normative.

In the same series as the RM-ODP are a number of other standards and recommendations, and, of these, the chief that concerns this Recommendation | International Standard is:
- The Enterprise Language [Rec. ITU-T X.911 | ISO/IEC 15414], which refines and extends the enterprise language defined in Rec. ITU-T X.903 | ISO/IEC 10746-3 to enable full enterprise viewpoint specification of an ODP system.

0.2 UML

The Unified Modelling Language (UML) is a visual language for specifying and documenting the artefacts of systems. It is a general-purpose modelling language that can be used with all major object and component methods and that can be applied to all application domains (e.g., in health, finance, telecommunications, or aerospace) and implementation platforms (e.g., J2EE, CORBA®, .NET).

The version of UML currently adopted as an International Standard (ISO/IEC 19505) is UML 2.4.1. UML version 2 has been structured modularly, with the ability to select only those parts of the language that are of direct interest. It is extensible, so it can be easily tailored to meet the specific user requirements. The UML specification defines thirteen types of diagram, divided in two categories that represent, respectively, the static structure of the objects in a system (structure diagrams) and the dynamic behaviour of the objects in a system (behaviour diagrams). In addition, UML incorporates extension mechanisms that allow the definition of new dialects of UML (managed using UML profiles) to customize the language for particular platforms and domains.
The UML specification is defined using a metamodelling approach (i.e., a metamodel is used to specify the model that comprises UML). That metamodel has been constructed so that the resulting family of UML languages is fully aligned with the rest of the OMG specifications (e.g., MOF™, OCL, XMI®) and to allow the exchange of models between tools.

0.3 Overview and motivation

Part 3 of the reference model, Rec. ITU-T X.903 | ISO/IEC 10746-3 defines a framework for the specification of ODP systems 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.

This Recommendation | International Standard defines:

- use of the viewpoints prescribed by the RM-ODP to structure UML system specifications;
- rules for expressing RM-ODP viewpoint languages and specifications with UML and UML extensions (e.g., UML profiles).

It allows UML tools to be used to process viewpoint specifications, facilitating the software design process. Currently there is growing interest in the use of UML for system modelling. However, there is no widely agreed approach to the structuring of such specifications. This adds to the cost of adopting the use of UML for system specification, hampers communication between system developers and makes it difficult to relate or merge system specifications where there is a need to integrate IT systems.

The RM-ODP defines essential concepts necessary to specify open distributed processing systems from five prescribed viewpoints and provides a framework for the structuring of specifications for distributed systems. However, the RM-ODP prescribes neither a notation, nor a model development method.

This Recommendation | International Standard provides the necessary framework for ODP system specification using UML. It defines both a UML based notation for the expression of such specifications, and an approach for structuring of them using the notation, thus providing the basis for model development methods.

By defining how UML and UML extensions should be used to express RM-ODP viewpoint specifications, the standard enables the ODP viewpoints and ODP architecture to provide the needed framework for system specification using UML.

This Recommendation | International Standard contains the following annexes:

- Annex A: An example of ODP specifications using UML;
- Annex B: An example of the representation of deontic concepts.

These annexes are not normative.
ISO/IEC 19793:2015 (E)

INTERNATIONAL STANDARD
ITU-T RECOMMENDATION

Information technology – Open distributed processing –
Use of UML for ODP system specifications

1 Scope
This Recommendation | International Standard defines use of the unified modelling language (UML 2.4.1 superstructure specification, ISO/IEC 19505-2, for expressing system specifications in terms of the viewpoint specifications defined by the reference model of open distributed processing (RM-ODP, Rec. ITU-T X.901 to X.904 | ISO/IEC 10746 Parts 1 to 4) and the Enterprise Language (Rec. ITU-T X.911 | ISO/IEC 15414). It covers:

a) the expression of a system specification in terms of RM-ODP viewpoint specifications using defined UML concepts and extensions (e.g., structuring rules, technology mappings, etc.);

b) relationships between the resultant RM-ODP viewpoint specifications.

This Recommendation | International Standard is intended for the following audiences:

– ODP modellers who want to use the UML notation for expressing their ODP specifications in a graphical and standard way;
– UML modellers who want to use the RM-ODP concepts and mechanisms to structure their UML system specifications;
– modelling tool suppliers, who wish to develop UML-based tools that are capable of expressing RM-ODP viewpoint specifications.

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 | International 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 Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards


2.2 Additional References

3 Definitions

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

3.1 Definitions from ODP standards

3.1.1 Modelling concept definitions

This Recommendation | International Standard makes use of the following terms as defined in Rec. ITU-T X.902 | ISO/IEC 10746-2:

- abstraction; action; activity; architecture; atomicity; behaviour (of an object); binding; class; client object; communication; composition; component object [2-5.1]; composite object; configuration (of objects); conformance point; consumer object; contract; creation; data; decomposition; deletion; distributed processing; distribution transparency; <X> domain; entity; environment; environment contract; epoch; error; establishing behaviour; failure; fault; <X> group; identifier; information; initiating object; instance; instantiation (of an <X> template); internal action; interaction; interchange reference point; interface; interface signature; interworking reference point; introduction; invariant; location in space; location in time; name; naming context; naming domain; notification; object; obligation; ODP standards; ODP system; open distributed processing; perceptual reference point; permission; persistence; producer object; programmatic reference point; prohibition; proposition; quality of service; reference point; refinement; role; server object; spawn action; stability; state (of an object); subdomain; subtype; supertype; system; <X> template; term; terminating behaviour; trading; type (of an <X>); viewpoint (on a system).

3.1.2 Viewpoint language definitions

This Recommendation | International Standard makes use of the following terms as defined in Rec. ITU-T X.903 | ISO/IEC 10746-3:

- binder; capsule; channel; cluster; community; computational behaviour; computational binding object; computational object; computational interface; computational viewpoint; dynamic schema; engineering viewpoint; distributed binding; enterprise object; enterprise viewpoint; <X> federation; information object; information viewpoint; interceptor; invariant schema; node; nucleus; operation; protocol object; static schema; stream; stub; technology viewpoint; <viewpoint> language.

3.2 Definitions from the Enterprise Language

This Recommendation | International Standard makes use of the following terms as defined in Rec. ITU-T X.911 | ISO/IEC 15414:

- actor (with respect to an action); agent; artefact (with respect to an action); authorization; commitment; community object; declaration; delegation; evaluation; field of application (of a specification); interface role; objective (of an <X>); party; policy; prescription; principal; process; resource (with respect to an action); scope (of a system); step; violation.

3.3 Definitions from the Unified Modeling Language

This Recommendation | International Standard makes use of the following terms as defined in ISO/IEC 19505-2:

- abstract class; action; activity; activity diagram; aggregate; aggregation; association; association class; association end; attribute; behaviour; behaviour diagram; binary association; binding; call; class; classifier; classification; class diagram; client; collaboration; collaboration occurrence; comment; communication diagram; component; component diagram; composite; composite structure diagram; composition; concrete class; connector; constraint; container; context; delegation; dependency; deployment diagram; derived element; diagram; distribution unit; dynamic classification; element; entry action; enumeration; event; exception; execution occurrence; exit action; export; expression; extend; extension; feature; final state; fire; generalizable element; generalization; guard condition; implementation; implementation class; implementation inheritance; import; include; inheritance; initial state; instance; interaction; interaction diagram; interaction overview diagram; interface; internal transition; lifeline; link; link end; message; metaclass; metamodel; method; multiple classification;
4 Abbreviations

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

- **BEO** Basic Engineering Object
- **IXIT** Implementation extra Information for Test
- **MOF** Meta Object Facility
- **OCL** Object Constraint Language
- **ODP** Open Distributed Processing
- **OMG** Object Management Group
- **QoS** Quality of Service
- **RM-ODP** Reference Model of Open Distributed Processing
- **UML** Unified Modeling Language
- **UOD** Universe Of Discourse
- **XMI** XML Metadata Interchange

**NOTE** – UML, CORBA, XMI, MOF, OMG, Object Management Group, and Unified Modeling Language are either registered trademarks or trademarks of Object Management Group, Inc. in the United States or other countries.

5 Conventions

In the text that follows, the following conventions apply.


Rec. ITU-T X.911 | ISO/IEC 15414 (RM-ODP Enterprise Language) is referred to as "the Enterprise Language".

The UML superstructure specification (see [2.2]) is referred to as "the UML specification". The UML notation defined in the UML specification is referred to as "UML".

References to the normative text of this Recommendation | International Standard, to the text of Parts 2 and 3 of the RM-ODP, to the Enterprise Language and to UML are expressed in one of these forms:

- [n.n] – a reference to clause n.n of this Recommendation | International Standard.
- [Part 2 – n.n] – a reference to clause n.n of RM-ODP Part 2;
- [Part 3 – n.n] – a reference to clause n.n of RM-ODP Part 3;
- [E/L – n.n] – a reference to clause n.n of the Enterprise Language;
- [UML – n.n] – a reference to clause n.n of the UML specification;

For example, [Part 2 – 9.4] is a reference to subclause 9.4 of Part 2 of the RM-ODP; and [6.5] is a reference to clause 6.5 of this Recommendation | International Standard. These references are for the convenience of the reader.

**NOTE** – The clauses correspond to the specific dated versions of the documents referenced in clause 2.

In the clauses that follow, except in the headings, terms in *italic* typeface are terms of the RM-ODP viewpoint languages as defined in Parts 2 and 3 of the RM-ODP, or in the Enterprise Language. UML concepts are shown in *sans-serif* typeface. UML *stereotype* names are shown in normal font, enclosed in guillemets (« and »).

The following conventions apply to the UML diagrams:
Association end names are placed at the end of the association that is adjacent to the class playing the role. Association end names are omitted if they do not add meaning to the diagram. In this case, the implied association end name is the name of the class at that end of the association, but starting in lower case.

Cardinalities of associations are placed adjacent to the class that has the cardinality.

Where there are no attributes, the attribute part of the class box is suppressed.

Black diamonds are used to represent whole-part associations, with no cardinality or role name at the whole end of the association, and no role name at the part end of the association. The meaning is that the part cannot exist without exactly one instance of the whole.

Nouns are used in association end names, rather than verbs.

Class names representing ODP concepts start with upper case.

Arrowheads accompanying association names are avoided.

Icons associated with stereotypes are used in some of the UML figures in this Recommendation | International Standard. This is done to aid understanding, but the icons are not normative.

6 Overview of modelling and system specification approach

6.1 Introduction

This clause provides an introduction to this Recommendation | International Standard, covering:

- an overview of ODP system specification concepts;
- an overview of UML concepts;
- an explanation of the relationships between ODP models, the subjects of those models (universes of discourse), and the UML models that express the ODP models;
- an overview of the structuring principles for system specifications defined in the document;
- an explanation of the concept of correspondences (relationships) between viewpoint specifications.

6.2 Overview of ODP concepts (extracted from RM-ODP Part 1)

An overview of the ODP modelling concepts and the structuring rules for their use is given in RM-ODP Part 1 (Rec. ITU-T X.901 | ISO/IEC 10746-1: Overview) and the concepts and structuring rules are formally defined in RM-ODP Parts 2 and 3. The text that follows (i.e., the rest of [6.2]), is abstracted from the text in RM-ODP Part 1. RM-ODP Parts 2 and 3 are the authoritative standards, and should be followed in case of any conflict between those Parts and this clause.

The framework for system specification provided by the RM-ODP has four fundamental elements:

- an object modelling approach to system specification;
- the specification of a system in terms of separate but interrelated viewpoint specifications;
- the definition of a system infrastructure providing distribution transparencies for system applications;
- a framework for assessing system conformance.

6.2.1 Object modelling

Object modelling provides a formalization of the well-established design practices of abstraction and encapsulation:

- Abstraction allows the description of system functionality to be separated from details of system implementation;
- Encapsulation allows the hiding of heterogeneity, the localization of failure, the implementation of security and the hiding of the mechanisms of service provision from the service user.
The object modelling concepts cover:

- basic modelling concepts: providing rigorous definitions of a minimum set of concepts (action, object, interaction and interface) that form the basis for ODP system descriptions and are applicable in all viewpoints;
- specification concepts: addressing notions such as type and class that are necessary for reasoning about specifications and the relations between specifications, providing general tools for design, and establishing requirements on specification languages;
- structuring concepts: building on the basic modelling concepts and the specification concepts to address recurrent structures in distributed systems, and covering such concerns as policy, obligation, naming, behaviour, dependability and communication.

6.2.2 Viewpoint specifications

A viewpoint (on a system) is an abstraction that yields a specification of the whole system related to a particular set of concerns. Five viewpoints have been chosen to be both simple and complete, covering all the domains of architectural design. These five viewpoints (see Figure 1) are:

- the enterprise viewpoint, which is concerned with the purpose, scope and policies governing the activities of the specified system within the organization of which it is a part;
- the information viewpoint, which is concerned with the kinds of information handled by the system and constraints on the use and interpretation of that information;
- the computational viewpoint, which is concerned with the functional decomposition of the system into a set of objects that interact at interfaces – enabling system distribution;
- the engineering viewpoint, which is concerned with the infrastructure required to support system distribution;
- the technology viewpoint, which is concerned with the choice of technology to support system distribution.

For each viewpoint there is an associated viewpoint language which can be used to specify a system from that viewpoint. The object modelling concepts give a common basis for the viewpoint languages and make it possible to identify relationships between the different viewpoint specifications and to assert correspondences between the models of the system in different viewpoints (see [6.7]).

NOTE – Although the different viewpoints can be independently defined and there is no explicit order imposed by the RM-ODP for specifying them, a common practice is to start by developing the enterprise specification of the system, and then prepare the
information and computational specifications. These two specifications may have constraints over each other. An iterative specification process is quite common too, whereby each viewpoint specification may be revised and refined as the other two are developed. Correspondences between the elements of these three viewpoints are defined during this process. After that, the engineering specification of the system is prepared, based on the computational specification. Correspondences between the elements of these viewpoints are then defined together with the newly specified elements. Finally, the technology specification is produced based on the engineering specification. Again, some refinements may be performed on the rest of the viewpoint specifications, due to the new requirements and constraints imposed by the particular selection of technology.

6.2.3 Distribution transparency

Distribution transparencies enable complexities associated with system distribution to be hidden from applications where these complexities are irrelevant to the application's purpose. For example:

- access transparency masks differences of data representation and invocation mechanisms for services between systems;
- location transparency masks the need for an application to have information about location in order to invoke a service;
- relocation transparency masks the relocation of a service from applications using it;
- replication transparency masks the fact that multiple copies of a service may be provided in order to provide reliability and availability.

ODP standards define functions and structures to realize distribution transparencies. However, there are performance and cost trade-offs associated with each transparency and only selected transparencies will be relevant in many cases. Thus, a conforming ODP system shall implement those transparencies that it supports in accordance with the relevant standards, but it is not required to support all transparencies.

6.2.4 Conformance

The basic characteristics of heterogeneity and evolution imply that different parts of a distributed system can be purchased separately, from different vendors. It is therefore very important that the behaviours of the different parts of a system are clearly defined, and that it is possible to assign responsibility for any failure to meet the system's specifications.

The framework defined to govern the assessment of conformance addresses these issues. RM-ODP Part 2 defines four classes of reference points: programmatic reference point, perceptual reference point, interworking reference point, and interchange reference point. The reference points in those classes are the candidate for conformance points. Part 2 covers:

- identification of the reference points within an architecture that provide candidate conformance points within a specification of testable components;
- identification of the conformance points within the set of viewpoint specifications at which observations of conformance can be made;
- definition of classes of conformance point;
- specification of the nature of conformance statements to be made in each viewpoint and the relation between them.

6.2.5 Enterprise language

The enterprise language provides the modelling concepts necessary to model an ODP system in the context of the business or organization in which it operates. An enterprise specification defines the purpose, scope, and policies of an ODP system and it provides the basis for checking conformance of system implementations. The purpose of the system is defined by the specified behaviour of the system while policies capture further restrictions of the behaviour between the system and its environment, or within the system itself related to the business decisions of the system owners.

NOTE 1 – An enterprise specification of a system may therefore be thought of as a statement of the "requirements" for the system. However, it must be emphasized that it is not fundamentally different from any other element of the specification for the system.

In an enterprise specification, the system is modelled by one or more enterprise objects within the communities of enterprise objects that model its environment, and by the roles in which these objects are involved. These roles model, for example, the users, owners and providers of information processed by the system.

NOTE 2 – There is a question of modelling style to be considered that has particular significance for an enterprise specification, which is intended to be approachable for a subject matter expert. This is concerned with whether to name model elements in terms of instances or types. Thus it is common practice to express an enterprise specification in terms of anonymous objects,
named by their type, e.g., including in enterprise specifications phrases such as "a customer enterprise object fulfils the role applicant", when what is actually meant is "an (anonymous) enterprise object, conforming to the enterprise object type customer, fulfils the role applicant".

An important aspect of an enterprise specification is the expression of deontic constraints, such as obligation, permission and prohibition. Concepts are included to simplify the expression of the dynamics of these constraints by representing them as objects that can be transferred between communities, allowing the description of delegation and of transfer of responsibility.

6.2.6 Information language

The individual components of a distributed system should share a common understanding of the information they communicate when they interact, or the system will not behave as expected. These items of information are handled, in one way or another, by information objects in the system. To ensure that the interpretation of these items is consistent, the information language defines concepts for the specification of the meaning of information stored within, and manipulated by, an ODP system, independently of the way the information processing functions themselves are to be implemented.

Information held by the ODP system about entities in the real world, including the ODP system itself, is modelled in an information specification in terms of information objects and their relationships and behaviour. Basic information elements are modelled by atomic information objects. More complex information is modelled as composite information objects each modelling relationships over a set of constituent information objects.

The information specification comprises a set of related schemata, namely, the invariant, static and dynamic schemata:

- an invariant schema models relationships between information objects that must always be true, for all valid behaviours of the system;
- a static schema models assertions that must be true at a single point in time. A common use of static schema is to specify the initial state of an information object;
- a dynamic schema specifies how the information can evolve as the system operates.

6.2.7 Computational language

The computational viewpoint is directly concerned with the distribution of processing, but not with the interaction mechanisms that enable distribution to occur. The computational specification decomposes the system into computational objects performing individual functions and interacting at interfaces. It thus provides the basis for decisions on how to distribute the jobs to be done because objects can be located independently, assuming communications mechanisms can be defined in the engineering specification to support the behaviour at the interfaces to those objects.

The heart of the computational language is the computational object model, which constrains the computational specification by defining:

- the form of interface an object can have;
- the way that interfaces can be bound and the forms of interaction that can take place at them;
- the actions an object can perform, in particular the creation of new objects and interfaces, and the establishment of bindings.

The computational object model provides the basis for ensuring consistency between different engineering and technology specifications (including programming languages and communication mechanisms) since they must be consistent with the same computational object model. This consistency allows open interworking and portability of components in the resulting implementation.

The computational language enables the specifier to model constraints on the distribution of an application (in terms of environment contracts associated with individual interfaces and interface bindings of computational objects) without specifying the actual degree of distribution in the computational specification; this latter is specified in the engineering and technology specifications. This ensures that the computational specification of an application is not based on any unstated assumptions affecting the distribution of engineering and technology objects. Because of this, the configuration and degree of distribution of the hardware on which ODP applications are run can easily be altered, subject to the stated environment constraints, without having a major impact on the application software.

6.2.8 Engineering language

The engineering language focuses on the way object interaction is achieved and on the resources needed for it to take place. It defines concepts for describing the infrastructure required to support selectable, distribution transparent
interactions between objects, and rules for structuring communication channels between objects, and for structuring systems for the purposes of resource management. These rules can be modelled as engineering templates (for example, an engineering channel template).

Thus the computational viewpoint is concerned with when and why objects interact, while the engineering viewpoint is concerned with how they interact. In the engineering language, the main concern is the support of interactions between computational objects. As a consequence, there are very direct links between the viewpoint descriptions: computational objects are visible in the engineering viewpoint as basic engineering objects and computational bindings, whether implicit or explicit, are visible as either channels or local bindings.

The concepts and rules are sufficient to enable specification of internal interfaces within the infrastructure, enabling the definition of distinct conformance points for different transparencies and the possibility of standardization of a generic infrastructure into which standardized transparency modules can be placed.

The engineering language assumes a virtual machine that corresponds to a platform offering minimal support for distribution.

NOTE – The functionality of the virtual machine assumed by the engineering language corresponds, for example, to a set of computing systems with stand-alone operating system facilities plus communication facilities. In practice, the functionality available from current vendor technology, for example when it offers a CORBA or J2EE environment, already provides significant elements of the functionality to be covered by the engineering specification.

Thus, the engineering specification is interpreted in this Recommendation | International Standard as defining the mechanisms and functions required to support distributed interaction between objects in an ODP system, making use of the supporting functionality provided by the specific vendor technology defined by the technology specification.

6.2.9 Technology language

The technology specification describes the implementation of the ODP system in terms of a configuration of technology objects modelling the hardware and software components of the implementation. It is constrained by cost and availability of technology objects (hardware and software products) that would satisfy this specification. These may conform to implementable standards, which are effectively templates for technology objects. Thus, the technology viewpoint provides a link between the set of viewpoint specifications and the real implementation, by listing the standards used to provide the necessary basic operations in the other viewpoint specifications; the aim of the technology specification is to provide the extra information needed for implementation and testing by selecting standard solutions for basic components and communication mechanisms.

6.3 Overview of UML concepts

The unified modelling language (UML) is a visual language for specifying, constructing and documenting the artefacts of systems. It is a general-purpose modelling language that can be used with all major object and component methods and that can be applied to all application domains (e.g., in health, finance, telecommunications, or aerospace) and implementation platforms (e.g., J2EE, CORBA, .NET). However, not all of UML modelling capabilities are necessarily useful in all domains or applications. Therefore, the UML specification has a modular structure, with the ability to select only those parts of the language that are of direct interest, and is extensible, so it can be easily customized.

The UML specification defines thirteen types of diagram, divided in two categories that represent, respectively, the static structure of the objects in a system (structure diagrams), and the dynamic behaviour of the objects in a system (behaviour diagrams). In addition, the UML specification incorporates extension mechanisms that allow the definition of new dialects of UML to customize the language for particular platforms and domains.

6.3.1 Structural models

Structural models specify the structure of objects in a model. They are represented in:

– class diagrams, which show a collection of declarative (static) model elements, such as classes, types, and their contents;
– object diagrams, which encompass objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a communication diagram;
– component diagrams, which show the organizations and dependencies among components;
– deployment diagrams, which represent the execution architecture of systems. They represent system artefacts as nodes, which are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments;
– composite structure diagrams, which depict the internal structure of a classifier, including the interaction points of the classifier to other parts of the system. They show the configuration of parts that jointly perform the behaviour of the containing classifier;
– package diagrams, which depict how model elements are organized into packages and the dependencies among them, including package imports and package extensions.

6.3.2 Behavioural models

Behavioural models specify the behaviour of objects in a model. They are represented by:
– use case diagrams, each of which illustrates the relationships among actors and the system, and use cases;
– state machine diagrams, which specify the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions;
– activity diagrams, which depict behaviour using a control and data-flow model;
– interaction diagrams, which emphasize object interactions and can be one of the following:
  – sequence diagrams, that depict interactions by focusing on the sequence of messages that are exchanged, along with their corresponding event occurrences on the lifelines. Unlike a communication diagram, a sequence diagram includes time sequences, but does not include object relationships. A sequence diagram can exist in a generic form (that describes all possible scenarios) and in an instance form (that describes one actual scenario). Sequence diagrams and communication diagrams express similar information, but show it in different ways;
  – communication diagrams, which focus on the interactions between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numbering scheme. Sequence diagrams and communication diagrams express similar information, but show it in different ways;
  – interaction overview diagrams, which represent interactions through a variant of activity diagrams in a way that promotes overview of the control flow; in these diagrams each node can itself be an interaction diagram;
  – timing diagrams, which show the change in state or condition of a lifeline (representing a classifier instance or classifier role) over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli.

6.3.3 Model management

Model management concerns the structuring of a model, including any extensions used, in terms of the groupings of model elements that comprise it. There are three grouping elements:
– models, which are used to capture different views of a physical system;
– packages, which are used within a model to group model elements;
– subsystems, which represents behavioural units in the physical system being modelled.

6.3.4 Extension mechanisms

UML provides a rich set of modelling concepts and notations that have been carefully designed to meet the needs of typical software modelling projects. However, users may sometimes require additional features beyond those defined in the UML specification.

UML can be extended in two ways. First, a new dialect of UML can be defined by using profiles to customize the language for particular platforms (e.g., J2EE/EJB, .NET/COM+) and domains (e.g., in health, finance, telecommunications, or aerospace). Alternatively, a new language related to UML can be specified by reusing part of the UML InfrastructureLibrary package and augmenting it with appropriate metaclasses and metarelationships. The former case defines a new dialect of UML, while the latter case defines a new member of the UML family of languages.

A profile is a kind of package that extends a reference metamodel. The primary extension construct is the stereotype, which defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of or in addition to the ones used for the base metaclass being extended. Just like a class, a stereotype may have properties, which are referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties are referred to as tagged values.
Constraints are frequently defined in a profile, and typically define well-formedness rules that are more constraining, but consistent with, those specified by the reference metamodel. The constraints that are part of the profile are evaluated when the profile has been applied to a package, and need to be satisfied in order for the model to be well formed.

6.4 Universes of discourse, ODP specifications and UML models

In using the techniques described in this Recommendation | International Standard, it is necessary to understand the relationships between the subject of a model, i.e., its universe of discourse (UOD), ODP specifications for that UOD, and how those ODP specifications are expressed in UML.

The four main sets of notions involved in understanding these relationships are:

- the entities, and the relationships amongst them, in the UOD being modelled;
- the ODP specifications that model that UOD;
- the UML models that express the ODP specifications;
- the UML notation (diagramming techniques and other mechanisms) by means of which the UML models are represented.

There are three important kinds of relationship between these notions:

- first, in the same way that an ODP object models an entity (a concrete or abstract thing of interest), an ODP specification models a UOD. The modeller uses the concepts and structuring rules of RM-ODP Part 2, together with those of the relevant ODP viewpoint languages (RM-ODP Part 3 and the Enterprise Language), to produce a specification that models relevant facts and assertions about the entities that exist in the UOD. The rules for this kind of relationship are stated in Parts 2 and 3 of the RM-ODP and in the Enterprise Language;
- second, each model element (i.e., instance of an ODP viewpoint language concept) in the ODP specifications is expressed by one or more UML elements (instance of a UML metaclass, specialized as necessary through the relevant profile) in a UML model, which is thus an expression of the ODP specification. The rules for this kind of relationship are stated in this Recommendation | International Standard;
- third, the UML notation is used to represent, graphically or otherwise, the underlying UML model. The rules for this kind of relationship are stated in the UML standard.

This Recommendation | International Standard addresses the three simple relationships described above, and the terms that are highlighted above are invariably used to refer to them.

While there are other derived relationships between elements in this chain (e.g., between UOD and UML model), they are not otherwise referred to in this Recommendation | International Standard. These relationships are illustrated in Figure 2.
6.5 Modelling concepts and UML profiles for ODP viewpoint languages and correspondences

Clauses 7 to 11 of this Recommendation | International Standard are devoted, in turn, to each of the five ODP Viewpoints (enterprise, information, computational, engineering, and technology).

The first subclause of each of these clauses provides an overview of the ODP modelling concepts for that viewpoint. The ODP viewpoint modelling concepts are described using text as well as a simplified set of UML class diagrams, which show the major modelling concepts for the ODP viewpoint as classes, and binary associations (including cardinality constraints) that may exist between these viewpoint concepts. These diagrams together with the text can be considered as specifying MOF compliant metamodels for the subset of the ODP viewpoint concepts defined in Parts 2 and 3 of the ODP reference model that are used in this Recommendation | International Standard.

NOTE 1 – In the case of the Enterprise Language, the metamodel is standardized in Rec. ITU-T X.911 | ISO/IEC 15414 and reproduced here; if there is any discrepancy, the Enterprise Language version is definitive.

The second subclause of each of these clauses provides a specification of a UML profile for that ODP viewpoint. UML based ODP viewpoint models can be expressed using the notation defined for the UML profile for that viewpoint.

Any ODP viewpoint model expressed using the UML profile for that ODP viewpoint satisfies the constraints specified in each of the corresponding ODP viewpoint metamodels defined in this Recommendation | International Standard.

NOTE 2 – It is an implementation issue whether the constraints defined in each ODP viewpoint metamodel are enforced by tools which construct ODP viewpoint models using that viewpoint’s ODP profile.

Clause 12 deals with correspondences between viewpoints, and is structured in the same way as clauses 7 to 11.

6.6 General principles for expressing and structuring ODP system specifications using UML

This clause defines the structuring style for ODP system specifications, expressed using the UML profiles defined in Clauses 7 to 12 of this Recommendation | International Standard. ODP system specifications that are in compliance with this Recommendation | International Standard will use this structuring style.

The ODP system specification will consist of a single UML model stereotyped as «ODP_SystemSpec», that contains a set of models, one for each viewpoint specification, each stereotyped as «<X>_Spec», where «<X>» is the viewpoint concerned. Each viewpoint specification, which consists of a coherent set of instances of the concepts described in that viewpoint language, uses the appropriate UML profile for that language, as described in Clauses 7 to 11 of this Recommendation | International Standard. There will also be a set of correspondence specifications (see clause 12).

In this Recommendation | International Standard, stereotypes are used to represent domain specific specializations of UML metaclasses in order to express the semantics of the RM-ODP viewpoint language concerned.

In general, the way in which the UML is used to express a given viewpoint specification (which will consist of a coherent set of instances of the concepts described in each viewpoint language) is such that:

- each of the viewpoint language concepts is expressed by one or more extended UML metaclasses (expressed by the use of stereotypes);
the relationships (meta-associations) between the viewpoint language concepts (e.g., "a community has exactly one objective" in the enterprise language) is similarly expressed, preferably by meta-associations between the corresponding UML metaclasses (e.g., "Class may be associated with Class") or, failing that, by use of specific additional UML elements.

This is done in a way that is consistent with the semantics of the UML metamodel.

6.7 Correspondences between viewpoint specifications

6.7.1 ODP Correspondences

The correspondences between viewpoint specifications are defined in Part 3 of the RM-ODP and in the Enterprise Language. The text that follows in this clause is abstracted from these standards, which remain the authoritative standards, and should be followed in case of conflicts between this Recommendation | International Standard and those standards.

A set of specifications of an ODP system written in different viewpoint languages should not make mutually contradictory statements 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 specification to another viewpoint specification, showing that the consistency requirement is met.

The key to consistency is the idea of correspondences between different viewpoint specifications, i.e., a statement that some terms or structures in one specification correspond to other terms and specifications in a second specification. The underlying rationale in identifying correspondences between different viewpoint specifications of the same ODP system is that there are some entities that are modelled in one viewpoint specification, which are also modelled in another viewpoint specification. The requirement for consistency between viewpoint specifications is driven by the fact that what is specified in one viewpoint specification about an entity needs to be consistent with what is said about the same entity in any other viewpoint specification. This includes the consistency of that entity's properties, structure and behaviour.

The specifications produced in different ODP viewpoints are each complete statements in their respective languages, with their own locally significant names, and so cannot be related without additional information in the form of correspondence statements that make clear how constraints from different viewpoints apply to particular elements of a single system to determine its overall behaviour. The correspondence statements are statements that relate the various different viewpoint specifications, but do not form part of any one of them. The correspondences can be established in two ways:

by declaring correspondences between terms in two different viewpoint languages, stating how their meanings relate. This implies that the two languages are defined in such a way that they have a common, or at least a related, set of foundation concepts and structuring rules. Such correspondences between languages necessarily imply and entail correspondences relating to all things of interest which the languages are used to model (e.g., things modelled by objects or actions);

by considering the extension of terms in each language, and asserting that particular entities being modelled in the two specifications are in fact the same entity. This relates the specifications by identifying which observations need to be interpretable in both specifications.

The correspondence statements to be provided in a system specification are specified in Part 3 and in the Enterprise Language of the RM-ODP, and in clauses 7 to 11 of this Recommendation | International Standard. They fall into two categories:

some correspondences are required in all ODP specifications; these are called required correspondences. If the correspondence is not valid in all instances in which the concepts related occur, the specification simply is not a valid ODP specification;

in other cases, there is a requirement that the specifier provides a list of items in two specifications that correspond, but the content of this list is the result of a design choice; these are called required correspondence statements.

NOTE – In RM-ODP Part 3, the following correspondences are explicitly specified:

between computational and information ([Part 3 – 10.1]);
between engineering and computational ([Part 3 – 10.2]).

In the Enterprise Language standard, the following correspondences are specified;

between enterprise and information ([E/L – 11.2]);
between enterprise and computational ([E/L – 11.3]);
6.7.2 Expressing ODP correspondences in UML

Correspondences between ODP modelling elements of different viewpoints are expressed using the UML profile defined in clause 12 of this Recommendation | International Standard. The main concept introduced is the correspondence link. A correspondence link is established between two viewpoint specifications, and each of its ends refers to a set of terms involved in the correspondence relationship. A correspondence statement is expressed by a constraint applied to this link, and is used for checking consistency between viewpoint specifications.

7 Enterprise specification

7.1 Modelling concepts

An enterprise specification uses the RM-ODP enterprise language. The modelling concepts and the structuring rules of the enterprise language are defined in [Part 3 – 5] and expanded upon in [E/L – 6 and 7]. They are summarized in this clause. In case of conflict between the explanations herein and the text in Part 3 or the Enterprise Language, the latter documents should be followed.

The set of diagrams at the end of this clause (i.e., at [7.1.8]) summarizes a metamodel for the enterprise language, defined in Rec. ITU-T X.911 | ISO/IEC 15414.

7.1.1 System concepts

An enterprise specification describes an ODP system and relevant aspects of its environment. An ODP System is a kind of enterprise object. The enterprise objects that interact with a given enterprise object form part of the environment of that enterprise object.

The ODP System has a scope, which defines the behaviour that the system is expected to exhibit. An enterprise specification has a field of application which describes its usability properties.

These system concepts are illustrated in Figure 3.

7.1.2 Community concepts

The fundamental concept of the enterprise language is a community, which is a configuration of enterprise objects, formed to meet an objective. Any objective may be refined into a set of subobjectives. A community is specified in a contract, which models the agreement amongst the entities to work together to meet the objective. Thus the contract:

- states the objective for which the community exists;
- governs the structure, the behaviour and the policies of the community;
- constrains the behaviour of the members of the community;
- states the rules for the assignment of enterprise objects to roles.

Each enterprise object models some entity (abstract or concrete thing of interest) in the UOD. A particular kind of enterprise object is a community object, which models, as a single object, an entity that is elsewhere in the model refined as a community.

The configuration of a community is modelled in terms of the way enterprise objects interact in fulfilling roles, which identify behaviours intended to meet the objective of the community concerned.

The community concepts are illustrated in Figure 4.

7.1.3 Behaviour concepts

A behaviour is a collection of actions (things that happen), with constraints on when they occur. An enterprise object may be involved in (play roles in) an action in one or more of the following three ways:

- if it participates in the action it is an actor with respect to that action;
- if it is referenced (i.e. mentioned) in the action, it is an artefact with respect to that action;
- if it is essential to the (performance of) that action, and requires allocation or may become unavailable, it is a resource with respect to that action.
A role identifies a specific behaviour of an enterprise object in a community. Such behaviour is observable as a set of interactions in which the object participates, and relationships between them. This implies that the behaviour of an object has to be viewed in the context of the corresponding behaviour of the objects with which it interacts.

Communities may be open or closed; that is they may or may not interact with their environment. Where a role that is in (i.e., is part of the configuration of) a community identifies behaviour that takes place with the participation of one or more objects that are not in that community, it is an interface role.

The modelling of behaviour may be structured into one or more processes, each of which is a graph of steps taking place in a prescribed manner and which contributes to the fulfilment of an objective. In this approach, a step is an abstraction of an action in which the enterprise objects that participate in that action may be unspecified. A step may be refined as a process, itself consisting of a set of steps.

A violation is a specific behaviour of an enterprise object that is prohibited in a community contract. The contract may specify some specific behaviour that is to take place when a violation occurs.

The behaviour concepts are illustrated in Figures 4 and 5.

### 7.1.4 Deontic concepts

The specification of enterprise behaviour typically involves the expression of deontic constraints such as obligations, permissions and prohibitions. These are incorporated into an object-based model by introducing enterprise objects called deontic tokens. If an active enterprise object has an associated deontic token, then the corresponding deontic constraint applies to the object's behaviour. However, deontic tokens are not themselves active enterprise objects and are not directly involved in interactions by taking action roles. Each deontic token is associated with exactly one active enterprise object. There are three types of deontic token:

- a burden represents an obligation on the objects with which it is associated;
- a permit represents a permission held by the objects with which it is associated;
- an embargo represents a prohibition affecting the objects with which it is associated.

These deontic constraints are created or modified by specific types of action which are called speech acts. A speech act may result in the creation of deontic tokens or the transfer of such tokens between objects playing particular action-roles in the speech act. The destruction of a token at the end of its lifecycle is also generally performed by a speech act, although tokens may destroy themselves as a result of a timeout or other trigger.

NOTE – The set of tokens held by the objects concerned determines whether a speech act can take place and what its consequences are. For example:

- it may be necessary for an object to hold a permit before it can perform a speech act;
- having an embargo may prevent an object from performing a speech act, even though the action would otherwise be permitted by the object's role;
- a burden held by an object may be discharged as a result of its performing a speech act;
- the performance of a delegation speech act may transfer a group of tokens (for example, burdens and permits) to the object to which responsibility is delegated.

A deontic token may be in either an active or a pending state. When it is in an active state, the constraint it carries is applied to control the behaviour of the active enterprise object that holds it. However, when it is in the pending state, this constraint is masked so that it does not affect the current behaviour.

The deontic concepts are illustrated in Figures 4, 7 and 8.

### 7.1.5 Policy concepts

A policy is a constraint on a system specification foreseen at design time, but whose detail is determined subsequent to the original design, and capable of being modified from time to time in order to manage the system in changing circumstances. It identifies the specification of behaviour, or constraints on behaviour, that can be changed during the lifetime of the ODP system, or that can be changed to tailor a single specification to apply to a range of different ODP systems.

The specification of a policy includes:

- the name of the policy;
- the rules, modelled as obligations, permissions, prohibitions and authorizations;
- the elements of the enterprise specification affected by the policy;
the policy envelope that constrains the possible restrictions or behaviours that are acceptable as policy values;

any behaviour for changing the policy;

a default policy value to be used until any explicit initial change takes place.

Where there is a requirement to model dynamic policy setting, a policy can be changed by a behaviour.

A policy may also constrain the structure (configuration) of a community, by governing the assignment of roles to enterprise objects. Such a policy is called an assignment policy.

NOTE – For a given policy envelope, only one policy value is in force at a point in time. This policy value may be selected from a set of values defined in the policy envelope or it may be a statement in a policy language that is consistent with constraints in the policy envelope.

The policy concepts are illustrated in Figure 6.

7.1.6 Accountability concepts

Accountability concepts concern the modelled behaviour of parties. A party is an enterprise object modelling a natural person or any other entity considered to have some of the rights, powers and duties of a natural person, and which can therefore be considered accountable for its actions. A party may delegate authority to another enterprise object (which may or may not be a party), in which case it is referred to as the principal in that action of delegation, and the enterprise object to whom authority is delegated is the agent of that party.

Only parties can take part in accountable actions. Such actions may take the following forms:

– prescription: an action that establishes a rule;
– commitment: an action resulting in an obligation by one or more of the participants in the act to comply with a rule or perform a contract;
– declaration: an action that establishes a state of affairs in the environment of the object making the declaration;
– evaluation: an action that assesses the value of something;
– delegation: an action that assigns authority, responsibility or a function to another object.

The accountability concepts are illustrated in Figure 7.

7.1.7 Structure of an enterprise specification

An enterprise specification is structured in terms of communities and community objects.

Each community is modelled in terms of the following concepts and the relationships between them:

– the objective and subobjectives (of the community);
– the behaviour of the community, modelled in terms of actions and constraints on the order in which they may occur. Behaviour can be structured to emphasize:
  – roles fulfilled by enterprise objects that interact as members of the community;
  – processes that model sequences of actions, carried out by one or more enterprise objects;
  – enterprise objects that fulfil the roles in the community;
  – policies constraining the behaviour;

Some enterprise objects may be composite objects and are subclassified as community objects and refined as communities.

At some level of detail the ODP system will be present in the model as an enterprise object.

7.1.8 Summary of the enterprise language metamodel

The diagrams below (Figures 3 to 8) illustrate the concepts of the enterprise language and the relationships between them.
NOTE – The concept of environment was introduced in Part 2 in order to allow description of the properties of some particular object by introducing a representation of all the other elements in a model with which it might interact, directly or indirectly. As such, in particular, it represents some abstraction of the other objects in the model, but this abstraction relationship is not visible in any model.
Figure 4 – Community concepts

Constraints on DeonticToken

```plaintext
(-,- Tokens can only be applied to Conditional Actions
  self.actionRole -> forAll (action.oclIsTypeOf(ConditionalAction)))
```

```plaintext
(-,- derived attribute:
  context DeonticToken
def referencedEOs = Set(EnterpriseObject) =
  self.actionRole actionRoleFiller)
```

Constraints on Community Behaviour

```plaintext
(self.objective ->notEmpty())
```

Constraints on Action

```plaintext
(-,- Actions are atomic, i.e., cannot be decomposed
context Action
  inv self.component ->notEmpty()
(-,- At least one ActionRole has to be an actor
context Action
  inv AtLeastOneActorAsActionRole:
    self.actionRole ->oclIsTypeOf(Actor) ->notEmpty())
```

Rec. ITU-T X.906 (10/2014) 17
Figure 5 – Behaviour concepts

Figure 6 – Policy concepts
7.2 UML profile

NOTE – This Recommendation | International Standard defines UML expressions for concepts likely to be found in concrete elements within practical specifications. Some of the ODP concepts are abstract, expressing the categorization of the practical concepts. The abstract concepts are mentioned, but no UML expression is offered.

This clause specifies how the ODP enterprise concepts described in the previous clause are expressed in UML in an enterprise specification. A brief explanation of the UML concepts used in the expression of each concept is given, together with a justification of the expression used.

7.2.1 ODP system

An ODP System is an enterprise object. It is expressed in UML by an instanceSpecification of a UML class stereotyped as «EV_ODPSystem», see [7.2.6]. That class expresses the enterprise object type. Note also that modelling purposes
may require that an ODP system be further detailed as a community, in which case the enterprise object that models it is classified as a community object and refined as a community, see [7.2.4].

7.2.2 Scope

The scope of an ODP system is the set of behaviours that the system is expected to exhibit, e.g., its roles. It is not, therefore, expressed by any single UML element, but by the set of elements that express its behaviour.

7.2.3 Field of application

The field of application is a property of the enterprise specification as a whole, and is expressed by a tag definition of stereotype «Enterprise_Spec». This tag definition is named EV_FieldOfApplication, and is of type string. That string contains the description of the field of application of the enterprise specification.

7.2.4 Community

A community is modelled in terms of its type, which is expressed by a component stereotyped as «EV_Community». This is included in a package stereotyped as «EV_CommunityContract» that contains the specification of the community, i.e., its objective, its behaviour, and any enterprise objects and object types that are specific to the community concerned (see [7.2.9]). Where a specific entity (e.g., organizational unit) is being modelled it is expressed by an instanceSpecification of a component stereotyped as «EV_Community».

Any component expressing a community will have exactly one association, stereotyped as «EV_ObjectiveOf» to a class stereotyped as «EV_Objective», that expresses the objective of the community, and a set of realizations, each stereotyped as «EV_CommunityBehaviour», to the UML classifier elements expressing its roles and the associated behaviour (interactions, actions, steps and processes).

See also [7.2.8] and [7.2.9].

7.2.5 Enterprise object

An enterprise object is generally specified in terms of its type, which is expressed by a class stereotyped as «EV_Object».

NOTE — The UML concept of class is different to the ODP concept of class. A UML class is a "description" of a set of objects, while an ODP class is the set of objects itself. Therefore, the UML concept of class is closer to the ODP concept of type, and there is no UML concept corresponding to the ODP concept of class. Therefore, no UML expression for the ODP concept of class is provided.

Any class stereotyped as «EV_Object» may have any number of associations, each stereotyped as «EV_FulfilsRole», with any number of classes stereotyped as «EV_Role» in one or more community, modelling the fact that the enterprise objects of that type fulfil the roles.

Where an enterprise object is required to represent a specific entity in the UOD, it is expressed by an instanceSpecification of a class that is stereotyped as «EV_Object».

7.2.6 Object types and templates as enterprise objects

There are cases where there is the need to model the type or template of an enterprise object at the instance level. An example is the case of a generic factory, which is invoked by passing it a representation of a template (which has type template), and responds by instantiating the template and returning a reference to the created object. To indicate that an object is derived from a given template, we need to represent both the template object and the instantiated object in the model. Likewise for types, to indicate that an object conforms to a given type, we need to represent both the object and its object type in the model.

Both type objects and template objects are enterprise objects, and therefore are expressed by classes that express their type or template. To distinguish them from other enterprise objects, such classes are stereotyped «EV_TypeObject» or «EV_TemplateObject», respectively. Both stereotypes inherit from «EV_Object».

The relationship between an enterprise object and the object that represents its template, or the objects that represent its types can be expressed as an attribute of the class that expresses the enterprise object.

For example, in some specifications, such as in the ODP trading function specification, there is the need to specify the type of a service, so the trader can locate objects implementing such a service. The diagram shown in Figure 9 represents the specification of an enterprise object, PrintService, and of its type, PrintServiceType, expressed so that the object is able to know and access its type (i.e., the type of the object is accessible as part of its metadata, by means of an attribute of the class that expresses its specification)
Figure 9 – An explicit representation of the type of an enterprise object so that the object can access its type

7.2.7 Community object

A community object is an enterprise object that is refined in the model as a community. Like any other enterprise object a community object is modelled in terms of its type, which is expressed by a class stereotyped as «EV_CommunityObject», which is, in turn, a specialization of «EV_Object». This class has a dependency, stereotyped as «EV_RefinesAsCommunity», to the component stereotyped as «EV_Community» which expresses the type of the community that refines it.

7.2.8 Objective

An objective (of a community) is expressed by a class, stereotyped as «EV_Objective». This class has an association, stereotyped as «EV_ObjectiveOf» with the component, stereotyped as «EV_Community» that describes the community being specified.

NOTE – When an objective is refined into subobjectives, the subobjective is also expressed by a class stereotyped «EV_Objective» and the relationship between objective and subobjectives will be a composition.

7.2.9 Contract

A contract for a community specifies the objective of that community, and how that objective can be met (i.e., its behaviour and policies). It is the specification of that community as it appears in the enterprise specification. The expression of contract is by a package stereotyped as «EV_CommunityContract».

In the name space of the package will be the UML elements expressing the community itself, its objective, its roles and the associated behaviour (actions, interactions, steps and processes), and the policy and accountability concepts specific to the community. Relationships between all these UML elements may also be included in this package’s namespace. The package may also contain some or all of the elements expressing the enterprise objects that fulfil its roles. (Those elements expressing enterprise objects that fulfil roles in other communities may be contained in any one of the packages expressing those communities.)

7.2.10 Behaviour

7.2.10.1 General

NOTE – In this clause phrases such as "interactions between roles" and "steps performed by roles" should be read as "interactions between enterprise objects fulfilling roles" and "steps performed by enterprise objects fulfilling roles" respectively.

A behaviour is a set of actions with constraints on when they may occur. It is not expressed by any single UML element. It is expressed by a set of elements expressing the behaviour as a set of processes of a community in which the steps are behaviours of roles in the community. Where required, the behaviour of a role can be further detailed in terms of a set of elements expressing the behaviour in terms of internal actions of the role and interactions between the role and other roles in the community. Where behaviour needs to be expressed in a more generic way than given below, a state machine stereotyped as «EV_Behaviour» is used.

Annex A illustrates the application of the concepts described in the following clauses (7.2.10.2 and 7.2.10.3).

7.2.10.2 Behaviour as processes and steps

Where the behaviour is modelled in terms of processes of a community, a process is expressed by an activity stereotyped as «EV_Process» in the namespace of the component, stereotyped as «EV_Community», that expresses the community that uses this process to achieve its objective. This activity has a realization link, stereotyped as «EV_CommunityBehaviour» from that component. Within this activity:

– the steps of the process are expressed by callBehaviorActions, stereotyped as «EV_Step»;
– the refinement of a step, as a process, is expressed by associating the relevant callBehaviorAction, stereotyped as «EV_Step», that expresses the step, with an activity, stereotyped as «EV_Process», that expresses the refinement;
activityPartitions (stereotyped as «EV_Role») represent classes (also stereotyped as «EV_Role») that express the roles of the enterprise objects in the name space of the package (stereotyped as «EV_CommunityContract») that expresses the community in which the role is specified; similarly, activityPartitions (stereotyped as «EV_Object») represent classes (also stereotyped as «EV_Object») that express the enterprise objects defined as local to the community concerned;

where a step is not refined as a process, the callBehaviorAction, stereotyped as «EV_Step», that expresses the step, is associated with an opaqueBehavior specified in the context of the corresponding class stereotyped as «EV_Role» that expresses the role of the enterprise object that performs the step;

NOTE – An opaqueBehavior can express, in an appropriate language, any level of detail about the step that is required to meet the modelling objectives.

the artefacts that are referenced in the steps are expressed by objectNodes, stereotyped as «EV_Artefact».

In general, the complete behaviour for a role is modelled by the actions for that role in a number of processes.

7.2.10.3 Behaviour as interactions between roles

The detailed behaviour of individual roles may be expressed by the following combination of UML elements:

one or more classes each having one or more associations with the class stereotyped as «EV_Role» that expresses the role being specified. Each of these classes is stereotyped as «EV_Interaction». The relationship is expressed with an association, stereotyped as «EV_InteractionInitiator» or «EV_InteractionResponder» as appropriate;

each class stereotyped as «EV_Interaction» will have associations with classes that are stereotyped as «EV_Role», where there is an interaction between these roles. An «EV_Interaction» is composed of signals, each also stereotyped as «EV_Interaction». Enterprise objects that are referenced in the interactions are represented by the values of the properties of the signals;

each class stereotyped as «EV_Interaction» shall have an operation +occur():void that defines the behaviour that takes place when the interaction occurs. It can also be used to specify pre- and post-conditions on the interaction, which can be represented as OCL constraints on the occur() operation. This operation links the static, template-like, view of interactions with the occurrences of the interaction in the community behaviour;

one or more stateMachines for which the context is the class stereotyped as «EV_Role», that define the constraints on the receiving and sending of information by an enterprise object fulfilling the role and any associated internal actions of the enterprise object. Each of these stateMachines shows the sending and receiving of the signals, each stereotyped as «EV_Artefact», associated with the interactions of the role, and thus shows the logical ordering of these interactions, and defines the internal actions of the role in terms of the behaviors associated with the states.

An assignment policy is expressed in the same way as any other policy; see 7.2.15.

The internal actions identified in (the states of) the stateMachines for the «EV_Role» correspond to the actions in an activityPartition expressing the role in the corresponding activityDiagrams, and the properties of the signals correspond to the objectNodes in the corresponding activityDiagrams.

7.2.10.4 Interface role

An interface role is expressed by a class stereotyped as an «EV_InterfaceRole», which inherits from «EV_Role». The part of the behaviour identified by the interface role that takes place with the participation of one or more external objects (objects that do not form part of the decomposition of the community object that is refined by that community) is modelled by an interaction with a role that identifies the required behaviour of the external objects. This behaviour is expressed by a class stereotyped as «EV_Interaction» that has associations with each of the classes (stereotyped as «EV_InterfaceRole») that express the interface role on the one hand and some roles or local objects within the community (stereotyped as «EV_Role» or «EV_Object») on the other.

7.2.10.5 Violation

A violation is expressed by a state machine stereotyped as «EV_Violation» that inherits from «EV_Behaviour».
7.2.11 Action Roles

7.2.11.1 Actor (with respect to an action)

The concept actor is a relationship between an enterprise object and an action. There is no single UML element that expresses an instance of the RM-ODP enterprise language concept, actor. Actors in a model may be identified from either or both of:

- an examination of the interaction model where the existence of actors will be indicated by the associations, stereotyped as «EV_FulfilsRole», between the classes stereotyped as «EV_Role» and «EV_Object», respectively, taken in combination with the stateMachine that expresses the behaviour of the relevant role;
- in an examination of the process model, the presence of an «EV_Step» in an «EV_Role» activityPartition indicates that the enterprise object fulfilling the role is an actor for the step concerned.

7.2.11.2 Artefact (with respect to an action)

The concept artefact is also a relationship between an enterprise object and an action. In an interaction model, an artefact referenced in an action is expressed by a signal stereotyped as «EV_Artefact», which has two associations:

- one association, stereotyped as «EV_ArtefactRole», will be with the «EV_Object» class expressing the enterprise object that is an artefact with respect to the action;
- the other association, stereotyped as «EV_ArtefactReference», will be with the «EV_Interaction» class that expresses the action or interaction for which the enterprise object is an artefact.

In a process model, it is possible to express each instance of artefact with a single UML element, namely an objectFlow stereotyped as «EV_Artefact».

7.2.11.3 Resource (with respect to an action)

No specific UML metaclass is extended to express this concept. If required, the fact that some behaviour requires the existence of an enterprise object as a resource may be stated in a comment on that behaviour.

7.2.12 Deontic concepts

7.2.12.1 Burden

A burden is expressed in UML either as a class stereotyped as «EV_Burden» or as an ObjectNode stereotyped as «EV_Burden».

The presence of an obligation is implied by the representation of a burden. If a less specific expression is required, the fact that some behaviour places or fulfils an obligation may be stated in a constraint stereotyped as «EV_Obligation» on that behaviour.

NOTE – The specifier selects an appropriate level of detail for the specification. Obligations are either reified as burdens or represented as constraints, but the style chosen in a particular specification should be consistent.

7.2.12.2 Permit

A permit is expressed in UML either as a class stereotyped as «EV_Permit» or as an ObjectNode stereotyped as «EV_Permit».

The presence of a permission is implied by the representation of a permit. If a less specific expression is required, the fact that some behaviour requires or creates a permission may be stated in a constraint stereotyped as «EV_Permission» on that behaviour.

NOTE – The specifier selects an appropriate level of detail for the specification. Permissions are either reified as permits or represented as constraints, but the style chosen in a particular specification should be consistent.

7.2.12.3 Embargo

An embargo is expressed in UML either as a class stereotyped as «EV_Embargo» or as an ObjectNode stereotyped as «EV_Embargo».

The presence of a prohibition is implied by the representation of an embargo. If a less specific expression is required, the fact that some behaviour requires or creates a prohibition may be stated in a constraint stereotyped as «EV_Prohibition» on that behaviour.
NOTE – The specifier selects an appropriate level of detail for the specification. Prohibitions are either reified as embargos or represented as constraints, but the style chosen in a particular specification should be consistent.

7.2.13 Policy

Policies are expressed in UML using a combination of elements, which together are used to express the following:

– the policy itself, including its current value and the envelope that defines the range of values that are possible;
– the objects and the behaviour constrained by the policy;
– the behaviour by which the policy value may be changed and objects that are allowed to exhibit that behaviour.

The policy is expressed by a class stereotyped as «EV_PolicyDeclaration», with a constraint stereotyped as «EV_PolicyEnvelopeRule» expressing the range permitted for the policy.

Each policy value is expressed by a class stereotyped as «EV_PolicyValue» which has the role "current value" in an association with the «EV_PolicyDeclaration» class that expresses the policy. In this way, the policy allows the policy envelope to restrict the current policy value.

![Diagram of UML expression of a policy](image)

**Figure 10 – Pattern for UML expression of a policy**

Where the enterprise specification includes elements modelling the behaviour concerned with setting the policy value, this is modelled by roles identifying behaviour that may be detailed as processes or interactions, with associations, stereotyped as «EV_PolicySettingBehaviour», between the classes expressing the policy envelope and the classes expressing the behaviour.

The relationships between a policy and the behaviours that it constrains are expressed by one or more dependencies, stereotyped as «EV_AffectedBehaviour», from the classes expressing the behaviours to the class expressing the policy.

Unless the set of policy values is pre-determined, a set of constraints stereotyped as «EV_PolicyEnvelopeRule» expressing rules governing acceptable policy values is attached to the «EV_PolicyDeclaration» class.

Attached to each «EV_PolicyValue» class is a set of constraints stereotyped as «EV_PolicyValueRule», which together express behaviour rules related to the policy value. These rules, which may comprise obligations, permissions, prohibitions, authorizations, or other expressions, may be expressed in OCL or other suitable notation.

The pattern for expression of policy and its impact on other parts of an enterprise specification is shown in Figure 10.
7.2.14 Accountability concepts

7.2.14.1 Party

A party is an enterprise object modelling an entity with some of the rights, powers and duties of a natural person. It is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Party», which must also be stereotyped as «EV_Object».

7.2.14.2 Accountable action

An action may be accountable when it is part of the behaviour identified by a role fulfilled by a party. This is expressed in UML with an association, stereotyped as «EV_Accountable», between the class expressing the role and the class or activity expressing the interaction or process respectively in which the accountable party participates.

NOTE – Where this construct is used for a process, this only indicates that the role is accountable for those steps that it performs, and not for those performed by some other role. This is a limitation of the semantics of the UML approach chosen, as it is not possible to associate a classifier with the element expressing steps.

7.2.14.3 Authorization

An authorization is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Authorization», which is a specialization of «EV_Behaviour».

7.2.14.4 Delegation

A Delegation is expressed in UML by an association, stereotyped as «EV_Delegation», between two classes stereotyped as «EV_Object» with association ends showing the party which is the principal and the enterprise object which is the agent to whom the delegation is made.

7.2.14.5 Principal

A principal is an enterprise object modelling an entity responsible for the acts of its agent in consequence of some delegation. It is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Principal», which is a specialization of «EV_Object».

7.2.14.6 Agent

An agent is an enterprise object modelling an entity performing acts on behalf of a principal in consequence of some delegation. It is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Agent», which is a specialization of «EV_Object».

7.2.14.7 Prescription

A prescription is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Prescription», which is a specialization of «EV_Behaviour».

7.2.14.8 Commitment

A commitment is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Commitment», which is a specialization of «EV_Behaviour».

7.2.14.9 Declaration

A declaration is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Declaration», which is a specialization of «EV_Behaviour».

7.2.14.10 Evaluation

An evaluation is expressed in UML by an instanceSpecification of a class stereotyped as «EV_Evaluation», which is a specialization of «EV_Behaviour».

7.2.15 Summary of UML extensions for the enterprise language

The enterprise language profile (EV_Profile) specifies how the enterprise viewpoint modelling concepts relate to and are expressed in standard UML using stereotypes, tag definitions, and constraints.

The following diagrams (Figures 11 to 15) show a graphical representation of the UML profile for the enterprise language, using the notation provided by UML.
Figure 11 – Model management

Figure 12 – Classifiers
Figure 13 – Activities

Figure 14 – Constraints
7.3 Enterprise specification structure (in UML terms)

An enterprise specification is contained in a model, stereotyped as «Enterprise_Spec». At the top level within this model there are one or more packages, stereotyped as «EV_CommunityContract», that include, where necessary, classes, each stereotyped as «EV_CommunityObject», expressing the relevant communities as community objects.
Within each «EV_CommunityContract» package, there is a single component, stereotyped as «EV_Community» and a single class, stereotyped as «EV_Objective», as well as other elements, packaged as convenient, expressing behaviour (roles, processes and interactions), and enterprise objects that are local to the community.

7.4 Viewpoint correspondences for the enterprise language

7.4.1 Contents of this clause

This clause describes the correspondence concepts for the enterprise language, but not how they are expressed in UML. The latter is covered in clause 12.

7.4.2 Enterprise and information viewpoint specification correspondences

In general, not all the elements of the enterprise specification of a system need to correspond to elements of its information specification. However, the information viewpoint shall conform to the policies of the enterprise viewpoint and, likewise, all enterprise policies shall be consistent with the static, dynamic, and invariant schemata of the information specification.

Where there is a correspondence between enterprise and information elements (e.g., between an enterprise object and the information object that stores the relevant information about it), the specifier shall provide:

- for each enterprise object in the enterprise specification, a list of those information objects (if any) that model information or information processing concerning the entity modelled by that enterprise object;
- for each role in each community in the enterprise specification, a list of those information object types (if any) that specify information or information processing of an enterprise object fulfilling that role;
- for each policy in the enterprise specification, a list of the invariant, static and dynamic schemata of information objects (if any) that correspond to the enterprise objects to which that policy applies; an information object is included if it corresponds to the enterprise community that is subject to that policy;
- for each action in the enterprise specification, the information objects (if any) subject to a dynamic schema constraining that action;
- for each relationship between enterprise objects, the invariant schema (if any) which constrains objects in that relationship;
- for each relationship between enterprise roles, the invariant schema (if any) which constrains objects fulfilling roles in that relationship.

7.4.3 Enterprise and computational viewpoint specification correspondences

Not all the elements of the enterprise specification of a system need to correspond to elements of its computational specification. In particular, not all states, behaviours and policies of an enterprise specification need to correspond to states and behaviours of a computational specification. There may exist transitional computational states within pieces of computational behaviour which are abstracted as atomic transitions in the enterprise specification.

Where there is a correspondence between enterprise and computational elements, the specifier shall provide:

- for each enterprise object in the enterprise specification, that configuration of computational objects (if any) that realizes the required behaviour;
- for each interaction in the enterprise specification, a list of those computational interfaces and operations or streams (if any) that correspond to the enterprise interaction, together with a statement of whether this correspondence applies to all occurrences of the interaction, or is qualified by a predicate;
- for each role affected by a policy in the enterprise specification, a list of the computational object types (if any) that exhibit choices in the computational behaviour that are modified by the policy;
- for each interaction between roles in the enterprise specification, a list of computational binding object types (if any) that are constrained by the enterprise interaction;
- for each enterprise interaction type, a list of computational behaviours capable of carrying out an interaction of that enterprise interaction type.

7.4.4 Enterprise and engineering viewpoint specification correspondences

Not all the elements of the enterprise specification of a system need to correspond to elements of its engineering specification. Where there is a correspondence between enterprise and engineering elements, the specifier shall provide:
– for each enterprise object in the enterprise specification, the set of those engineering nodes (if any) with their nuclei, capsules, and clusters, all of which support some or all of its behaviour;
– for each interaction between roles in the enterprise specification, a list of engineering channel types and stubs, binders, protocol objects and interceptors (if any) that are constrained by the enterprise interaction.

NOTE 1 – The engineering nodes may result from rules about assigning support for the behaviour of enterprise objects to nodes. These rules may capture policies from the enterprise specification.
NOTE 2 – The engineering channel types and stubs, binders or protocol objects may be constrained by enterprise policies.

7.4.5 Enterprise and technology viewpoint specification correspondences

In accordance with [Part 2 – 15.5] and [Part 3 – 5.3], an implementer provides, as part of the claim of conformance, the chain of interpretations that permits observation at conformance points to be interpreted in terms of enterprise concepts. While there may be specific correspondences between enterprise policies and technology viewpoint specifications that require the use of particular technologies, there are neither required correspondences nor required correspondence statements.

NOTE – Although there are no required viewpoint correspondences between enterprise and technology specifications, there may be cases where part of an enterprise specification has a direct relationship with a technology specification or a choice of technology. Such examples include enterprise policies covering performance (e.g., response time), reliability, and security.

8 Information specification

8.1 Modelling concepts

An information specification uses the RM-ODP information language. The modelling concepts and the structuring rules of the information language are defined in [Part 3 – 6]. They are summarized in this clause. Except where otherwise stated, in case of conflict between the explanations herein and the text in Part 3, the latter document should be followed.

The set of diagrams at the end of this clause (i.e., at [8.1.10]) summarizes a metamodel for the information language. The information viewpoint is concerned with information modelling. It focuses on the semantics of information and information processing in the ODP system. The individual components of a distributed system must share a common understanding of the information they communicate when they interact, or the system will not behave as expected. These items of information are handled, in one way or another, by one or more objects in the system. To ensure that the interpretation of these items is consistent, the information language defines concepts for the specification of the meaning of information stored within, and manipulated by, an ODP system, independently of the way the information processing functions themselves are to be implemented.

In the ODP reference model, the information language uses a basic set of concepts and structuring rules, including those from Part 2 of RM-ODP, and three concepts specific to the information viewpoint: invariant schema, static schema, and dynamic schema.

8.1.1 Information object

Information held by the ODP system about entities in the real world, including the ODP system itself, is modelled in an information specification in terms of information objects, and their relationships and behaviour.

Basic information elements are modelled by atomic information objects. More complex information is modelled as composite information objects modelling relationships over a set of constituent information objects. Information objects, as any other ODP object, exhibit behaviour, state, identity, and encapsulation.

NOTE – Information objects may have operations, although information operations are names for significant stimuli for state changes, and are not necessarily the same as computational operations.

8.1.2 Information object type

The type of an information object is a predicate characterizing a collection of information objects.

8.1.3 Information object class

A class of information objects is the set of all information objects satisfying a given type.
8.1.4 Information object template

An information object template is the specification of the common features of a collection of information objects in sufficient detail that an information object can be instantiated using it. Information object templates may reference static, invariant and dynamic schemata.

8.1.5 Information action and action types

An action is a model of something that happens in the real world. Types of actions are modelled by action types. An action in the information viewpoint is associated with at least one information object.

Actions can be either internal actions or interactions. An internal action always takes place without the participation of the environment of the object. An interaction takes place with the participation of the environment of the object. Objects can only interact at interfaces. ODP interactions are instances of ODP communications.

8.1.6 Invariant schema

An invariant schema is 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.

An invariant schema can also describe the specification of the types of one or more information objects, that will always be satisfied by whatever behaviour the objects might exhibit.

8.1.7 Static schema

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

8.1.8 Dynamic schema

A dynamic schema is a specification of the allowable state changes of one or more information objects, subject to the constraints of any invariant schemata. A dynamic schema specifies how the information can evolve as the system operates. In addition to describing state changes, dynamic schemata can also create and delete information objects, and allow reclassifications of instances from one type to another. Furthermore, in the information language, a state change involving a set of objects can be regarded as an interaction between those objects. Not all the objects involved in the interaction need to change state; some of the objects may be involved in a read-only manner.

8.1.9 Structure of an information specification

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 these objects, and environment contracts for the objects in the system. More precisely, an information specification is structured in terms of:

- a configuration of information objects, described by a set of static schemata;
- the behaviour of those information objects, described by a set of dynamic schemata; and
- the constraints that apply to either of the above (invariant schemata).

The different schemata may apply to the whole system, or they may apply to particular domains within it. Particularly in large and rapidly evolving systems, the reconciliation and federation of separate information domains will be one of the major tasks to be undertaken in order to manage information.

There are also some considerations that need to be taken into account when specifying the information viewpoint of an ODP system:

- information objects are either atomic or are modelled as a composition of component information objects. When an information object is a composite object, the schemata are composed as well;
- 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;
- the configuration of information objects is independent from distribution, i.e., there is no sense or focus on distribution in this viewpoint.
8.1.10 Summary of the information language metamodel

The diagram below (Figure 16) illustrates the concepts of the information language and the relationships between them. The descriptions of the concepts have been given above. The descriptions of the relationships between the concepts are included in the description of the concepts.

![Diagram of information language concepts](image)

Figure 16 – Information language concepts

8.2 UML profile

This clause specifies how the ODP information concepts described in the previous clause are expressed in UML in an information specification. A brief explanation of the UML concepts used in the expression of each concept is given, together with a justification of the expression used.

NOTE – In this clause UML expressions are only defined for those concepts for which use has been demonstrated through an example, included in the main body of this Recommendation | International Standard or in its annexes. Where no example has been identified, the concept concerned is mentioned, but no UML expression is offered.

8.2.1 Information object

An information object is generally specified in terms of its type, which is expressed by a class stereotyped as «IV_Object».

Where an information object is required to represent a specific entity in the UOD, it is expressed by an instanceSpecification of a class that is stereotyped as «IV_Object».

8.2.2 Object types and templates as information objects

There are cases where there is the need to model the type or template of an information object at the instance level. An example is the case of a generic factory, which is invoked by passing it a representation of a template (which has type template), and responds by instantiating the template and returning a reference to the created object. To indicate that an object is derived from a given template, we need to represent both the template object and the instantiated object in the model. Likewise for types, to indicate that an object conforms to a given type, we need to represent both the object and its object type in the model.

Both type objects and template objects are information objects, and therefore are expressed by classes that express their type or template. To distinguish them from other information objects, such classes are stereotyped «IV_TypeObject» or «IV_TemplateObject», respectively. Both stereotypes inherit from «IV_Object».
The relationship between an information object and the object that represents its template, or the objects that represent its types, can be expressed as an attribute of the class that specifies the information object.

For example, the diagram shown in Figure 17 represents the specification of an information object, Loan, and of its type, MyLoanType, expressed so that the object is able to know and access its type (i.e., the type of the object is accessible as part of its metadata, by means of an attribute of the class that expresses its specification).

![Diagram](image-url)

**Figure 17 – An explicit representation of the type of an information object so that the object can access its type**

### 8.2.3 Information action and action types

An interaction is expressed by a signal sent or received by the stateMachines of the information objects concerned. An action type is expressed by a signal stereotyped as «IV_Action».

In the information viewpoint, actions are mainly used for describing events that cause state changes, or for implementing communications between objects, i.e., flows of information.

In an information specification, an internal action is expressed by an internal transition of a state of the stateMachine for the information object concerned.

### 8.2.4 Relationships between information objects and between information object types

A relationship between information object types, when modelled as part of the state of the objects of those types, can be expressed by an association between the classes expressing those types. Instances of these associations (i.e., links) will express the relationships between the information objects.

When associations between information objects are modelled in ODP as invariant schemata, the UML expressions defined in clause 8.2.5 apply.

### 8.2.5 Invariant schema

Invariant schemata may impose different kinds of constraints in an information specification.

First, invariant schemata can provide the specification of the types of one or more information objects, that will always be satisfied by whatever behaviour the objects might exhibit. This kind of invariant schema may be expressed in a UML Package stereotyped as «IV_InvariantSchema», which specifies a set of object types (in terms of the set of classes that express such object types), their possible relationships (expressed by associations), and constraints on those object types, on their relationships, and possibly on their behaviours (expressed by the specification of the corresponding stateMachines). The association multiplicities and the constraints on the different modelling elements will constrain the possible states and state changes of the elements to which they apply.

**NOTE 1** – OCL is the recommended notation for expressing the constraints on the modelling elements that form part of the UML expression of an invariant schema. However, other notations can be used when OCL does not provide enough expressive power, or is not appropriate due to the kind of expected user of the specification. For example, a temporal logic formula or an English text can be used for expressing a constraint that imposes some kind of fairness requirement on the behaviour of the system (e.g., "Objects of class X will produce requests to objects of class Y, no later than a given time T after condition A on objects of classes X, Y and Z is satisfied").

There are cases, however, in which an invariant schema in an information viewpoint specification is defined over a set of concrete information objects. Such a kind of invariant schema may be expressed by a package stereotyped as «IV_InvariantSchema», that contains the corresponding set of objects. The constraints on these objects, together with the specifications of the classifiers of these objects, constrain the possible states and state changes of the objects.

**NOTE 2** – The classifiers of the objects will constrain the possible states and state changes of the objects to which they apply (through the associations, stateMachines, and constraints of these classifiers).

Finally, individual constraints stereotyped as «IV_InvariantSchema» can also be used to express invariant schemata.
8.2.6 Static schema

A static schema is expressed by a package stereotyped as «IV_StaticSchema» of objects, their attribute links, their link ends, which have an associated target link end which is navigable, and their classifiers.

NOTE – The possible associations of the information objects described in a static schema with other objects not contemplated in the schema need not be included in the package, since they are not part of the specification provided by the schema. Therefore, whenever the absence of an association instance (i.e., a link) needs to be expressed, it should be explicitly stated (e.g., by using constraints attached to the appropriate objects).

8.2.7 Dynamic schema

A dynamic schema is expressed in terms of stateMachines for the information objects in the information specification, stereotyped as «IV_DynamicSchema». The actions that relate to the state changes are expressed by signals that are sent and received on transitions of the stateMachines.

8.2.8 Summary of the UML extensions for the information language

The information language profile (IV_Profile) specifies how the information viewpoint modelling concepts relate to, and are expressed in, standard UML using stereotypes, tag definitions, and constraints.

Figure 18 shows the graphical representation of the UML profile for the information language, using the notation provided by UML.

![Graphical representation of the information language profile](image)

**Figure 18 – Graphical representation of the information language profile**

8.3 Information specification structure (in UML terms)

All the elements expressing the information specification are defined within a model, stereotyped «Information_Spec». Such a model contains the packages that express the invariant, static and dynamic schemata of the system.

These packages may be defined and organized as follows:
in the first place, a set of «IV_InvariantSchema» packages with class diagrams will define the information object and object types of the system, their relationships, and the constraints on these elements;

second, a set of «IV_StaticSchema» packages with object diagrams will express the state of the system, or parts of it, at specific locations in time that may be of interest to any of the system stakeholders. The classifiers of the instance Specifications of these diagrams should have been previously defined in the «IV_InvariantSchema» packages that define the structure and composition of the system;

third, dynamic schemata expressed by individual state Machines will be associated with the corresponding elements in the previous packages. Thus, individual state Machines will be associated with the corresponding classifiers or instance Specifications. Likewise, constraints describing invariants and pre- and post-conditions of signals will be associated to the states of the state Machines and with the corresponding classifier definitions;

Finally, a set of «IV_InvariantSchema» constraints will impose further constraints on the elements of all the previous packages. Such constraints can be either directly attached to the corresponding elements, establishing an implicit context by attachment, or they can form part of a separate piece of specification in which the context of each constraint is explicitly established by naming.

8.4 Viewpoint correspondences for the information language

8.4.1 Contents of this clause

This clause describes the correspondence concepts for the information language, but not how they are expressed in UML. The latter is covered in clause 12.

8.4.2 Enterprise and information viewpoint specification correspondences

In general, not all the elements of the enterprise specification of a system need to correspond to elements of its information specification. However, the information viewpoint shall conform to the policies of the enterprise viewpoint and, likewise, all enterprise policies shall be consistent with the static, dynamic, invariant schemata of the information specification.

Where there is a correspondence between information and enterprise elements (e.g., between an enterprise object and the information object that stores the relevant information about it), the specifier shall provide:

- for each enterprise object and for each artefact role in an enterprise action, the corresponding configuration of information objects (if any) that model them in the information viewpoint;
- for each enterprise role, action and process in the enterprise viewpoint, the corresponding dynamic and invariant schema definitions in the information viewpoint that specify that behaviour;
- for each enterprise policy in the enterprise viewpoint, the constraints in the corresponding schemata that implement it, since enterprise policies may become constraints in any of the schemata.

NOTE – In the case of a notional incremental development process of the ODP viewpoint specifications, whereby the information specifications are developed taking into account the previously defined enterprise specifications, information objects may be discovered through examination of an enterprise specification. For example, each artefact referenced in any actions in which an ODP System participates will correspond in some way with one or more information objects.

8.4.3 Information and computational viewpoint specification correspondences

Not all the elements of the information specification of a system need to correspond to elements of its computational specification. In particular, not all states of an information specification need to correspond to states of a computational specification. There may exist transitional computational states within pieces of computational behaviour that are abstracted as atomic transitions in the information specification.

Where an information object corresponds to a set of computational objects, the static and invariant schemata of the 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.
8.4.4 Information and technology viewpoint specification correspondences

While there may be specific correspondences between information schemata and technology viewpoint specifications that require the use of particular technologies, there are neither required correspondences nor required correspondence statements.

NOTE – There may be cases where part of an information viewpoint specification has a direct relationship with a technology viewpoint specification or a choice of technology. Such examples include invariant schemata covering performance (e.g., response time) or security.

9 Computational specification

9.1 Modelling concepts

A computational specification uses the RM-ODP computational language. The modelling concepts and the structuring rules of the computational language are defined in [Part 3 – 7]. Some of the concepts in Part 2 of RM-ODP are also used when defining the computational language concepts. The concepts and structuring rules are summarized in this clause. Except where otherwise stated, in case of conflict between the explanations herein and the text in Parts 2 or 3, the latter document should be followed.

The set of diagrams at the end of this clause (i.e., at [9.1.22]) summarizes a metamodel for the computational language.

NOTE – Another partial metamodel for the computational language can be found in Rec. ITU-T X.960|ISO/IEC 14769: Type Repository Function, which is concerned with the storage and management of computational type systems. That metamodel is therefore a partial view concentrating on the computational type system, rather than on system design in general. Readers should be aware that:

a) cardinality constraints on types are not, in general, the same as the cardinality constraints on instances – an interface must be associated with an object, but an interface type can be defined independently of an object type;
b) the different focus there leads to different choices of primary relations, so that some relations that are explicit in that metamodel are derived in this representation, and vice versa.

If there is any ambiguity, statements in this Recommendation | International Standard take precedence.

9.1.1 Computational object

An object is a model of an entity. An object is characterized by its behaviour and, dually, by its state. An object is distinct from any other object. An object is encapsulated, i.e., any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment.

A computational object is an object as seen in the computational viewpoint. It models functional decomposition and interacts with other computational objects. Since it is an object, it has state and behaviour, and interactions are achieved through interfaces.

9.1.2 Interface [Part 2 – 8.4]

An interface is an abstraction of the behaviour of an object that consists of a subset of the interactions of that object together with a set of constraints on when they can occur.

9.1.3 Interaction [Part 2 – 8.3]

An interaction is one of two defined kinds of actions. Action itself is defined as something that happens, and every action of interest for modelling purposes is associated with at least one object. The set of actions associated with an object is partitioned into internal actions and interactions. An internal action always takes place without the participation of the environment of the object. An interaction takes place with the participation of the environment of the object.

9.1.4 Environment contract [Part 2 – 11.2.3]

Environment contract is a contract between an object and its environment, including Quality of Service (QoS) constraints, usage and management constraints.

QoS constraints include:

– temporal constraints (e.g., deadlines);
– volume constraints (e.g., throughput);
dependency constraints covering aspects of availability, reliability, maintainability, security and safety (e.g., mean time between failures).

QoS constraints can imply usage and management constraints. For instance, some QoS constraints (e.g., availability) are satisfied by provision of one or more distribution transparencies (e.g., replication).

An environment contract can describe both:

- requirements placed on an object's environment for the correct behaviour of the object;
- constraints on the object behaviour in a correct environment.

9.1.5 Behaviour (of an object) [Part 2 – 8.6]

Behaviour of an object is a collection of actions with a set of constraints on when they may occur.

The specification language in use determines the constraints that may be modelled. Constraints may include, for example, sequentiality, nondeterminism, concurrency or real-time constraints.

Behaviour may include internal actions.

The actions that actually take place are restricted by the environment in which the object is placed.

9.1.6 Signal [Part 3 – 7.1.1]

A signal is an atomic shared action resulting in one-way communication from an initiating object to a responding object.

9.1.7 Operation [Part 3 – 7.1.3]

An operation is an interaction between a client object and a server object which is either an interrogation or an announcement.

9.1.2 Announcement [Part 3 – 7.1.3]

An announcement is 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.

9.1.9 Interrogation [Part 3 – 7.1.4]

An interrogation is 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.

9.1.10 Flow [Part 3 – 7.1.5]

A flow is 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.

9.1.11 Signal interface [Part 3 – 7.1.6]

A signal interface is an interface in which all the interactions are signals.

9.1.12 Operation interface [Part 3 – 7.1.7]

An operation interface is an interface in which all the interactions are operations.


A stream interface is an interface in which all the interactions are flows.

9.1.14 Computational object template [Part 3 – 7.1.9]

A computational object template is an object template which comprises a set of computational interface templates that the object can instantiate, a behaviour specification and an environment contract specification.
9.1.15 Computational interface template [Part 3 – 7.1.9]
A computational interface template is 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 environment contract specification.

9.1.16 Signal interface signature [Part 3 – 7.1.11]
A signal interface signature is 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 that instantiates the template.

9.1.17 Operation interface signature [Part 3 – 7.1.12]
An operation interface signature is 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 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.

9.1.18 Stream interface signature [Part 3 – 7.1.13]
A stream interface signature is 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.

9.1.19 Binding object [Part 3 – 7.1.14]
A binding object is a computational object that supports a binding between a set of other computational objects.

9.1.20 Binding [Part 2 – 13.4, Part 3 – 7. 2.3]
A binding behaviour is an establishing behaviour between two or more interfaces (and hence between their supporting objects). The contractual context, resulting from a given establishing behaviour, is called a binding.

In Part 3, 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 objects directly. A compound binding action can be expressed in terms of primitive binding actions linking two or more computational objects via a binding object.

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.

9.1.21 Transparency schema [Part 3 – 16]
A transparency schema identifies those transparencies required by a computational specification. These transparencies are constraints for a mapping from the computational specification to a specification that uses specific ODP functions and engineering structures. It defines a combination of distribution transparencies assumed by the computational specification.

NOTE – As described in [Part 3 – 16], the distribution transparencies include access transparency, failure transparency, location transparency, migration transparency, persistence transparency, relocation transparency, replication transparency, and transaction transparency.
9.1.22 Structure of a computational specification

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

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

The set of computational objects specified by the computational specification constitute a configuration that will change as the computational objects instantiate further computational objects or computational interfaces, perform binding actions, effect control functions upon binding objects, delete computational interfaces or delete computational objects.

The computational language defines a set of rules that constrain a computational specification. These comprise:

- interaction rules, binding rules and type rules that provide distribution transparent interworking;
- template rules that apply to all computational objects and computational interfaces;
- failure rules that apply to all computational objects and identify the potential points of failure in computational activities.

9.1.23 Summary of the concepts of the computational metamodel

Figure 19 illustrates the concepts of the computational language and the relationships between them. The descriptions of the concepts have been given above. The descriptions of the relationships between the concepts are included in the description of the concepts.

NOTE – Some of the relationships between computational language concepts are not shown in Figure 19, e.g., the relationship between interface and signature, since they are related through their supertypes.
Figure 19 – Computational language concepts
The following restrictions apply to the elements of the diagram shown in Figure 19:

- A **binding object** is associated with at least two different **objects**;
- A **binding object** binds two or more **objects** through the same type of **interface** (signal, announcement, interrogation, or flow);
- All **interfaces** associated with a **signal interface signature** are **signal interfaces** [9.2.9], and all its constituent **interaction signatures** are **signal signatures**:
  
  ```
  context Signal inv SignalSignature: self.interface->forAll(oclIsTypeOf(SignalInterface))
  ```

- All **interfaces** associated with an **operation interface signature** are **operation interfaces** [9.2.9], and all its constituent **interaction signatures** are **announcement, interrogation, invocation or termination signatures**:
  
  ```
  context Announcement inv AnnouncementSignature: self.interface->forAll(oclIsTypeOf(AnnouncementInterface))
  context Invocation inv InvocationSignature: self.interface->forAll(oclIsTypeOf(InvocationInterface))
  context Termination inv TerminationSignature: self.interface->forAll(oclIsTypeOf(TerminationInterface))
  context OperationInterface inv OperationInterfaceSignature: self.specifier->forAll(oclIsTypeOf(OperationInterface))
  ```

- All **interfaces** associated with a **stream interface signature** are **stream interfaces** [9.2.9]:
  
  ```
  context Flow inv StreamSignature: self.interface->forAll(oclIsTypeOf(StreamInterface))
  context StreamInterface inv StreamInterfaceSignature: self.specifier->forAll(oclIsTypeOf(StreamInterface))
  ```

### 9.2 UML profile

This clause specifies how the ODP computational concepts described in the previous clause are expressed in UML in a computational specification. A brief explanation of the UML concepts used in the expression of each concept is given, together with a justification of the expression used.

**NOTE 1** – In this clause UML expressions are only defined for those concepts for which use has been demonstrated through an example, included in the main body of this Recommendation | International Standard or in its annexes. Where no example has been identified, the concept concerned is mentioned, but no UML expression is offered.

**NOTE 2** – The concepts and rules of the computational language concern the decomposition of the system’s functionality into computational objects performing individual functions and interacting at interfaces and thus provide the basis for decisions on how to distribute the tasks to be done. This level of abstraction deals with aspects related to the software architecture of the system, and therefore the appropriate UML mechanisms for modelling software architectures are used in this text (components, ports, and interfaces).

**NOTE 3** – The computational viewpoint assumes that the specifier selects a certain level of refinement below which the use of the concept of **computational object** ceases to be essential; these lower level specification concerns, such as the realization of the behaviour of **computational objects**, are outside the scope of the profile described here, and are addressed by other specification techniques and languages, including the direct use of UML concepts and rules. Thus, this profile covers the specification of **computational objects** at the level of UML components that interact through their ports, but leaves open to the specifier the way in which the internal realization of such components is specified.

#### 9.2.1 Computational object

A **computational object** is generally specified in terms of its **template**, which is expressed by a component stereotyped as «CV_Object».

The attribute **isIndirectlyInstantiated** of such a component should be set to true. This attribute constrains the kind of instantiation that applies to a component. If false, the component is instantiated as an addressable instance. If true (default value), the component is defined at design-time, but at runtime (or execution-time) an instance specified by the component does not exist, that is, the component is instantiated indirectly, through the instantiation of its realizing classifiers or parts.

Where a **computational object** is required to represent a specific entity in the UOD, it is expressed by an instance **Specification** of a component that is stereotyped as «CV_Object».

Where there is the need to express a **computational object type**, it is also expressed by a UML component, stereotyped as «CV_Object». The attribute **isIndirectlyInstantiated** of the component stereotyped «CV_Object» should be set to true.
When a component stereotyped as «CV_Object» expresses a computational object template, the attribute isAbstract of such a component should be set to false, meaning that the component needs to provide all the information required to instantiate objects.

9.2.2 Object types and templates as computational objects

There are cases where there is the need to model the type or template of a computational object at the instance level. An example is the case of a generic factory, which is invoked by passing it a representation of a template (which has type template), and responds by instantiating the template and returning a reference to the created object. To indicate that an object is derived from a given template, we need to represent both the template object and the instantiated object in the model. Likewise for types, to indicate that an object conforms to a given type, we need to represent both the object and its object type in the model.

Both type objects and template objects are computational objects, and therefore are expressed by components that express their type or template. To distinguish them from other computational objects, such components are stereotyped «CV_TypeObject» or «CV_TemplateObject», respectively. Both stereotypes inherit from «CV_Object».

The relationship between a computational object and the object that represents its template, or the objects that represent its types, can be expressed as an attribute of the class that specifies the computational object.

For example, in some specifications, such as in the ODP Trading Function specification, there is the need to specify the type of a service, so the trader can locate objects implementing such a service. The diagram shown in Figure 20 represents the specification of a computational object, PrintService, and of its type, PrintServiceType, expressed so that type can be manipulated by computational operations.

![Figure 20 – An explicit representation of the type of a computational object so that the object can access its type](image)

9.2.3 Binding object

A binding object is a kind of computational object, and is expressed by an instanceSpecification of a component, stereotyped as «CV_BindingObject», that represents its type or template.

The following two restrictions apply to binding objects, and therefore to components stereotyped «CV_BindingObject»:

- Any binding object is associated with at least two different objects;
- Any binding object binds two or more objects through the same type of interface (signal, announcement, interrogation, or flow).

9.2.4 Environment contract

An environment contract of a computational object is expressed by a set of constraints (stereotyped «CV_EnvironmentContract») applied to the component that expresses the computational object.

9.2.5 Signal

A signal is expressed by a message, stereotyped as «CV_Signal», sent by an initiating object and received by a responding object.

9.2.6 Announcement

An announcement is expressed by a message, stereotyped as «CV_Announcement», sent by a client object and received by a server object with no response expected.

9.2.7 Invocation

An invocation is a part of interrogation and is expressed by a message, stereotyped as «CV_Invocation», sent by a client object and received by a server object.

9.2.8 Termination

A termination is a part of an interrogation and is expressed by a message, stereotyped as «CV_Termination», sent by a server object and received by a client object.
9.2.9 Computational interface

Computational interface templates are expressed by ports, that can be stereotyped as \( \text{CV\_SignalInterface} \), \( \text{CV\_OperationInterface} \) or \( \text{CV\_StreamInterface} \) depending on the type of interface (signal, operation or stream). Thus, an interface of a computational object is expressed by a port of a component instance, instantiated from the corresponding component that expresses the object's computational interface template.

In order to express the causality of an operation interface, the stereotype \( \text{CV\_OperationInterface} \) has a tag definition, causality, of type OperationCausality (an Enumeration type whose literals are client and server).

In order to express the causality of a signal interface, the stereotype \( \text{CV\_SignalInterface} \) has a tag definition, causality, of type SignalCausality (an Enumeration type whose literals are consumer and producer).

The stereotype \( \text{CV\_StreamInterface} \) does not have any tag definition, because stream interfaces do not have causality.

9.2.10 Computational interface signature

A computational interface signature is expressed by an interface, stereotyped as \( \text{CV\_SignalInterfaceSignature} \), \( \text{CV\_OperationInterfaceSignature} \) or \( \text{CV\_StreamInterfaceSignature} \) depending on the type of interface signature (signal, operation or stream).

9.2.11 Computational signature

A computational signature can be expressed by a reception, an operation, or an interface, depending on the sort of signature. Receptions are used to express signatures of computational interactions which are expressed by individual signals (signals, announcements, invocations and terminations). Operations can be used to express interrogation signatures that are composed of an invocation signature and a termination signature. Finally, interfaces are used for expressing flow signatures [9.2.18].

9.2.12 Signal signature

A signal signature is expressed by a reception, stereotyped as \( \text{CV\_SignalSignature} \). This stereotyped reception expresses an action template which includes the name for the signal, the number, names and types of its parameters, and indication of whether it is initiating or responding.

9.2.13 Announcement signature

An announcement signature is a signature for an announcement. An announcement signature is expressed by a reception, stereotyped as \( \text{CV\_AnnouncementSignature} \). This stereotyped interface expresses an action template which includes the name for the invocation, the number, names and types of its parameters, and an indication of whether it is a client or a server.

9.2.14 Invocation signature

An invocation signature is a signature for an invocation in an interrogation. An invocation signature is expressed by a reception, stereotyped as \( \text{CV\_InvocationSignature} \). This stereotyped reception expresses an action template which includes the name for the invocation, the number, names and types of its parameters, and an indication of whether it is a client or a server.

9.2.15 Termination signature

A termination signature is a signature for a termination for interrogation. A termination signature is expressed by a reception, stereotyped as \( \text{CV\_TerminationSignature} \). This stereotyped reception expresses an action template which includes the name for the termination, the number, names and types of its parameters, and indication of whether it is a client or a server.

The Stereotype \( \text{CV\_TerminationSignature} \) has a tag definition, invocation, whose type is Reception, that refers to the invocation for which this reception is a termination.

9.2.16 Interrogation signature

An interrogation signature is a signature for an interrogation, which comprises signatures for an invocation and a termination.

In the case of an interrogation signature comprising one invocation signature and one termination signature, the interrogation signature can be expressed by an operation, stereotyped as \( \text{CV\_InterrogationSignature} \). This stereotyped operation expresses an action template which includes the name for the invocation, the number, names and types of its parameters, the indication of whether it is a client or a server, and the number, names and types of the termination's parameters.
Alternatively, an interrogation signature can be modelled in terms of one invocation signature [9.2.14] and separate termination signatures [9.2.15].

NOTE – This alternative modelling approach may be used, for example, in the case of an interrogation comprising one invocation and possibly multiple kinds of termination.

9.2.17 Bindings

An explicit primitive binding is expressed by an assembly connector, stereotyped as «CV_PrimitiveBinding». Such a connector can be defined from a required interface to a provided interface, or from a required port to a provided port.

For example, suppose the following representation in UML of operation interface signatures ServiceA and Service, as shown in Figure 21:

![Figure 21 – Two operation interface signatures](image)

Then, the diagram shown in Figure 22 represents an explicit primitive binding between the corresponding interfaces of computational objects ClientA and Server:

![Figure 22 – An explicit primitive binding between two interfaces](image)

As another example, assuming the specification of operation interface signatures ServiceA and Service as above, the diagram shown in Figure 23 represents an explicit primitive binding between the corresponding interfaces of computational objects ClientA and Server, but showing explicitly the interface signatures of both interfaces (stereotypes and tag values of the ports representing such interfaces have been omitted for clarity):

![Figure 23 – An explicit primitive binding between two interfaces showing their interface signatures](image)

The following restrictions apply to assembly connectors, stereotyped as «CV_PrimitiveBinding»:

- If they connect interfaces, they are both stereotyped «CV_OperationInterfaceSignature» and the operation interface signature expressed by the client interface is a subtype of the operation interface signature expressed by the server interface [Part 3 – 7.2.3];
- If they connect ports, then: (a) these ports are stereotyped «CV_SignalInterface», «CV_OperationInterface» or «CV_StreamInterface», (b) their stereotypes coincide, and (c) the interface expressed by the client port is compatible with the interface expressed by the server port, according to the primitive binding rules defined in [Part 3 – 7.2.3];
- If they connect ports stereotyped «CV_StreamInterface», the fact that stream interfaces do not have causality implies that the assignment of direction (that is, the designation of the client element) is irrelevant.

An implicit primitive binding can only happen between interfaces specifying operation interface signatures, and only when the required interface coincides with the provided interface; then there is no need to represent the connector.

NOTE – In this case the "ball and socket" connection representation can be used, as shown in Figure 24.
9.2.18 Flow

A flow is expressed by a property, stereotyped as «CV_Flow». The property belongs to an interface stereotyped as «CV_StreamInterfaceSignature», which represents the stream interface signature where the flow is defined.

The name of the property expresses the name of the flow. The type of the property expresses the flow signature, which is expressed by an interface, stereotyped as «CV_FlowSignature». The causality of the flow (consumer or producer) is expressed by the tag definition, causality, of stereotype «CV_Flow». The type of this tag definition is FlowCausality (an Enumeration type whose literals are producer and consumer).

For example, the diagram shown in Figure 25 represents the software architecture of a teleconference system, composed of two kinds of computational objects (Presenter and Participant) interacting at their computational interfaces.

The Presenter object provides one operation interface for control (expressed by the port ctrl, stereotyped «CV_OperationInterface», whose signature is expressed by the interface IControl), and one stream interface (expressed by the port c, stereotyped «CV_StreamInterface», whose signature is expressed by the interface AVConference). This stream interface defines four flows, one for producing video frames, two for producing audio frames, and one for consuming audio).

The Participant object offers the dual interfaces, one for controlling the Participant, and one for binding to its stream interface.

Control interfaces are bound using an implicit binding, whilst the stream interfaces are bound using a primitive binding, i.e., no binding object is required.

9.2.19 Transparency schema

A transparency schema is expressed by a stereotype «CV_Transparency» defining a set of tags applied to a model that is stereotyped as «Computational_Spec», a «CV_Interface» or a «CV_Object». There is one tag for each of the
transparencies defined in [Part 3 – 16], except for access and location transparencies, which are mandatory for any computational specification.

The type of these tag definitions is boolean, and indicates whether the particular transparency is required for the computational specification or not.

9.2.20 Summary of the UML extensions for the computational language

The computational language profile (CV_Profile) specifies how the computational viewpoint modelling concepts relate to, and are expressed in, standard UML using stereotypes, tag definitions, and constraints.

The following shows diagrammatic representations of this UML profile.

---

**Figure 26 – Graphical representation of the computational language profile (using the UML notation)**

The following constraints apply to the elements of the profile:

- The constraint `baseComponent.isIndirectlyInstantiated=true` means that the component is defined at design-time, but at runtime (or execution-time) an instance specified by the component does not exist, that is, the component is instantiated indirectly, through the instantiation of its realizing classifiers or parts;
- A component expressing a computational object template has ports and interfaces for interaction with other computational objects.

In addition, the elements of the computational language (shown in Figure 19) are subject to a set of restrictions, as described in [9.1.22]. The constraints that implement those restrictions on the corresponding profile elements should also apply.
9.3 Computational specification structure (in UML terms)

All the elements expressing the computational specification are defined within a model, stereotyped «Computational_Spec». Such a model contains packages that express:

- a configuration of computational objects with dependencies among those objects using required and provided interfaces and signatures they provide, with a component diagram;
- structure of computational objects including composition and decomposition of computational objects, with a component diagram;
- environment contract for computational objects, with constraints on elements;
- interactions between computational objects, and interactions between composed computational objects within a computational object, with UML activity diagrams, state charts, and interaction diagrams.

9.4 Viewpoint correspondences for the computational language

9.4.1 Contents of this clause

This clause describes the correspondence concepts for the computational language, but not how they are expressed in UML. The latter is covered in clause 12.

9.4.2 Enterprise and computational viewpoint specification correspondences

The specifier shall provide:

- for each enterprise object in the enterprise specification, the configuration of computational objects (if any) that realizes the required behaviour;
- for each interaction in the enterprise specification, a list of those computational interfaces and operations or streams (if any) that correspond to the enterprise interaction, together with a statement of whether this correspondence applies to all occurrences of the interaction, or is qualified by a predicate;
- for each role affected by a policy in the enterprise specification, a list of the computational object types (if any) that exhibit choices in the computational behaviour that are modified by the policy;
- for each interaction between roles in the enterprise specification, a list of computational binding object types (if any) that are constrained by the enterprise interaction;
- for each enterprise interaction type, a list of computational behaviour types (if any) capable of modelling (i.e., acting as a carrier for) the enterprise interaction type.

If a process based approach is taken, the specifier shall provide:

- for each step in the process, a list of participating computational objects which may fulfil one or more of actor roles, artefact roles, or resource roles.

9.4.3 Information and computational viewpoint specification correspondences

This Recommendation | International Standard does not prescribe exact correspondences between information objects and computational objects. In particular, not all states of a computational specification need to correspond to states of an information specification. There may exist transitional computational states within pieces of computational behaviour that 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.

9.4.4 Computational and engineering viewpoint specification correspondences

Each computational object that 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 that corresponds to the computational object supporting the computational interface.
Where transparencies that replicate objects are involved, each computational interface of the objects being replicated corresponds 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 modelled 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 shall 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 that 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 that provides the necessary consistent view of the computational interaction.

The transparency prescriptions in [Part 3 – 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 optimisations and therefore not excluded.

10 Engineering specification

10.1 Modelling concepts

This clause is based on the modelling concepts for use in an engineering specification that are defined, together with the structuring rules for their use, in [Part 3 – 8]. The explanations of the concepts in the text that follows are not normative and, in case of conflicts between these explanations and the text in [Part 3 – 8], the latter should be followed.

An engineering specification includes the definition of mechanisms and functions required to support distributed interaction between objects in an ODP system. The concepts, rules and structures contained in an engineering specification (the engineering language) are dependent upon the functionality offered by the platform chosen for the ODP system.

The modelling concepts and structuring rules defined in [Part 3 – 8] assume a platform that offers only minimal support for distribution. Where the platform for the system offers significant support for distribution, a language and a UML profile appropriate for that platform can be used.

The set of diagrams at the end of this clause (i.e., at [10.1.5]) summarizes a metamodel for the engineering language.
10.1.1 Basic concepts

10.1.1.1 Basic engineering object

A basic engineering object is an engineering object that requires the support of a distributed infrastructure.

10.1.1.2 Cluster

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

10.1.1.3 Cluster manager

A cluster manager is an engineering object that manages the basic engineering objects in a cluster.

10.1.1.4 Capsule

A capsule is a configuration of engineering objects forming a single unit for the purpose of encapsulation of processing and storage.

10.1.1.5 Capsule manager

A capsule manager is an engineering object that manages the engineering objects in a capsule.

10.1.1.6 Nucleus

A nucleus is an engineering object that coordinates processing, storage and communications functions for use by other engineering objects within the node to which it belongs.

10.1.1.7 Node

A node is a configuration of engineering objects forming a single unit for the purpose of location in space, and that embodies a set of processing, storage and communication functions.

10.1.1.8 Engineering interfaces and signatures

Engineering objects expose engineering interfaces. The set of related concepts dealing with interfaces and their corresponding signatures exactly parallel those defined in the computational viewpoint for computational objects. They are signal interface, operation interface, stream interface, signal interface signature, operation interface signature and stream interface signature.

10.1.2 Channel concepts

10.1.2.1 Channel

A channel is 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.

10.1.2.2 Stub

A stub is 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.

10.1.2.3 Binder

A binder is an engineering object in a channel, which maintains a distributed binding between interacting basic engineering objects.

10.1.2.4 <X> Interceptor

An <X> interceptor is 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.
Protocol object

A protocol object is 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, capsules, or nodes).

Communication domain

A communication domain is a set of protocol objects capable of interworking.

Communication interface

A communication interface is 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.

Identifier concepts

Binding endpoint identifier

A binding endpoint identifier is 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.

Engineering interface reference

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

Engineering interface reference management domain

An engineering interface reference management domain is a set of nodes forming a naming domain for the purpose of assigning engineering interface references.

Engineering interface reference management policy

An engineering interface reference management policy is a set of permissions and prohibitions that govern the federation of engineering interface reference management domains.

Cluster template

A cluster template is an object template for a configuration of objects, with any activity required to instantiate those objects and establish the initial bindings.

Checkpointing concepts

Checkpoint

A checkpoint is 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.

Checkpointing

Checkpointing is to create a checkpoint. Checkpoints can only be created when the engineering object involved satisfies a pre-condition stated in a checkpointing policy.

Cluster checkpoint

A cluster checkpoint is a cluster template containing checkpoints of the basic engineering objects in a cluster.

Deactivation

Deactivation is to checkpoint a cluster, followed by deletion of the cluster.

Cloning

Cloning is to instantiate a cluster from a cluster checkpoint.

Recovery

Recovery is to clone a cluster after cluster failure or deletion.

Reactivation

Reactivation is to clone a cluster following its deactivation.
10.1.4.8 Migration

Migration is to move a cluster to a different capsule.

10.1.5 ODP functions in the context of the engineering viewpoint specifications

Part 3 of RM-ODP describes a set of functions required to support open distributed processing [Part 3 – 11 to 15]. They are grouped in four main categories:

- Management functions: node management function, object management function, cluster management function, and capsule management function;
- Coordination functions: event notification function, checkpointing and recovery function, deactivation and reactivation function, group function, replication function, migration function, engineering interface reference tracking function, transaction function and ACID transaction function;
- Repository functions: storage function, information organization function, relocation function, type repository function, and trading function;
- Security functions: access control function, security audit function, authentication function, integrity function, confidentiality function, non-repudiation function, and key management function;

This clause is only concerned with expressing the engineering specification of these ODP functions.

NOTE – Part 3 is not explicit about the detailed specification of these functions, neither does it explain how the specifications for individual functions can be combined to form specifications for components of ODP systems. Only two of these functions, the Type Repository and the Trading Function, are further refined and more extensively described. "Rec. ITU-T X.960 | ISO/IEC 14769 – Type Repository Function” and “Rec ITU-T X.950 | ISO/IEC 13235 – ODP Trading Function” contain their complete specifications.

10.1.6 Summary of the engineering language metamodel

The diagrams below (Figures 27 to 34) illustrate the concepts of the engineering language and the relationships between them. The descriptions of the concepts have been given above. The descriptions of the relationships between the concepts are included in the description of the concepts.

10.1.6.1 Engineering Objects

![Diagram of Engineering Objects]

Figure 27 – Engineering objects
The following restrictions apply to the elements of the diagram shown in Figure 28:

- All interfaces associated with a **signal interface signature** are signal interfaces:
  
  **context** SignalInterface **inv** SignalInterfaceSignature:
  
  self.specifier.oclIsTypeOf(SignalInterfaceSignature)

- All interfaces associated with an **operation interface signature** are operation interfaces:
  
  **context** OperationInterface **inv** OperationInterfaceSignature:
  
  self.specifier.oclIsTypeOf(OperationInterfaceSignature)

- All interfaces associated with a **stream interface signature** are stream interfaces:
  
  **context** StreamInterface **inv** StreamInterfaceSignature:
  
  self.specifier.oclIsTypeOf(StreamInterfaceSignature)

### 10.1.6.2 Node structure

The node structure is about structuring of a node with nucleus, capsule, cluster and various engineering objects.

![Diagram of node structure](image)

**BEO- Basic Engineering Object**

### Figure 29 – Engineering language – basic concepts

The following constraints apply to the elements of the engineering language shown in Figure 29:

- In order for two **basic engineering objects (BEOs)** to be locally bound to each other, they must reside in the same cluster:
context BEO inv SameCluster:
    self.locallyBoundObject->forAll (obj | obj.cluster = self.cluster)
– A BEO binds to the node management interface provided by the Nucleus associated with the Node that contains the Capsule that contains the Cluster that contains the BEO:
    context BEO inv NodeManagerDerivationRule:
        self.nodeManager = self.cluster.capsule.node.manager
– The engineering object's node manager should be the same as the node manager associated with the node that contains the Capsule that contains the engineering object:
    context EngineeringObject inv NodeManagerDerivationRule2:
        self.nodeManager = self.capsule.node.manager
– The Capsule to which a Cluster belongs is the Capsule to which the Cluster's manager belongs:
    context Cluster inv CapsuleDerivationRule: self.capsule = self.manager.capsule
– Derivation Rule: The CapsuleManager to which the ClusterManager is bound is the CapsuleManager of the Capsule that contains the Clusters that the CapsuleManager manages:
    context ClusterManager inv CapsuleManager:
        self.cluster->forAll (c : capsule | c.manager = self.capsuleManager)
– The set of other engineering objects that the Capsule owns and the set of ClusterManagers that the Capsule owns are disjoint:
    context Capsule inv NoOtherEOisClusterManager:
        self.otherEngObject-intersection(self.clusterManager)->isEmpty()
– The set of other engineering objects that the Capsule owns and the set of CapsuleManagers that the Capsule owns are disjoint:
    context Capsule inv NoOtherEOisCapsuleManager:
        not self.otherEngObject->includes(self.manager)

10.1.6.3 Channels

This clause is about model elements that enable communication around channels.

**Figure 30 – Engineering language model – channels**

The following constraints apply to the concepts illustrated in the diagram of Figure 30:
– Each Stub to which a BEO is related must be part of a Channel to which the BEO is related:
    context BEO inv SameChannel:
        self.stub->forAll (stub | self.channel->exists (channel | channel = stub.channel))
– For each Channel to which a BEO is related, the BEO must be related to exactly one Stub that is part of that Channel:
context BEO inv OneStubPerChannel:
   self.channel->forAll (channel | self.stub->select (stub | stub.channel = channel )->size () = 1 )
   – The collection of BEOs that are the end points linked by a Channel is derived by adding to the collection, for each Stub in the Channel, the BEO to which the Stub is related:

context Channel inv EndPointDerivationRule:
   self.endPoint->includesAll(self.stub.bEO) and self.stub.bEO->includesAll(self.endPoint)
   – The BEOs constituting a Channel’s endpoints must each reside in different Clusters:

context Channel inv EndPointsInDifferentClusters:
   self.endPoint->forAll (ep1, ep2 | ep1.cluster <> ep2.cluster)
   – The BEO and Binder to which a Stub is related are parts of the same Channel of which the Stub is a part:

context Stub inv SameChannelStub:
   self.bEO.channel = self.channel and self.binder.channel = self.channel
   – The Stub to which a Binder is related, and the ProtocolObjects to which the Binder is related, are all parts of the same Channel of which the Binder is a part:

context Binder inv SameChannelBinder:
   self.protocolObject->forAll (po | po.channel = self.channel) and self.stub.channel = self.channel
   – The ProtocolObjects for which an Interceptor provides protocol conversion must be part of the same Channel of which the Interceptor is a part:

context Interceptor inv SameChannelInterceptor:
   self.protocolObject->forAll (po | po.channel = self.channel)
   – Any Interceptor to which a ProtocolObject is related and the Binder to which the ProtocolObject is related are part of the same Channel of which the ProtocolObject is a part:

context ProtocolObject inv SameChannelPO:
   self.interceptor->forAll (i | i.channel = self.channel) and self.binder.channel = self.channel
   – In order for two ProtocolObjects to be associated, they must be of the same type:

context ProtocolObject inv SameType:
   self.boundProtocolObject->forAll (po | po.type = self.type)

10.1.6.4 Domains

This clause is about kinds of domains and object membership of domains that make up domains.

![Figure 31 – Domains](image_url)

The following restrictions apply to the model elements depicted in Figure 31:

– All members of a subdomain are members of its parent domain:

context Domain inv SubDomainIsSubSet:
   self.subDomain->forAll (subDomain | self.member->includes(subDomain.member) )

– Controlling objects should be associated to the corresponding domains:

context SecurityDomain inv ControllingObject:
   self.controllingObjectoclIsTypeOf(SecurityAuthority)

context ManagementDomain inv ControllingObject:
   self.controllingObjectoclIsTypeOf(ManagementAuthority)

context AddressingDomain inv ControllingObject:
   self.controllingObjectoclIsTypeOf(AddressingAuthority)
context NamingDomain inv ControllingObject:
    self.controllingObjectoclIsTypeOf(NamingAuthority)

10.1.6.5 Identifiers

This clause is mainly about identity, domain and policy management, with respect to nodes and objects.

![Diagram](image)

Figure 32 – Engineering language model – identifiers

10.1.6.6 Checkpoints

This clause is about checkpoints and checkpointing behaviour.

![Diagram](image)

Figure 33 – Engineering language model – checkpoints

10.1.6.7 ODP functions

Figure 34 shows the ODP functions introduced in [10.1.5].
10.2 UML profile

This clause specifies how the ODP engineering concepts described in the previous clause are expressed in UML in an engineering specification. A brief explanation of the concepts used in the expression of each concept is given, together with a justification of the expression used.

NOTE 1 – In this clause UML expressions are only defined for those concepts for which use has been demonstrated through an example, included in the main body of this Recommendation | International Standard or in its annexes. Where no example has been identified, the concept concerned is mentioned, but no UML expression is offered.

NOTE 2 – Concepts are presented in the order in which they appear in Part 3.

NOTE 3 – The concepts and rules of the engineering language concern definition of mechanisms and functions required to support distributed interaction between objects in an ODP system, something which deals with aspects related to the software architecture of the system (e.g., distribution or replication) and therefore the appropriate UML mechanisms for modelling software architectures are used (components, ports, interfaces).

NOTE 4 – The engineering viewpoint assumes that the specifier selects a certain level of refinement below which the use of the concept of engineering object ceases to be essential; these lower level specification concerns, such as the realization of the behaviour of engineering objects, are outside the scope of the profile described here, and are addressed by other specification techniques and languages, including the direct use of UML concepts and rules. Thus, this profile covers the specification of engineering objects at the level of UML components that interact through their ports, but leaves open to the specifier the way in which the internal realization of such components is specified.

10.2.1 Engineering object templates and types

An engineering object is generally specified in terms of its template, which is expressed by a component stereotyped as «NV_Object».

Figure 34 – Engineering language model – ODP functions
The attribute isIndirectlyInstantiated of the component stereotyped «NV_Object» should be set to false. This attribute constrains the kind of instantiation that applies to a component. If false, the component is instantiated as an addressable instance.

The stereotype has the following attributes:
- deployedNode: String (defines a reference to a node where an engineering object is deployed);
- securityDomain: String (defines a reference of a security domain it may belong to);
- managementDomain: String (defines a reference of a management domain it may belong to).

Where an engineering object is required to represent a specific entity in the UOD, it is expressed by instanceSpecification of a component that is stereotyped as «NV_Object». Basic engineering objects are particular kinds of engineering objects. Therefore, the stereotype «NV_BEO» that expresses such objects, inherits from «NV_Object».

Where there is the need to express an engineering object type, it is also expressed by a component, stereotyped as «NV_Object». The attribute isIndirectlyInstantiated of the component stereotyped «NV_Object» should be set to false.

When a component stereotyped as «NV_Object» expresses an engineering object template, the attribute isAbstract of such a component should be set to false, meaning that the component needs to provide all the information required to instantiate objects.

### 10.2.2 Object types and templates as engineering objects

There are cases where there is the need to model the type or template of an engineering object at the instance level. An example is the case of a generic factory, which is invoked by passing it a representation of a template (which has type template), and responds by instantiating the template and returning a reference to the created object. To indicate that an object is derived from a given template, we need to represent both the template object and the instantiated object in the model. Likewise for types, to indicate that an object conforms to a given type, we need to represent both the object and its type in the model.

Both type objects and template objects are engineering objects, and therefore are expressed by components, that express its type or template. To distinguish them from other engineering objects, such components are stereotyped «NV_TypeObject» or «NV_TemplateObject», respectively. Both stereotypes inherit from «NV_Object».

The relationship between an engineering object and the object that represents its template, or the objects that represent its types, can be expressed as an attribute of the class that specifies the engineering object.

For example, in some specifications, such as in the ODP trading function specification, there is the need to specify the type of a service, so a trader can locate objects implementing such a service. The diagram shown in Figure 35 represents the specification of a engineering object, PrintService, and of its type, APrintServiceType, expressed so that type can be manipulated by engineering operations.

![Figure 35](image.png)

Figure 35 – An explicit representation of the type of an engineering object so that the object can access its type

### 10.2.3 Cluster

A cluster is expressed by an instanceSpecification of a component, stereotyped as «NV_Cluster». The component stereotyped as «NV_Cluster» expresses the cluster type or template. It includes a configuration of basic engineering objects and has bindings to required channels for communication.

### 10.2.4 Cluster manager

A cluster manager is expressed by an instanceSpecification of a component, stereotyped as «NV_ClusterManager». The component stereotyped as «NV_ClusterManager» expresses the cluster manager type or template.

### 10.2.5 Capsule

A capsule is expressed by an instanceSpecification of a component, stereotyped as «NV_Capsule». The component stereotyped as «NV_Capsule» expresses the capsule type or template.
ISO/IEC 19793:2015 (E)

10.2.6 Capsule manager
A capsule manager is expressed by an instanceSpecification of a component, stereotyped as «NV_CapsuleManager». The component stereotyped as «NV_CapsuleManager» expresses the capsule manager type or template.

10.2.7 Nucleus
A nucleus is expressed by an instanceSpecification of a component, stereotyped as «NV_Nucleus». The component stereotyped as «NV_Nucleus» expresses the nucleus type or template.

10.2.8 Node
A node is expressed by an instanceSpecification of a component, stereotyped as «NV_Node». The component stereotyped as «NV_Node» expresses the node type or template.

10.2.9 Channel
A channel is expressed by an instanceSpecification of a component, stereotyped as «NV_Channel». The component stereotyped as «NV_Channel» expresses the channel type or template. It consists of stubs, binders, protocol objects, and possibly <X> interceptors.

10.2.10 Stub
A stub is expressed by an instanceSpecification of a component, stereotyped as «NV_Stub». The component stereotyped as «NV_Stub» expresses the stub type or template.

10.2.11 Binder
A binder is expressed by an instanceSpecification of a component, stereotyped as «NV_Binder». The component stereotyped as «NV_Binder» expresses the binder type or template.

10.2.12 <X> Interceptor
An interceptor is expressed by an instanceSpecification of a component, stereotyped as «NV_Interceptor». The component stereotyped as «NV_Interceptor» expresses the interceptor type or template.

10.2.13 Protocol object
A protocol object is expressed by an instanceSpecification of a component, stereotyped as «NV_ProtocolObject». The component stereotyped as «NV_ProtocolObject» expresses the protocol object type or template.

10.2.14 Communication domain
A communication domain is expressed by a package, stereotyped as «NV_CommunicationDomain».

10.2.15 Engineering interfaces

10.2.15.1 Communication interface
A communication interface is expressed by a port stereotyped as «NV_CommunicationInterface», through which a protocol object is associated with other protocol objects or interceptors for a communication.

10.2.15.2 Operation interface
An operation interface is expressed by a port stereotyped as «NV_OperationInterface», through which a basic engineering object is associated with a channel or with another basic engineering object.

10.2.15.3 Stream interface
A stream interface is expressed by a port stereotyped as «NV_StreamInterface», through which a basic engineering object is associated with a channel or with another basic engineering object.

10.2.15.4 Signal interface
A signal interface is expressed by a port stereotyped as «NV_SignalInterface», through which a basic engineering object is associated with a channel or with another basic engineering object.

10.2.15.5 Engineering interface signature
An engineering interface signature is expressed by an interface, stereotyped «NV_SignalInterfaceSignature», «NV_OperationInterfaceSignature» or «NV_StreamInterfaceSignature» depending on the type of interface signature (signal, operation or stream).
10.2.16 Binding endpoint identifier

A binding endpoint identifier is expressed by a valueSpecification.

10.2.17 Engineering interface reference

An engineering interface reference is expressed by a class.

10.2.18 Engineering interface reference management domain

An engineering interface reference management domain is expressed by a package, stereotyped as «NV_InterfaceReferenceManagementDomain».

10.2.19 Engineering interface reference management policy

An engineering interface reference management policy is expressed by a constraint, stereotyped as «NV_InterfaceReferenceManagementPolicy».

10.2.20 Checkpoint

A checkpoint is expressed by an instanceSpecification of a component, stereotyped as «NV_Checkpoint». The instanceSpecification of a component expresses a checkpointed object's states at the time of checkpointing.

10.2.21 Checkpointing

A checkpointing is expressed by an activity, UML operation, and UML action stereotyped as «NV_Checkpointing».

10.2.22 Cluster checkpoint

A cluster checkpoint is expressed by an instanceSpecification of a component, stereotyped as «NV_ClusterCheckpoint». The instanceSpecification of a component expresses a checkpointed cluster's state at the time of checkpointing.

10.2.23 Deactivation

A deactivation is expressed by an activity, an operation, or an action stereotyped as «NV_Deactivation».

10.2.24 Cloning

A cloning is expressed by an activity, an operation, or an action stereotyped as «NV_Cloning».

10.2.25 Recovery

A recovery is expressed by an activity, an operation, or an action stereotyped as «NV_Recovery».

10.2.26 Reactivation

A reactivation is expressed by an activity, an operation, or an action stereotyped as «NV_Reactivation».

10.2.27 Migration

A migration is expressed by an activity, an operation, or an action stereotyped as «NV_Migration». A migration, as an ODP function, can also be expressed by an interface (see [10.2.28]).

10.2.28 ODP functions

The ODP functions described in [10.1.5] are expressed by interfaces, stereotyped as «NV_X», where X is the name of the function.

More precisely, the following stereotypes extend the UML metaclass interface to express the corresponding ODP function:

«NV_NonRepudiation» and «NV_KeyManagement».
10.2.29 Summary of the UML extensions for the engineering language

The engineering language profile (NV_Profile) specifies how the engineering viewpoint modelling concepts relate to, and are expressed in, standard UML using stereotypes, tag definitions, and constraints.

Figure 36 shows diagrammatic representations of this profile.
Figure 36 – Graphical representation of the engineering language profile (using the UML notation)

NOTE 1 – In the diagrams above, infrastructure mechanisms are not well represented using UML. It may be necessary to introduce roles for standard functional objects, like trader in the ODP Trading Function standard and recovery manager for recovery function, to cover these mechanisms as well as the ODP functions.

NOTE 2 – Not all management functions are shown in the above figure, e.g., thread management for nucleus.
10.3 Engineering specification structure (in UML terms)

An engineering specification defines the infrastructure required to support the functional distribution of an ODP system. This includes:

- 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).

NOTE – Some ODP functions have been standardized, others have been defined only in outline. Where a suitable definition exists, it can be brought into the engineering specification.

An engineering specification models a system in terms of:

- a configuration of engineering objects, structured as clusters, capsules and nodes (that will be expressed with UML component diagrams, including instanceSpecification of component for capsule, clusters, basic engineering objects, capsule manager, cluster manager, and nucleus);
- the activities that occur within those engineering objects (that will be expressed with UML activity diagrams);
- the interactions of those engineering objects (that will be expressed with UML sequence diagrams).

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

- channel rules [Part 3 – 8.2.1], interface reference rules [Part 3 – 8.2.2], distributed binding rules [Part 3 – 8.2.3] and relocation rules [Part 3 – 8.2.4] for the provision of distribution transparent interaction among engineering objects;
- cluster rules [Part 3 – 8.2.5], capsule rules [Part 3 – 8.2.6] and node rules [Part 3 – 8.2.7] governing the configuration of engineering objects;
- failure rules [Part 3 – 8.2.9].

Those rules will be expressed with UML or OCL constraints for relevant elements.

All the elements expressing the engineering specification will be defined within a model, stereotyped «Engineering_Spec». Such a model contains packages that express:

- the structure of a node, including nucleus, capsules, capsule managers, clusters, cluster managers, stub, binder, protocol objects, interceptors, and basic engineering objects, with a component diagram;
- channels, with component diagrams and a package;
- domains, with a package;
- interactions among those engineering objects, with activity diagrams, stateMachines and interaction diagrams.

10.4 Viewpoint correspondences for the engineering language

10.4.1 Contents of this clause

This clause describes the correspondence concepts for the engineering language, but not how they are expressed in UML. The latter is covered in Clause 12.

10.4.2 Engineering and computational viewpoint specification correspondences

NOTE – The correspondence between an engineering specification and a computational specification can be derived from [9.4.4].

10.4.3 Engineering and technology viewpoint specification correspondences

Each engineering object corresponds to a set of one or more technology objects. The correspondence and implementable standards for each technology object are dependent on the choice of technology.

The engineering viewpoint specification does not have any correspondences to implementation.

Engineering objects and their interfaces correspond to technology objects and their interfaces, and thus will become basic information source for testing in the technology viewpoint.
11 Technology specification

11.1 Modelling concepts

A technology specification uses the RM-ODP technology language. The modelling concepts and the structuring rules of the technology language are defined in [Part 3 – 9]. They are summarized in this clause. Except where otherwise stated, in case of conflict between the explanations therein and the text in Part 3, the latter document should be followed.

The set of diagrams at the end of this clause (i.e., at [11.1.4]) summarizes a metamodel for the technology language.

11.1.1 Implementable standard

A template for a technology object.

11.1.2 Implementation

A process of instantiation whose validity can be subject to test.

11.1.3 Implementation eXtra Information for Test (IXIT)

Provides extra information for testing.

11.1.4 Summary of the technology language metamodel

Figure 37 below illustrates the concepts of the technology language and the relationships between them. The descriptions of the concepts have been given above. The descriptions of the relationships between the concepts are included in the description of the concepts.

![Figure 37 – Model of the technology language](image)

11.2 UML profile

This clause specifies how the ODP technology concepts described in the previous clause are expressed in UML in a technology specification. A brief explanation of the UML concepts used in the expression of each concept is given, together with a justification of the expression used.

NOTE – In this clause UML expressions are only defined for those concepts for which use has been demonstrated through an example, included in the main body of this Recommendation | International Standard or in its annexes. Where no example has been identified, the concept concerned is mentioned, but no UML expression is offered.

11.2.1 Technology object

A technology object is generally specified in terms of its type, which is expressed by an artefact or a node, stereotyped as «TV_Object». Technology object types can be used to characterize the different kinds of technology objects that are used in a technology specification (such as PCs, application servers, LANs, WANs, etc.).

Where a technology object is required to represent a specific entity in the UOD, it is expressed by instanceSpecification of an artefact or a node that is stereotyped as «TV_Object».

11.2.2 Object types and templates as technology objects

There are cases where there is the need to model the type or template of a technology object at the instance level. An example is the case of a technology object, which needs to know the types of the objects it interacts with in order to fix the appropriate QoS constraints that rule their interactions.

Both type objects and template objects are technology objects, and therefore are expressed by nodes or artefacts, that express its type or template. To distinguish them from other technology objects, such classes are stereotyped «TV_TypeObject» or «TV_TemplateObject», respectively. Both stereotypes inherit from «TV_Object».

The relationship between a technology object and the object that represents its template, or the objects that represent its types can be expressed as an attribute of the node or artefact that specifies the technology object.
11.2.3 Implementable standard

An implementable standard is expressed by a component, stereotyped as «TV_ImplementableStandard».

11.2.4 Implementation

An implementation is expressed by an activity, stereotyped as «TV_Implementation».

11.2.5 IXIT

An IXIT is expressed by a comment, stereotyped as «TV_IKIT».

11.2.6 Summary of the UML extensions for the technology language

The technology language profile (TV_Profile) specifies how the engineering viewpoint modelling concepts relate to, and are expressed in, standard UML using stereotypes, tag definitions, and constraints.

Figure 38 shows diagrammatic representations of this profile. See clause [A.5] for a detailed specification of the stereotypes described here.

Figure 38 – Graphical representation of the technology language profile (using the UML notation)

The following restrictions apply to the elements depicted in Figure 38. They are derived from the corresponding constraints on the elements shown in Figure 37 and on their relationships:

- every technology object type is associated with at least one implementable standard.
- every implementation standard is associated with, or is implemented as, one or more technology objects.
- every implementation is associated with, or produces, one or more technology objects.

11.3 Technology specification structure (in UML terms)

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

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

NOTE 1 – Links between deployment boxes may be used to model physical communication lines (e.g., to model multiple lines for redundancies).

NOTE 2 – A network (e.g., the Internet) may be modelled with a deployment box connected with other deployment boxes.

A technology specification states:

- how the specifications for an ODP system are implemented, which may be modelled with component instances and the relationships between them with text explanation;
– a taxonomy of such specifications, which may be provided with names of implementable standards
described in stereotyped comments attached to a deployment diagram including a component instance
diagram;
– information required from implementers to support testing, which may be specified with a stereotyped
comment describing IXIT.

NOTE – Software architecture styles, such as SOA, MVC and N-tier, are considered mainly in the engineering viewpoint, since
they are closely related to the distribution strategy.

All the elements expressing the technology specification will be contained within a model, stereotyped
«Technology_Spec». Such a model contains packages that express:
– the structure of a node instance, including node instances within a node instance, artefacts, and networks,
using a deployment diagram; and
– communication links among nodes, using a deployment diagram.

11.4 Viewpoint correspondences for the technology language

This clause describes the correspondence concepts for the technology language, but not how they are expressed in UML.
The latter is covered in clause 12.

A set of one or more technology objects correspond to an engineering object, and they implement specified functionality
in corresponding engineering object in technology specific way.

NOTE 1 – The choice of specific technology in the technology viewpoint may constrain the possible architecture or platform styles
(or patterns) and deployment patterns in the engineering viewpoint specification.

NOTE 2 – A wide variety of factors, including procurement policy, extra-functional requirements etc., may influence the choice
of technology, and therefore the technology specification.

12 Correspondences specification

12.1 Modelling concepts

A correspondences specification is composed of a set of correspondence specifications.

A complete specification includes six correspondence specifications:
– between the enterprise specification and the information specification;
– between the enterprise specification and the computational specification;
– between the enterprise specification and the engineering specification;
– between the computational specification and the information specification;
– between the computational specification and the engineering specification; and
– between the engineering specification and the technology specification.

12.1.1 Correspondence specification

A correspondence specification is composed of a set of correspondence rules and a set of correspondence links. It
describes consistency relationships between terms belonging to two specifications based on different viewpoints.

When a correspondence rule and a correspondence link are related, this means that the constraint in the correspondence
rule must be enforced by the set of terms referenced by the correspondence link.

12.1.2 Correspondence rule

A correspondence rule is expressed by a constraint that must be enforced by a set of terms belonging to two specifications
from different viewpoints.

A correspondence rule may be:
– a correspondence statement as defined in clauses 7.4, 8.4, 9.4, 10.4, or 11.4;
– some other consistency rule resulting from a design choice.

12.1.3 Correspondence link

A correspondence link is established between two specifications from different viewpoints. Each end of the
correspondence link is called a correspondence endpoint.
12.1.4 Correspondence endpoint
A correspondence endpoint is composed of terms involved in the consistency relationship.

12.1.5 Term
A term is a linguistic construct which may be used to refer to an entity. The reference may be to any kind of entity including a model of an entity or another linguistic construct.

NOTE – From the definition extracted from [Part 2 – 5], an ODP term is analogous to a UML element.

12.1.6 Summary of the Correspondences metamodel
The modelling concepts introduced for a correspondences specification are summarized in Figure 39.

12.2 UML profile
This clause specifies how the modelling concepts for correspondences specification are expressed in UML.

12.2.1 Correspondence specification
A correspondence specification is expressed by a package, stereotyped as «CorrespondenceSpecification».

The relationship between a correspondence specification and the models expressing the viewpoints involved in the correspondence specification is expressed by a usage dependency, stereotyped as «CorrespondingSpecification». There are exactly two such dependencies for each correspondence specification.

12.2.2 Correspondence rule
A correspondence rule is expressed by a constraint, stereotyped as «CorrespondenceRule».

NOTE – The constraints expressing constraints defined in ODP standards may be defined outside the package expressing the correspondence specification to enable reuse among multiple specifications.

12.2.3 Correspondence link
A correspondence link is expressed either by a class, stereotyped as «CorrespondenceLink» or by a dependency stereotyped as «CorrespondenceLink».

It may be constrained by a constraint expressing the applicable correspondence rule.

The stereotype «CorrespondenceLink» has two tag definitions, named endPoint1 and endPoint2, which specify the two correspondence endpoints of the correspondence link (see [12.2.4]).

A constraint stereotyped as «CorrespondenceRule» is only applied to a class stereotyped as «CorrespondenceLink».

12.2.4 Correspondence endpoint
A correspondence endpoint is expressed by a tag definition of stereotype «CorrespondenceLink», which gives references to the elements expressing the terms involved in the correspondence relationship. Thus, this tag definition is typed by an element (see [12.2.5]) and has a multiplicity of 1..*.

NOTE – As many elements expressing ODP concepts cannot be used directly in a class diagram, tag definitions are used to allow indirect reference to those concepts.
12.2.5 Summary of the UML extensions for correspondences specification

Figure 40 shows a graphical representation of the UML profile for correspondences specifications.

13 Modelling conformance in ODP system specifications

13.1 Modelling conformance concepts

Conformance relates an implementation to a specification. Any proposition that is true in the specification must be true for its implementation. A conformance statement is a statement that identifies conformance points of a specification and the behaviour which must be satisfied at these points. Conformance statements will only occur in specifications which are intended to constrain some feature of a real implementation, so that there exists, in principle, the possibility of testing.

The RM-ODP [Part 2 – 15] identifies certain reference points in the architecture as potentially declarable as conformance points in specifications. That is, as points at which conformance may be tested and which will, therefore, need to be accessible for testing. However, the requirement that a particular reference point be considered a conformance point must be stated explicitly in the conformance statement of the specification concerned, together with the conformance criteria that should be satisfied at this point.

13.2 UML profile

Reference points are identified in the UML expression of an ODP specification by the use of the stereotype «ODP_ReferencePoint» (which extends a metaclass element) on the elements that express them. Conformance statements are expressed by comments stereotyped «ODP_ConformanceStatement», attached to the elements (stereotyped «ODP_ReferencePoint») that express the corresponding reference points. These comments describe the conformance criteria that should be satisfied at the reference point. Therefore, conformance criteria are those elements stereotyped «ODP_ReferencePoint», which have also attached a «ODP_ConformanceStatement» comment. It is possible to attach multiple «ODP_ConformanceStatement» comments to one element stereotyped «ODP_ReferencePoint», thus declaring several conformance criteria at the same reference point.

Figure 41 shows a diagrammatic representation of this UML profile.
14 Conformance and compliance to this Recommendation | International Standard

14.1 Conformance

Levels of conformance may vary. At the least, implementations of tools claiming conformance to this Recommendation | International Standard must support:

- one or more of the UML profiles for viewpoint languages defined in clauses 7 to 11; further conformance may be claimed if the tool concerned supports policing or enforcing of the constraints specified for the stereotypes defined in the relative profiles;
- specification of the correspondences, as defined in clause 12, between ODP modelling elements in the viewpoint models supported by the tool, as defined in clauses 7.4, 8.4, 9.4, 10.4, and 11.4;
- the structuring style for ODP system specifications defined in clause 6.6.

NOTE – Claims of conformance to the metamodels alone are outside the scope of this Recommendation | International Standard.

14.2 Compliance

Specifications claiming compliance with this Recommendation | International Standard shall:

- use the structuring style defined in clause 6.6;
- use the UML profiles for the viewpoint languages defined in clauses 7 to 11 of this Recommendation | International Standard to express the concepts and structuring rules for which they are defined;
- specify the correspondences between ODP modelling elements in different viewpoint models using the tracing mechanisms defined in clauses 7.4, 8.4, 9.4, 10.4, and 11.4;
- specify conformance using the UML profile for conformance defined in clause 13.

Compliance to this Recommendation | International Standard does not preclude the use in a specification of concepts and structuring rules in Part 2 and Part 3, and in the Enterprise Language, that are not covered by this Recommendation | International Standard and the definitions of corresponding UML profile elements.
Annex A

An example of ODP specifications using UML

(This annex does not form an integral part of this Recommendation | International Standard.)

The following example illustrates the results of use of UML for expressing ODP system specifications. This annex is not normative.

A.1 The Templeman Library system

A.1.1 Introduction

This is an example of an ODP specification of a library system, using UML. The example is about the computerized system that supports the operations of a university library, in particular those related to the borrowing process of the library items. The system should keep track of the items of the university library, its borrowers, and their outstanding loans. The library system will be used by the library staff (librarian and assistants) to help them record loans, returns, etc. The borrowers will not interact directly with the library system.

NOTE – In the following, the library system (or the system, for short) will refer to the computerized system that supports the library operations, while the library will refer to the business itself, i.e., the environment of the system.

Instead of a general and abstract library, this example is based on the regulations that rule the borrowing process defined at the Templeman Library at the University of Kent at Canterbury, a library that has been previously used by different authors for illustrating some of the ODP concepts.

A.1.2 Rules of operation of the library

The basic rules that govern the borrowing process of that library are as follows:

1. Borrowing rights are given to all academic staff, and to postgraduate and undergraduate students of the University;
2. Library books and periodicals can be borrowed;
3. The librarian may temporarily withhold the circulation of library items, or dispose them when they are no longer appropriate for loan;
4. For requesting a loan, the borrower must hand the books or periodicals to a library assistant;
5. There are prescribed periods of loan and limits on the number of items allowed on loan to a borrower at any one time. These rules may vary from time to time, the librarian being responsible for setting the chosen policy. Typical limits are detailed below:
   - undergraduates may borrow eight books. They may not borrow periodicals. Books may be borrowed for four weeks;
   - postgraduates may borrow 16 books or periodicals. Periodicals may be borrowed for one week. Books may be borrowed for one month;
   - teaching staff may borrow 24 books or periodicals. Periodicals may be borrowed for one week. Books may be borrowed for up to one year;
6. Items borrowed must be returned by the due day and time which is specified when the item is borrowed;
7. Borrowers who fail to return an item when it is due will become liable to a charge at the rates prescribed until the book or periodical is returned to the library, and may have borrowing rights suspended;
8. Borrowers returning items must hand them in to an assistant at the main loan desk. Any charges due on overdue items must be paid at this time;
9. Failure to pay charges may result in suspension by the librarian of borrowing facilities

In the following, we will refer to these rules as the “textual regulations” of the library system. They will be the starting point for the ODP specifications below.

It is important to note that the textual regulations above leave many details of the system unspecified, such as when or how a borrower suspension is lifted by the librarian, or the precise information that needs to be kept in the system for each user and library item. The specification process followed here will help uncover such missing details progressively, so the appropriate stakeholders of the system can determine them by making the corresponding decisions.
A.1.3 Expressing the library system specification in UML

This Annex describes a specification of the different ODP viewpoints of such a system, using UML. For each of the viewpoints, this specification uses the corresponding languages defined in RM-ODP and, where appropriate, expresses the languages in terms of the UML notation.

The UML specifications of the ODP system will consist of one top-level model stereotyped «ODP_SystemSpec» composed of five models with the specifications of the five ODP viewpoints (Figure A.1), together with the models that describe the correspondences between them. These models will be described in the following clauses.

Figure A.1 – UML specification of the ODP system

A.2 Enterprise specification in UML

A.2.1 Basic enterprise concepts

The enterprise viewpoint is an abstraction of the system that focuses on the purpose (i.e., objective), scope and policies for that system and its environment. It describes the business requirements and how to meet them, but without having to worry about other system considerations, such as particular details of its software architecture, its computational processes, or the technology used to implement it.

Four key concepts of the enterprise language are: system, scope, enterprise specification, and field of application. In the first place, the system to be specified is a computerized system that supports the operations of a university library, in particular those related to the borrowing process of the library items. This system has a name “the Templeman Library System” (or “TLS” for short).

The scope of the TLS system describes its expected behaviour, i.e., the way it is supposed to work and interact with its environment in the business context. In the enterprise language, the scope of the system is modelled as the set of roles it fulfils.

In UML, the enterprise specification of the TLS system is expressed by one model, stereotyped «Enterprise_Spec», which is shown in Figure A.2, and whose contents are further detailed in this clause.

In the figures that follow, to improve the clarity of the diagrams, the icons shown in Table A.1 have been used to represent instances of the corresponding stereotypes.
The ODP Enterprise Language specification does not prescribe any particular method for building the enterprise specification of a system, as the approach taken will depend very much on the system being specified, the business that it will support, and the constraints that arise from the environment in which the system will operate. For this example, the following process has been followed:

1. Identify the communities, with which the system is involved, and their objectives;
2. Define the behaviour required to fulfil the objectives of the communities. This may be in the form of processes, their corresponding actions, and the participant roles in them. Objects may participate in actions as actors (if they participate in or perform the action), artefacts (if they are referenced in the action), and resources (if they are essential to the action and may become unavailable or used up);
3. In addition, depending on the modelling objectives, behaviour may be modelled in the form of interactions between objects fulfilling roles. This approach is appropriate when it is required to model a behaviour in detail;
4. Identify the enterprise objects in each community, (either as typical instances of a type, or as unique instances) and how they fill the roles;
5. Identify the policies that govern the behaviour;
6. Identify any behaviours that may change the rules that govern the system, and the policies that govern such behaviours (changes in the structure, behaviour or policies of a community can occur only if the specification includes the behaviour that can cause those changes);
7. Identify the actions that involve accountability of the different parties, and the possible delegations;
8. Identify any behaviour that may change the structure or the members of each community during its lifetime, and the policies that govern such behaviour.

Of course, the order of these activities needs not necessarily be linear, and nor will all activities be appropriate for all modelling situations.

A.2.2 Communities

As shown in Figure A.2, the enterprise specification of the library example contains two communities (the Library and the Academic Community). Each of these is specified in a package, stereotyped as «EV_CommunityContract», containing a component, stereotyped as «EV_Community» (as well as other elements specifying other aspects of the community). Each of these components has a dependency, stereotyped as «EV_RefinesAsCommunity», from the relevant class stereotyped as «EV_CommunityObject» (Library and Academic Community) which expresses the community object that models the community when considered as a single object. (Note that the Academic Community is included only to illustrate the principle that, at the top level, there may be more than one community. The Academic Community is not further detailed in this example.) For convenience, these community objects are included in a package named as Enterprise Objects (global), which contains those enterprise objects that model entities whose scope is wider than the library itself. Examples of such enterprise objects are Person and the University admin system, with which the Library System has to interact.

Figure A.2 – UML Enterprise specification of the Library system

The field of application of the enterprise specification describes the properties that the environment of the ODP system must have for the specification to be used. It is expressed in a tagged value of the package, stereotyped as «Enterprise_Spec» that contains the enterprise specification of the system.

A community is a configuration of objects modelling a collection of entities (e.g., human beings, information processing systems, resources of various kinds, and collections of these) that are subject to some implicit or explicit contract governing their collective behaviour, and that has been formed for a particular objective.

The package containing the specification of the Library community is stereotyped «EV_CommunityContract», and contains the component, Library, stereotyped as «EV_Community» that expresses that community and owns the processes of the community, three packages containing, respectively, the roles in the community, the set of enterprise objects specific to the community (Library Enterprise Objects, which model its structure), and the policies for the community, and one class (stereotyped «EV_Objective») which has a tagged value that expresses the community objective as follows: “To allow the use, by authorized borrowers, of the varying collection of library items, as fairly and efficiently as possible”. This class has an association, stereotyped as «EV_ObjectiveOf», with the component expressing the Library community.
A.2.3 Processes

Processes specify behaviour in terms of (partially ordered) sets of steps, and are related to achieving some particular subobjective within the community. Steps are abstractions of actions, which may hide some of the objects participating in the actions.

![Figure A.3 – UML specification of the Library community](image)

A.2.3.1 Borrow item process

In Figure A.5, the behaviour of the Library system role in the Borrow item process is defined by the actions in the activityPartition for the Library system role. The complete behaviour of the Library system role is the composition of its behaviours in all of the processes in which it is involved.

The process starts with a Borrower (a role filled by a Library member) performing the step State loan requirement. (The exact mechanism and procedures for doing this are not stated at this time, but it could be as simple as the borrower taking the item concerned to a desk for processing by the library assistant.) This step implies that a Loan (enterprise object) has come into existence, and this fact is modelled by an artefact of Loan expressed as an objectFlow, named in the model borrower requests which has the type Loan.
Note that in this example, artefacts have been further detailed by identifying for each, a state of the enterprise object that the artefact represents a usage of. This is not mandated by the enterprise language but allows the use of a UML feature to build an important bridge to the information specification. The resultant stateMachine of the class that expresses the enterprise object can form the basis for expressing a dynamic schema for the associated information object type. For details of the stateMachine for the Loan enterprise object, see Figure A.13. In this case the artefact represents the enterprise object Loan in the state Requested by borrower.

The Assistant (a role filled by a Person who is of type Library Staff), next performs the step Check request, which references, as an artefact, the enterprise object Loan in the state Requested by assistant. Again, this step, being part of the human behaviour associated with the system’s operation, is not further detailed in this model, which is directed towards the system's specification. In a real life situation, such behaviour would need to be documented, and the model may be a good place to do it.

The remainder of Figure A.5 is largely self-explanatory and is not detailed further in textual form. It should be noted that the states of the enterprise object Loan, identified in the various artefacts, are not exhaustive. Other states, see Figure A.15, may also be discovered from considerations of other behaviour. For example, the sub-states of Loan extant, Valid and Overdue, are discovered from consideration of Fining interactions or Fining processes.

A.2.3.2 Add member process

As a further example of process modelling, Figure A.6 shows the top-level process involved when a prospective new member of the library applies to join. The diagram is largely self-explanatory, but it can be seen that through the use of artefacts a number of states of the Library member enterprise object (but by no means all of them), have been identified.

In this process, the Library system has 3 steps, which are detailed in two different ways. The simple case is for the two steps Create member and Refuse new member. Each of these is detailed by an opaqueBehavior owned by the role Library system see A.2.4.
The step Validate member is refined as a process, also named Validate member, which is owned by the process Add member. (It is unfortunate that there is no visual means of indicating which of the different detailing approaches is used; in the model, however, querying the model element concerned will show whether the detail of a step is provided by an activity expressing a process, a stateMachine expressing a set of actions, or an opaqueBehavior modelling directly the details of the behaviour required.) The step is performed, at the high level, by the Library system (enterprise object and role). In this example, for the purposes of illustration, it has been assumed that because appropriate technology is available, the actual check on validity of an application will be made by the agent best placed to do it, namely an enterprise object known as the University admin system (filling the role University admin system), and that a direct link will be made in order to check an applicant’s credentials. The details of this subprocess, as well as the required states of the enterprise object Library member, are shown in Figure A.7. As in other activity diagrams, some of the steps performed by either the Library system role or the University admin system role are detailed elsewhere in the model, as indicated by the small forked symbol under the name of the step. In each case they are detailed by an opaqueBehavior, owned by the relevant role.
A.2.4 Roles

From the textual description of the library (and, in real life more importantly, from discussions, interviews and workshops with stakeholders) we can identify several roles in the Library community, in particular borrowers with various privileges, librarians, library assistants, and the computerized system that supports the library operations (Library System). Figure A.8 shows these Library roles within the package that contains the specification of the community, each with a realization link to the component that expresses the community.
The behaviour identified by a role is expressed by the set of behavioralFeatures of the class that expresses the role. For example, the (partial) list of behavioralFeatures of the role Library system is specified in three opaqueBehaviors, which are Create member, Prepare validation request, and Refuse new member, and one stateMachine, Loan validation.

A.2.5 Interactions

Behaviour can also be modelled in terms of interactions between roles in a community. This is normally appropriate for modelling the detail of a particular interaction and the associated behaviour of the roles concerned where a process model lacks semantic power. In this example, we detail an interaction between the Assistant role and the Library System role, and the associated behaviour of the Library System role since we are concerned to specify in detail the behaviour which the Library System is required to provide.

The relationships between the classes expressing the interaction involved in the behaviour of requesting a loan, and those classes expressing the roles involved in this interaction is shown in Figure A.9. There is one interaction in this case: Process loan in which Assistant and Library System are involved. The relationship is expressed with an association, stereotyped as «EV_InteractionInitiator» or «EV_InteractionResponder» as appropriate. Note that, with delegated authority from the Librarian role, the Assistant role is performing an Accountable action, in performing its actions as part of the Process loan interaction.

Figure A.9 – Process loan interaction

Figure A.9 also shows that the Process loan interaction is initiated by the role Assistant and responded to by the role Library System and involves, through associations which are each stereotyped as «EV_ArtefactReference», three signals, each stereotyped as «EV_Artefact», expressing artefact roles of the Loan enterprise object: loan: request by assistant, loan: authorized and loan: disqualification respectively.

Figure A.10 shows the stateMachine for the behaviour defined for the role Library system, in the Process loan interaction, with the role Assistant.

Rec. ITU-T X.906 (10/2014) 77
This example has defined the behaviour of the **Library system role** in the **Request Item interaction**. The complete behaviour of the **Library system role** is the composition of its behaviours in all of the interactions in which it is involved (see A.4).

### A.2.6 Enterprise Objects

#### A.2.6.1 Actors

Roles are fulfilled by enterprise objects. The fulfilment of actor roles in a community by enterprise objects is governed by assignment rules. Using UML, the fact that an actor role may be fulfilled by an enterprise object is expressed by an association, stereotyped as «EV_FulfilsRole», between the classes that express the objects and the roles concerned. Assignment rules can be constrained by the policies of the system, in which case there would be links between the roles and elements expressing the policies. Figure A.11 shows the UML expression of the basic (i.e., unconstrained by any policies) assignment rules of the **Library community**.

![Figure A.10 – State diagram for Library system role in the Process loan interaction](image)

**Figure A.10 – State diagram for Library system role in the Process loan interaction**

**Figure A.11 – Actor role fulfilment and assignment rules**

#### A.2.6.2 Artefacts

Enterprise objects may also participate in actions by filling artefact roles. In this example, Loans are enterprise objects that model the relationship that is established between a borrower and an item when she requests the item, and continues for a period from either the loan being refused or the item, having been loaned, being returned. Loans fulfil artefact roles in several actions (from process model, see interaction model, see A.2.5, above). In this case, the actions are loan: authorized, loan: disqualified and loan: requested by assistant.

#### A.2.6.3 Summary of enterprise objects

In summary, the enterprise objects, and the relationships between them, that have roles (either actor or artefact) in the **Library community** are shown in Figure A.12. Note that the list of such items includes enterprise objects that have wider scope than just the **Library community**.
A.2.6.4 Enterprise object states

As noted in A.2.3.1, it is useful to model the states of the enterprise objects, because they may help specifying the corresponding information object types.

Figure A.13 is a state machine for the Loan enterprise object and is compiled from consideration of both the Process model (see A.2.3) and the Interaction model (see A.2.5).

Figure A.13 – States of the Loan enterprise object

In a similar fashion, Figure A.14 is an incomplete diagram representing those of the states of the enterprise object Library member that have been identified from the process models that have been developed, including the Add member process shown in Figure A.6 and Figure A.7. It should be noted that these state diagrams will only be "complete" when all behaviour that the system under consideration is involved in, has been defined.
A.2.7 Policies

A.2.7.1 General

In an enterprise specification the concept, policy, is intended to be used where the desired behaviour of the system may be changed to meet particular circumstances.

The Policies package specifies the community policies, which constrain the structure or the behaviour of the community, or both. Therefore, the elements of that package will constrain the elements of the other two packages in the Library Community package (Behaviour and Library Enterprise Objects).

Providing an independent and modular specification of policies will enable the definition and implementation of some traceability mechanisms, both between and within viewpoints. Within the UML expression of the enterprise specification of a system, we need to be able to list all the elements affected by a given policy, and all the policies that constrain a given element, in case there is a change in the specification's elements or policies. But such independent expression of enterprise policies may also allow the definition of correspondences between these policies and other related elements from different ODP viewpoints (such as information invariant schemata). We expect UML modelling tools to exploit such traceability mechanisms, checking for absences of policies for some of the modelling elements, and also for policy conflicts and inconsistencies at various levels.

In this relatively simple example, the aspects of the system that are most appropriate for use of this concept are in the rules regarding borrowing permissions (see A.1.2 rule (5)).

According to the considerations above, in order to be properly specified, policies need to identify the relevant enterprise elements to which they apply: roles, objects, actions, processes, communities, as well as their relationships. Such elements are precisely those described in the two other packages that form part of the enterprise specification of the system: Enterprise Objects and Behaviour.

A.2.7.2 Expressing ODP policies in UML

In this example we will express policies using the pattern shown in clause 7.1.5 and Figure 10, which corresponds to the elements that comprise the specification of an enterprise policy in the Enterprise Language [E/L – 7.9.2]:

- description: text with the description of the policy in natural language;
- controllingAuthority: an authority that controls the policy (in this case, a role);
- relatedBehaviour: an identified behaviour (i.e., role) that is subject to that authority;
- relatedObjects: optionally, an object or objects that may fulfill the roles involved;
- specificationConstraint: set of constraints on the modelling elements involved;
- affectedBehaviour: the subset of the related behaviour that is required, permitted, forbidden, or authorized.

The behaviours, roles and objects related to a policy specification in UML refer, of course, to the UML elements expressing these behaviours, roles and objects, respectively. Such elements will normally be used as contexts in the constraints that specify the policy. Note that all policy statements are made in a context that defines the elements in the specification to which the policy applies, and have a condition that specifies when the policy can be used. In this sense, OCL can be of real help. Each OCL constraint has a particular context, related to some element in the model. OCL statements can be directly associated to some elements in a diagram, establishing an implicit context by attachment, or
they can form part of a separate piece of specification in which the context of each statement is explicitly established by naming. Rules are expressed as constraints, using a given notation (such as OCL, or a specific policy language).

A.2.7.3 Expressing loan policies in the Templeman Library

Figure A.15 shows the structure of the Policies package.

![Figure A.15 – Structure of the policies package](image)

Details of the Lending Policies are shown in Figures A.16 and A.17, which for illustrative purposes offer both behavioural modelling styles (i.e. with processes and interactions). From this it can be seen that the Lending Limit Policy is set by a process Set lending limit policy (located in the Administrative Processes package), and impacts on the role Library System, when taking part in the process Borrow Item, or the interaction of the same name.

Similarly the Loan Duration policy is set by the interaction Set loan duration policy (located in the Administrative Interactions package), and impacts on the role Library System, when taking part in the process Fine Borrower, or the interaction of the same name.
A.2.8 Accountability

An enterprise specification should also identify those actions that involve accountability of a party, where a party models a natural person or any other entity considered to have some of the rights, powers and duties of a natural person. Principal parties are responsible for the acts of any parties acting as their delegated agents, including their possible commitments, prescriptions, evaluations, declarations, and further delegations.

Accountable parties in a given process or action are expressed in the UML diagram that defines such process or action. The stereotype «EV_Accountable» on an association between an actor and an action indicates the actor that is accountable for the action. Figure A.18 shows an example of the use of such a stereotype, indicating that the Assistant is the accountable party for the Process loan action.
Delegations are expressed in UML by associations between roles in activity diagrams stereotyped «EV_Delegation», showing the principal and agent parties of each delegation. Such associations allow delegated parties to initiate or participate in actions on behalf of their principals. In particular, Figure A.18 specifies that the Librarian can delegate his actions to an Assistant. As previously mentioned, the delegation may convey some information about its duration, conditions, further delegations allowed, etc. Attributes of the «EV_Delegation» stereotype may be used to express such kinds of information.

![Diagram showing delegation between Librarian and Assistant](image)

**Figure A.18 – Example of delegation**

### A.3 Information specification in UML

#### A.3.1 Overview

The information viewpoint is concerned with information modelling. An information specification defines the semantics of information and the semantics of information processing in an ODP system, without having to worry about other system considerations, such as particular details of its implementation, the computational process, or the nature of the distributed architecture to be used. The information specification in this clause defines both the basic concepts for information used in this specification, and the invariant, static and dynamic schemata.

In the figures that follow, to improve the clarity of the diagrams, the icons shown in Table A.2 have been used to represent instances of the corresponding stereotypes.

<table>
<thead>
<tr>
<th>Table A.2 – Information language icons.</th>
</tr>
</thead>
<tbody>
<tr>
<td>«IV_Object»</td>
</tr>
<tr>
<td>«IV_TypeObject»</td>
</tr>
<tr>
<td>«IV_Action»</td>
</tr>
<tr>
<td>«IV_InvariantSchema»</td>
</tr>
<tr>
<td>«IV_StaticSchema»</td>
</tr>
<tr>
<td>«IV_DynamicSchema»</td>
</tr>
</tbody>
</table>

According to [8.4], the UML information specification of the Library system is expressed by one model, stereotyped «Information_Spec», that contains a set of packages that express the invariant, dynamic, and static schemata of the ODP information specification in UML. Figure A.19 shows the information specification of the library system, composed of four main packages. The following subclauses define these packages and their contents.
Figure A.19 – Structure of the information viewpoint specification of the library system (excerpt)

A.3.2 Basic elements

From the textual regulations of the library, and from the objects, roles and artefacts identified in the enterprise specification, several information object types can be identified, namely Borrowers, library Items, Librarians and Library Assistants. They describe the information stored and handled by the Templeman Library system about them. In addition, a Calendar object should model the passage of time, and Loan objects will model the relationships between Borrowers and Items. Figure A.20 shows a class diagram with all the basic object types used in this information specification. UML class Person contains the personal information about the library users, librarians and assistants.

Figure A.20 – Object types of the information viewpoint specification of the library system

The attributes of each class define the information captured by this specification. Please notice that this information specification is built considering the elements of the enterprise specification described in clause A.2. The RM-ODP does not impose any methodology for the definition and use of the five viewpoints. However, for building the UML information viewpoint specifications of this particular example we have used its enterprise specifications. This approach greatly facilitates the definition of the ODP correspondences between the related entities that appear in the different viewpoints, and also simplifies the treatment of consistency among viewpoints. Viewpoint consistency tries to detect and resolve the possibility that different viewpoints may impose contradictory requirements on the same system.

In particular, this information specification incorporates the information kept in the system for each user and library staff (name, address, phone, e-mail), and for each Library item: title, author, ISBN or ISSN, its physical location, and its current status: on-loan, free, withheld (if the circulation of the item has been temporarily withheld), disposed (if the
item has been sold, donated, recycled, or discarded), **missing** (if the item is missing), or **other** (in case the item is in a status not contemplated by any of the previous options).

Information object **LibraryPolicies** contains the library system values associated to the **policies** identified in the Enterprise Viewpoint specification, such as the details about the daily rates to be charged to late-returners and the current loan limits and periods for the different kinds of users.

General and common parameters about the library are modelled by another information object (**Library**). Its attribute **isOpen** stores whether the library is open or not to the public, while its attribute **credit** stores the cumulative credit obtained by collecting the payment of the fines. The **Library** object is a composite information object which includes the information about the current library **Borrowers, Items** and **Loans**. It also gathers the information about the rest of the objects of the system, expressed in terms of associations between this object and the **Librarian, Assistants, LibraryPolicy** and **Calendar** objects.

The classes in Figure A.20 express the ODP information object types of the library system information specification. Please notice that the information specification captures the information handled by the Templeman Library system, and there is no need to represent the computerized system itself (as happened in the enterprise viewpoint specification). This is one notable difference between the enterprise and the rest of the viewpoints. The enterprise viewpoint focuses on the system and its environment (and therefore the system needs to be modelled as one of the enterprise objects in the specification), while the rest of the viewpoints focus on the information, functionality, distribution, and technology of the system itself.

The class diagram in Figure A.20 also expresses constraints on the kinds of objects and the kinds of links that can appear in a valid object configuration of the information specification. Such restrictions on the classes, their attributes, and the multiplicity of the associations specify some invariant schemata of the information specification (see [A.3.3]).

![Figure A.20 - Information object types of the library system](image)

**Figure A.20 – Information object types of the library system**

The information actions of this viewpoint specification are the ones described in Figure A.21. These actions have been identified from the processes and interactions defined in the enterprise viewpoint specification of the system. In the UML information specification, the information actions are expressed using a package that expresses the invariant schema that specifies the action types supported by the information objects of the system.

As information action types, they will all be expressed in this example by signals, which will trigger the state changes in the stateMachines of the objects. These stateMachines will express the dynamic schemata that will describe the state changes caused in the system by such information actions. Those dynamic schemata will be described later in clause A.3.5. Attributes of the signals model the information conveyed by the ODP interactions expressed by such signals.

Once we have defined the main information object types of the system, and the possible actions that may take place, the way the library system works (from the perspective of the information viewpoint) needs to be defined in terms of how information is processed. Invariant, static and dynamic schemata are the mechanisms defined for that purpose.
A.3.3 Invariant schemata

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 may exhibit. The following are examples of invariants that can be defined for the library system:

1. Both library users and items should have unique identifiers in the system;
2. No item can be simultaneously referenced by two loans in the system;
3. There should be at most one Librarian and at least one Assistant on duty while the library is open;
4. The number of pending loans in the system should be consistent with the sum of the values of attribute borrowedItems of all the Borrower objects;
5. Borrowers who do not pay their fines will be eventually suspended;
6. Suspended borrowers who settle their debts will eventually be reinstated, and their borrowing rights restored.

Please note how some of these invariants have been incorporated into the UML class diagram that describes the system structure (shown in Figure A.20) in terms of the multiplicity of the associationEnds. This is the case, for instance, of invariant 2 (which is represented by a multiplicity "1" in the corresponding associationEnd).

Other invariants can be naturally expressed in UML by associating OCL constraints to some of the elements of the specification. For example, invariant 1 imposes that the identifiers of users and Library items should be unique in the system. This invariant can be expressed in terms of OCL constraints on the Library class:

\[ \text{Invariant 1} \]
\[ \text{context Library} \]
\[ \text{inv UniqueltemIdentifiers: self.item->isUnique(id)} \]
\[ \text{inv UniqueMemberIdentifiers: self.borrower->isUnique(id)} \]

Invariant 2 imposes that no item can be simultaneously referenced by two loans in the system. As mentioned before, this invariant has been implemented by a multiplicity "1" in the corresponding association end.

Invariant 3 states that there should be at most one Librarian and at least one Assistant on duty while the library is open.

\[ \text{Invariant 3} \]
\[ \text{context Library \ inv AtMostOneLibrarianAndAtLeastOneAssistantWhileLibraryOpen:} \]
\[ \text{self.isOpen implies (self.librarian->select(onDuty)->size()<=1) and} \]
\[ \text{(self.assistant->select(onDuty)->notEmpty())} \]

Invariant 4, which imposes a consistency check on the system, such that the number of pending loans should be consistent with the sum of the number of pending loans of each user, can be also expressed by an OCL constraint on the Library class:

\[ \text{Invariant 4} \]
\[ \text{context Library \ inv ConsistentNumberOfLoans:} \]
\[ (\text{self.borrower.borrowedItems->sum}()) = (\text{self.loan->select(status=extant)->size}()) \]

Other invariants may need to be expressed using different notations. In fact, invariants 5 and 6 can be considered as predicates in a given discrete linear temporal logic that imposes some fairness constraints. OCL is not expressive enough to specify them, although we can always either use a textual description of such predicates, or use any other notation (in this case we will consider an extension of OCL with the temporal logic operators "always" and "eventually"):

\[ \text{Invariant 5: Borrowers who do not pay their fines will eventually be suspended.} \]
\[ \text{context Borrower inv: eventually always (fines = 0) or always eventually (suspended = true)} \]

\[ \text{Invariant 6: Suspended borrowers who have paid their fines will eventually be released} \]
\[ \text{context Borrower inv: eventually always (fines > 0) or always eventually (suspended = false)} \]

Finally, other OCL constraints may express invariants relating to well-formedness rules of the model. For instance, the following constraint restricts the valid values of Loan objects:

\[ \text{context Loan inv ValidLoan: issueDate <= dueDate} \]

Similarly, other OCL expressions can help determining the value of some of the system attributes, e.g., when the library is open:

\[ \text{context Library inv OpeningTimes:} \]
\[ (\text{hour(self.clock.now) >= 8}) \text{ and (hour(self.clock.now) <5) implies self.isOpen = true} \]
All these *invariants* are expressed as constraints, and specified in the *InformationObjects* package, associated with the corresponding elements.

A.3.4 Static schemata

Static schemata provide instantaneous views of information, for example at system initialization, or at any other specific moment in time that is relevant to any of the system stakeholders. This specification of the instantaneous state of the *objects* is precisely the one provided by UML object diagrams (also known as *snapshots* in some UML dialects).

For instance, the UML package shown in Figure A.22 expresses the initial state of the system, just before the library opens for the first time, when there are no items, borrowers, or loans. There are, however, one clock, one assistant, and one librarian registered in the Library at that moment in time. At least one assistant should be present in order for such a configuration of *objects* to respect the *invariant schemata* specified by the multiplicity of the associations in the class diagram shown in Figure A.20. Please note as well how the constraints on the *Library* object explicitly specify the multiplicity of the links, and how this *static schemata* defines the initial values of the variables that store the system policies, as described in the textual regulations of the library.

![UML object diagram](image)

**Figure A.22 – Static schema with the initial state of the Library system**

Similarly, the UML object diagram shown in Figure A.23 expresses a *static schema* that models the state of the system at a moment in time (namely, at year end, when the state of the system should be recorded to serve as an inventory), in which there are only two *Borrowers* (John and Mary), one *Librarian* (Emerald), two *Assistants* (Eve and Pete), three *Books* (one copy of Ulysses and two copies of Dubliners), and one *Periodical* (today's edition of The Times). There is only one *Loan* (Mary borrowed one copy of Ulysses in March).
A.3.5 Dynamic schemata: Description of the system behaviour

The way the system evolves is dictated by the behaviour of the objects of the system, which in the information viewpoint is modelled in terms of a set of dynamic schemata. They describe the allowed state changes of the system or of any subset of its constituent information objects.

This clause presents dynamic information schemata that describe changes of state associated with the action types identified in A.3.2. In this case, such action types have been expressed by signals stereotyped «IV_Action».

NOTE – It is worth noting here that some authors have proposed the use of UML operations for expressing action types. However, this approach presents some limitations. For example, it forces actions to be owned by one object (i.e., the object to which the operation is assigned). In general, it may be the case that more than one ODP object might be related to a single action, because ODP interactions are pieces of shared behaviour, with no necessarily single owner or initiator. However, the interaction model of the UML is based on message exchange between objects, which forces all UML operations to be assigned to only one object. Thus, if ODP information actions are expressed by UML operations, the system designer has to decide, for every action, the object to which an operation expressing the ODP information action type is assigned. This is in general a difficult decision, and therefore more practical applications are required in order to identify a set of guidelines or patterns to support the practising modeller in assigning ODP action types to UML classifiers.

The behaviour of every information object is specified using UML stateMachines, which describe its state changes as a consequence of the occurrence of the signals that express the possible information actions previously specified. These stateMachines express the dynamic schemata of the ODP information specifications. Please notice how a signal causes changes in all stateMachines that define a transition for it. In this way we can model, in a natural manner, the fact that an ODP interaction may cause a state change in all objects related to that interaction, i.e., an ODP interaction is a piece of shared behaviour. This would be very difficult to do if ODP interactions were expressed by operations.

In this case, the dynamic schema of the library is specified in terms of the stateMachines of the classifiers that support the actions defined in clause A.3.2, namely the Librarian, Assistant, Borrower, Loan and Item. Figures A.24, A.25 and A.26 show some of these stateMachines, for illustration purposes.

The specification of these stateMachines has been developed based on the enterprise specification of their corresponding objects. Thus, Figure A.24 depicts the stateMachine of the Loan information object, based on the corresponding stateMachine of the Loan enterprise object depicted in Figure A.13.
Figure A.24 – StateMachine of the Loan information object

Likewise, Figure A.25 shows the stateMachine of the Borrower information object, based on the corresponding stateMachine of the Member enterprise object depicted in Figure A.14. In the information specification, two of the states of the enterprise object, Valid and Suspended, have been refined (defining them as composite states) to show the effects of fines imposed by the assistants and debt settlements (FineMember and PayFine actions). Figure A.25 also shows the transitions between the states, and the valid actions accepted in each one.

Figure A.25 – StateMachine of a Borrower information object

Finally, Figure A.26 shows the stateMachine of the Item information object.
NOTE – The previous diagrams show not only the effect of the actions on the corresponding information objects, but also the states in which the actions are allowed, serving as pre- and post-conditions for those actions.

A.3.6 Correspondences between the enterprise and the information specifications

Correspondences between the Enterprise and Information specifications are expressed in the corresponding package LibrarySystem (E-I Corr), as shown in Figure A.1. Correspondences are expressed using the correspondence profile (see [12.2]).

Figure A.27 shows an example of a correspondence between Loan enterprise and information objects. There is a top-level correspondence, LoanCorrespondence, which links these two types of objects. Such a correspondence is broken down into a set of correspondences, which establishes particular details of it.

Thus, correspondence LoanInstances establishes that the sets of Loan instances in the enterprise and information models should be consistent. This is specified by stating that the set of names of the instances of enterprise loans should include the set of names of the instances of information loans.

Similarly, correspondence CheckLoanAuthorization establishes a correspondence between the opaqueBehavior ValidateLoanRequest of the LibrarySystem role (see Figures A.5, A.10 and A.16), and the transitions between states of the Loan information object (see Figure A.24).

![Diagram of LoanCorrespondence and LoanInstances correspondences]

Figure A.27 – Example of correspondence between the enterprise and information specifications

Of course, top-level correspondence LoanCorrespondence is composed of more correspondences, not shown here for the sake of simplicity.
A.4 Computational specification in UML

A.4.1 Overview

The computational viewpoint is concerned with functional decomposition of an ODP system in distribution transparent terms. A computational specification defines units of function as computational objects, and the interactions among those computational objects, without considering their distribution over networks and nodes. This clause concentrates on the computational specification in UML of the borrowing process of the library system.

A.4.2 Computational objects and interfaces

The basic structure of the computational viewpoint specification of the Library system is shown in Figure A.28. Each package specifies the corresponding elements, and will be described in the following clauses. The elements of each package have been defined by making into components the functionality specified in the enterprise and information viewpoints, identifying the basic operations first and grouping them into interfaces. These interfaces define operations which handle data, as part of their parameters and return values. The types of these parameters are specified in the DataTypes package. Finally, the computational objects that own these interfaces are specified in the ObjectTemplates package.

![Basic structure of the computational viewpoint specification of the Library system](image)

It is interesting to note that the decomposition of the system functionality into computational objects that interact at interfaces provides the software architecture of the application. In UML, we express such architecture using a component diagram that describes the computational object templates and the computational interface templates at which these objects interact.

In the library system example, the software architecture of the application is composed, at the highest level, of three main components: one that describes the basic functionality of the system (LibrarySystemMainFunctionality), and other two (InterfaceToAssistant and InterfaceToLibrarian) which specify the user interfaces that the application will offer to assistants and librarians to interact with it, respectively. This is shown in Figure A.29.

In the figures that follow, to improve the clarity of the diagrams, the icons shown in Table A.3 have been used to represent instances of the corresponding stereotypes.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>«CV_Object»</td>
<td>Computational object</td>
</tr>
<tr>
<td>«CV_BindingObject»</td>
<td>Computational binding object</td>
</tr>
<tr>
<td>«CV_TemplateObject»</td>
<td>Computational template object</td>
</tr>
</tbody>
</table>

Each computational object is expressed as a component. Object interfaces are expressed as component ports. Finally, interface signatures are expressed as interfaces. Thus, computational objects interact which each other at computational interfaces (port instances), which are instantiated from their corresponding interface templates (ports). Each port uses or implements some interfaces, which specify the corresponding interaction signatures. In Figure A.29, the ports of the InterfaceToAssistant and InterfaceToLibrarian components make use of the services provided by the ports of the LibrarySystemMainFunctionality component.

Figure A.30 shows the interfaces that specify the interaction signatures of the Library system. They are all operation interface signatures, since our interaction mechanisms have been modelled as such. The way to identify these operations
is by inspecting the enterprise and information specifications, trying to capture and specify as computational operations the relevant enterprise processes, steps, and actions of the LibrarySystem enterprise object, together with the relevant actions of the information specification. The way to group them into operation interfaces that provide services depends on the designer's choice, and is usually based on the similarity of the functionality offered by each operation.

Once the high-level architecture of the application has been defined, the next step is to refine its components, specifying their internals. Figure A.31 shows the structure of the LibrarySystemMainFunctionality computational object. It has been refined into 6 computational objects (expressed as six components), each one dealing with a particular piece of functionality. Each component interacts with the rest through its ports, which express the corresponding computational interfaces. We can see how each port is of a particular type (described in the InterfaceTemplate package) and implements or uses several interfaces (which express the corresponding interface signatures shown in Figure A.30). The way to achieve such decomposition is something that again depends on the designer's choice.

![Figure A.29 – Component diagram with computational object templates and interface signatures of the system](image-url)
Figure A.30 – Interaction signatures

Figure A.31 – Internal structure of the LibrarySystemMainFunctionality computational object
The connections between the different components are shown using either the “ball and socket” notation that expresses implicit primitive bindings between the corresponding computational objects (see [9.2.17]), or some assembly connectors that express the explicit primitive bindings. We have also used some delegation connectors to map the external view of the component to its internal view (see [UML – 8.3.1]), specifying how the services provided by an external port are in fact provided by a port of one of its internal components.

Please note as well how in Figure A.31 we have included further information about the components, such as some of their internal realizing classifiers. For instance, the LoanMgr component is in charge of managing the loans in the system (representing, e.g., a database that stores and manages them) and thus contains a realizing classifier (the LoanManager «focus» class) which specifies its behaviour, and that owns the set of Loans of the systems (that represent the elements of the database). The structure and contents of such Loans are specified in the DataType package, which is shown in Figure A.32. These data types have been derived from the corresponding information object types (which in turn came from the enterprise artefacts, roles and objects).

Figure A.32 – Data types handled by the computational objects

A.4.3 Behaviour

Apart from the structural aspects, we need to specify the behaviour of the elements of a computational specification. State Machines can be used to express the internal behaviour of computational elements: ports, components and realizing classifiers. The way to use state Machines to represent that behaviour has already been illustrated in the enterprise and information specifications.

In case we want to specify object interactions, activities can be useful because they are abstractions of the many ways in which messages are exchanged between objects. Alternatively, UML interaction diagrams are more appropriate when messages and interaction protocols are the focus of design.

For illustration purposes, Figure A.33 shows a sequence diagram with the interactions that occur between the components of the computational specification during the borrowing process. First, an Assistant issues a loanRequest() operation, which is received by the component that implements it (BorrowingSystem). That component asks the MemberMgr for the details of the borrower, and then requests a validation. If the validation fails (reply message number 5), the BorrowingSystem registers and archives the loan in the system (through the LoanManager), and responds to the InterfaceToAssistant. Alternatively, i.e., if the member is valid (reply message number 11), the BorrowingSystem component asks the ItemMgr for details about the item to borrow, and the LoanMgr for the current loans of the borrower. Two alternative behaviours are possible then, depending on whether the request is valid or not (the conditions correspond to those specified in the information viewpoint: the loan limits are not exceeded, the item is indeed free, etc. – see Figure A.24). If the request is not valid then the loan is registered and archived, and a response is issued to the InterfaceToAssistant. Finally, if the loan request is valid then the item is marked as loaned (through the reserveItem() operation), the loan is registered in the system, and the Assistant is notified.

Please note how it is possible to incorporate environment contracts in the specification, expressed by constraints stereotyped «CVEnvironmentContract». In this case, the duration constraint corresponds to one of the requirements specified in the enterprise viewpoint, which mandated that the operation should not exceed 5 seconds (see Figure A.5).
A.4.4 Correspondences between the enterprise and computational specifications

Correspondences between the enterprise and computational specifications are expressed in the package LibrarySystem (E-C Corr), as shown in Figure A.1. Correspondences are expressed using the correspondence profile (see [12.2]).

Figure A.34 shows the LoanRequestDuration correspondence, that we have mentioned above between the requirement specified in the enterprise viewpoint about the duration of a loan request, and the corresponding environment contract in the computational specification. The other correspondence shown in this package relates the enterprise object type Loan with the computational object type Loan.

A.4.5 Correspondences between the information and computational specifications

Correspondences between the information and computational specifications are expressed in the package LibrarySystem (I-C Corr), as shown in Figure A.1. Correspondences are expressed using the correspondence profile (see [12.2]).
Figure A.35 shows an example of a correspondence between Loan information and computational objects: correspondence LoanCorrespondence links these two types of objects.

The other correspondence in that package establishes that the sets of Loan instances in the information model should be consistent with the objects stored by the LoanMgr component (i.e., with the loans stored in the application's database). This is specified by stating that the set of names of the instances of information object loan should coincide with the set of names of the instances stored by LoanMgr.

![Diagram](image)

Figure A.35 – Example of correspondence between the information and computational specifications

A.5 Engineering specification in UML

A.5.1 Overview

The engineering viewpoint is concerned with the mechanisms and functions required to support distributed interaction between objects in the system. An engineering specification defines the structure of node (nucleus, capsule, cluster, (basic) engineering objects), channel, and their management functions. This clause concentrates on the engineering specification in UML of the borrowing process of the Library system.

In the figures that follow, to improve the clarity of the diagrams, the icons shown in Table A.4 have been used to represent instances of the corresponding stereotypes.
### A.5.2 Computational objects

A set of computational objects, which this engineering specification will support, needs to be clarified. In this example, those are the computational objects defined in [A.4]: InterfaceToAssistant, InterfaceToLibrarian, and LibrarySystemMainFunctionality, which contains BorrowingSystem, FineSystem, LoanMgr, ItemMgr, MemberMgr, and LibraryAdmin. Those computational objects will be supported by corresponding basic engineering objects, which are deployed within clusters on nodes, and by the engineering infrastructure using nodes, nucleus, capsules, capsule managers, clusters, cluster managers, and channels.

**NOTE** – There are several architectural styles to define a physical deployment model, such as client-server, n-tier, model-view-controller (MVC), and service oriented architecture (SOA). In this example, n-tier and MVC architectural style are used. Even with the choice of architectural styles, there will be various types of node configurations, depending on requirements, such as performance, reliability, availability etc.

### A.5.3 Node configuration

The basic node configuration model for this example consists of four nodes: ClientTier, InteractionTier, EnterpriseTier, and EISTier (where EIS stands for enterprise information system) (see Figure A.36). An assistant (i.e., a user of the system) will use a desktop or notebook PC, which serves as ClientTier. A request from ClientTier node is
sent to a server node, which serves as the InteractionTier. A functional request is passed to another server node, which serves as the EnterpriseTier. Finally, data persistence is taken care of by yet another server node, which serves as the EIS_Server. The following diagram shows an overview of the node configuration.

A.5.4 Node structures

Each node consists of the node itself, a nucleus, and one or more capsules, capsule managers, clusters, cluster managers, BEOs, engineering objects, stubs, binders, protocol objects, and interceptors. In the node configuration above, BEOs are deployed as follows:

- BEOs for graphical user interface to access the system are deployed on the AssistantPC;
- BEOs to support n-tier and MVC architectural style are deployed on the InteractionServer and the EnterpriseServer; and
- BEOs for application specific computational objects are deployed on the EnterpriseServer and the EIS_Server.

Figure A.37 shows an example of a configuration of BEOs in the engineering specification. BEOs are deployed on various nodes and within clusters in capsules. BEOs can have interactions between them by using channels. For instance, Browser_APC1 BEO on AssistantPC1 node communicates with Controller BEO on InteractionServer node via Channel1.
Figure A.37 – Example of BEO configuration

NOTE – Capsule managers and cluster managers are not included, interfaces of various engineering objects are not shown for simplicity, and channels are shown using comments (see Figure A.39 for channel structure example).

As an example of a detailed node structure, Figure A.38 shows the internals of the EnterpriseServer node. In it, BorrowingSystem BEO makes use of the services provided by locally bound LoanMgr BEO, ItemManager BEO, and MemberMgr BEO. It also makes use of the services provided by FineSystem BEO in another cluster via a local channel. LoanMgr BEO then makes use of an external service (not shown in Figure A.38) via a channel to a BEO residing on another node. All BEOs in Cluster_ES1 are bound to ClusterMngr_ES1, and they all are bound to Nucleus_ES to have management services provided by the nucleus.
A.5.5 Channels

In this example, four channels exist: one between AssistantPC and InteractionServer, one between InteractionServer and EnterpriseServer, and two between Enterprise Server and IntegrationServers. The first channel comprises a stub, a binder, and a protocol object for the AssistantPC, and a stub, a binder, and a protocol object for the InteractionServer.

Figure A.39 shows an example of a channel called Channel_A, which allows binding and communication between a BEO of type2 and a BEO of type3. To allow both sides to use the channel, Channel_A provides interfaces to both ends.
A.5.6 Communication domain

Figure A.40 shows an example of a communication domain specification in UML, using packages that contain protocol objects which belong to the same communication domain.

A.5.7 Representing functions

Checkpointing, deactivation, cloning, recovery, reactivation, and migration are functions defined in [Part 3 – 8]. Since this Recommendation | International Standard defines stereotypes based on multiple metaclasses, the following is a series of diagrams describing the use of activities and actions to express functions. The diagrams are just examples. Users may add more actions or objectNodes to define their extended activity models.

Figure A.41 shows an example of a checkpointing process expressed as an activity, which contains a checkpointing action inside.

Figure A.42 shows an example of a deactivation process expressed as an activity, which contains a checkpointing action inside.
Figure A.42 – Deactivation process

Figure A.43 shows an example of a cloning process expressed as an activity.

Figure A.43 – Cloning process

Figure A.44 shows an example of a reactivation process expressed as an activity, which contains both deactivation and cloning actions inside.

Figure A.44 – Reactivation process

Figure A.45 shows an example of a recovery process expressed as an activity, which contains a cloning action inside.

Figure A.45 – Recovery process

Figure A.46 shows an example of a migration process expressed as an activity, which contains a deactivation action inside.
Finally, Figure A.47 is an example of a diagram showing the use of functions, which are represented in terms of an interface (TradingInterface) that specifies its signature, and the specific engineering object (Trader_A) that realizes or provides the function.

Figure A.47 – Use of functions

A.5.8 Channel creation and interface location

Figure A.48 shows how a nucleus provides a channel creation and an interface location interface to a BEO. Using this interface, the BEO can do several operations, such as publish its interface reference, bind to a trader, find a target object and a bind object, and get channel type and communication interface. The interface is stereotyped as «NV_NodeManagement».

Figure A.48 – Channel creation interface

A.5.9 Interface reference management domain

According to [Part 3 – 8.2.2], "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.

If necessary, it is possible to show a set of nodes belonging to an engineering interface reference management domain as in Figure A.49 below.

Figure A.49 – Interface reference management domain

A.5.10 Management functions

The diagram in Figure A.50 shows examples of interfaces that objects, nucleus, cluster manager, and capsule manager can provide. Interfaces are stereotyped to show their different types, and some operations are also stereotyped to show their kind of functionality. More operations than those described here can of course be specified, using the stereotypes defined in this Recommendation | International Standard.
Figure A.50 – Management functions

A.5.11 Correspondences between enterprise and engineering specifications

Correspondences between the enterprise and engineering specifications are expressed in the package LibrarySystem (E-N Corr), as shown in Figure A.1. Correspondences are expressed using the correspondence profile (see [12.2]). Figure A.51 shows the LibrarySystemCorrespondence that expresses the correspondence between the requirements specified in the enterprise viewpoint about the existence of Library system, and the nodes supporting the Library system in engineering viewpoint.
A.5.12 Correspondences between computational and engineering specifications

Correspondences between the computational and engineering specifications are expressed in the package **LibrarySystem (C-N Corr)**, as shown in Figure A.1. Correspondences are expressed using the correspondence profile (see [12.2]).

Figure A.52 shows the **ObjectCorrespondence** that describes the *correspondence* between the functionality specified in the computational viewpoint, and the *engineering objects* supporting this functionality in the engineering viewpoint.

**Figure A.51 – Example of correspondences between enterprise and engineering specifications**

NOTE – In this clause [A.5], generic UML constructs are used to complement ODP engineering specifications in UML. For instance, ports are used to represent an *engineering object's interfaces*, and operations are used to represent an *engineering interface's signatures*, since neither are defined in RM-ODP Part 3 engineering language.
A.6 Technology specification in UML

A.6.1 Overview

The technology viewpoint is concerned with the choice of technology to implement the ODP system. A technology specification defines technology objects (hardware, software, and network products), which implement implementable standards as its templates, implementation as a process of instantiation, and IXIT as implementation extra information for testing. This clause concentrates on the technology specification in UML of the borrowing process of the Library system.

The primary diagram for this specification is a deployment diagram.

In the figures that follow, to improve the clarity of the diagrams, the icons shown in Table A.5 have been used to represent instances of the corresponding stereotypes.

<table>
<thead>
<tr>
<th>Technology language icons</th>
</tr>
</thead>
<tbody>
<tr>
<td>«TV_Object»</td>
</tr>
<tr>
<td>«TV_TypeObject»</td>
</tr>
<tr>
<td>«TV_TemplateObject»</td>
</tr>
<tr>
<td>«TV_ImplementableStandard»</td>
</tr>
<tr>
<td>«TV_Implementation»</td>
</tr>
</tbody>
</table>

A.6.2 Node configuration

In a deployment diagram, a computer node is expressed as a node and lines are introduced to express communication links between the nodes. Different types of network can also be depicted as nodes.

Figure A.53 – Node configuration overview

Figure A.53 shows the node configuration of the Library system, in two parts. The upper part of the figure describes the deployment architecture of the system by showing the different technology objects types that will be used, and how they can be connected among themselves. The diagram shows that there will be three different kinds of computing resources (PCs, application servers and enterprise servers) and two different kinds of communication media (LAN and WAN).
PCs and application servers can be connected to LANs and WANs, whilst enterprise servers can only be connected to WANs.

The lower part of Figure A.53 describes the actual system, with concrete instance specifications of the nodes shown in the upper part of the diagram, showing the technology objects that will comprise the system, and how they are connected.

A.6.3 Node structure

Technology objects, such as those implementing the corresponding engineering objects, are deployed on each node. This is shown using a UML deployment diagram with the representation of hardware elements, software elements and network elements. There are cases where both the technology profile and standard profile (e.g., ExecutionEnvironment stereotype) need to be applied to the same element.

In UML, node structures can be specified both at the type level and at the object level. The diagram in Figure A.54 below shows the internal structure of the application servers used in the Library.

Figure A.54 – Node structure

A.6.4 IXIT

The truth of a statement in an implementation can only be determined by testing, and is based on a mapping from terms in the specification to observable aspects of the implementation. A test is a series of observable stimuli and events, performed at prescribed points known as reference points, and only at these points. These reference points are accessible interfaces. Four classes of reference point at which conformance tests can be applied are defined, which are programmatic reference point, perceptual reference point, interworking reference point, and interchange reference point.

Implementation extra information for testing (IXIT), which means additional information when performing a test against an implementation claiming to implement a defined specification or standard. In this respect, IXIT can be attached to any technology objects for interaction with a user, other technology objects in the same node, and other technology objects in other nodes.

Figure A.55 below shown an example of the addition of this additional information to two technology objects.
**A.6.5 Implementation**

The diagram shown in Figure A.56 below shows an example of an implementation process.

**A.6.6 Correspondences between engineering and technology specifications**

*Correspondences* between the computational and engineering specifications are expressed in the package **LibrarySystem (N-T Corr)**, as shown in Figure A.1. *Correspondences* are expressed using the correspondence profile (see [12.2]).

Figure A.57 shows the **BEO correspondence** that provides a correspondence link between an **ItemManager BEO** in the engineering viewpoint and an **ItemManager technology object** in technology viewpoint.
Figure A.57 – Example of correspondences between engineering and technology specifications
Annex B

An example of the representation of deontic concepts

(This annex does not form an integral part of this Recommendation | International Standard.)

This annex illustrates the use of the enterprise language deontic concepts to capture some of the behaviour constraints in a small specification fragment. The example shows how obligations and permissions can be expressed with precision without adding a large amount of detail to the basic enterprise behaviour.

B.1 The scenario

The scenario is concerned with a straightforward commercial situation, and is based on a minimal community (Figure B.1) containing just four roles, describing three active participants: customer, supplier and subcontractor and a role for the goods traded.

![Diagram of the community]

Figure B.1 – The community

The community also defines finer-grain objects to express commercial documents, such as purchase order and confirmation notes, goods items and tokens for the permits and burdens involved.

The broad view of the community behaviour is shown in Figure B.2. The customer places an order with a supplier, and that supplier decides to delegate the execution of the order to a subcontractor who prepares the goods and delivers them. At this point, the subcontractor informs the supplier that delivery is happening. The customer makes payment to the supplier after receipt of the goods.
B.2 Expressing the deontic constraints

To support the expression of the deontic constraints, we refactor this description so as to bring out as separate elements the participants in each significant interaction. These interactions are delineated and emphasized by placing them in distinct activityPartitions, labelled, where appropriate, by the interaction names as an aid to navigation to the details of the interactions (see Figure B.3).
Each of the interactions above is now elaborated in a separate interaction diagram which makes explicit the actor or artefact interaction roles and introduces the deontic tokens being manipulated. The dynamics of the tokens are then expressed in a series of interaction diagrams. These express constraints that take the form of pre- and post-conditions on the \( \text{occur()} \) operation associated with the interaction (see [7.2.10.3]).

Figure B.4 shows the PlaceOrder interaction. The customer provides a delivery permit that will enable the supplier to gain access to deliver the goods. This is transferred to the supplier when the action occurs. At the same time, an obligation is created at the supplier to deliver the goods in a timely fashion. Throughout this example, the PurchaseOrder (PO) is used as a primary key for identifying the appropriate tokens, since the participants could hold tokens for many other transactions.

![Interaction Diagram](image)

```plaintext
<context PlaceOrder::occur()>
  prog:
  \[ \text{customer delivery Permit one(purchaseOrder = self PurchaseOrder po)} \]
  \[ \text{post not self customer delivery Permit one(purchaseOrder = self PurchaseOrder po)} \]
  \[ \text{and self supplier delivery Permit one(purchaseOrder = self PurchaseOrder po)} \]
  \[ \text{and self supplier delivery Burden one(purchaseOrder = self PurchaseOrder po)} \]
</context>
```

Figure B.4 – The PlaceOrder interaction

We assume that the supplier now decides to delegate the fulfilment of this order to a subcontractor, which it does by performing a DelegatePO interaction (see Figure B.5). This interaction transfers the delivery permit and delivery burden to the subcontractor. It also establishes a burden for the subcontractor to confirm with the supplier when the delivery occurs, so that it is aware the delivery burden has been discharged and payment can be expected.
After preparation of the goods, the specific item requested is delivered, using the interaction shown in Figure B.6. The permit and burdens involved are already in the set held by the subcontractor. The delivery is achieved using the delivery permit and doing so discharges the delivery burden. However, it does create a payment burden on the customer to settle their debts.

Figure B.5 – The DelegatePO interaction

Figure B.6 – The DeliverGoods interaction
The other responsibility of the subcontractor on delivery is to confirm to the supplier that the process has been successful. This is done by the confirmation interaction (Figure B.7) and this interaction satisfies the confirmation burden held by the subcontractor, leaving it without further obligations with respect to this purchase.

**Figure B.7 – The confirmation interaction**

The final stage, in Figure B.8, is for the customer to pay the supplier, and so to discharge the payment burden. This completes our simple scenario, although any real system would also have behaviour dealing with violations such as non-payment or slow response in dealing with obligations.

**Figure B.8 – The Payment interaction**
INDEX

Abbreviations, 3
Accountability concepts, 15, 25
Accountable action, 25, 77
Action, 7, 13, 15, 21, 31, 39
Action role, 14, 23
Actor, 23, 47, 73, 78, 113
Agent, 25, 83
Announcement, 37, 42
Artefact, 23, 47, 63, 78, 113
Authorization, 25
Basic engineering object, 47, 49, 57, 62, 98
Behaviour, 9, 13, 14, 15, 21, 31, 36, 37, 67, 73, 94
Binder, 49, 58, 98
Binding, 8, 38, 44, 48, 50, 94, 104
Binding object, 38, 42
Burden, 14, 23, 111
Capsule, 49, 50, 52, 57, 98
Channel, 47, 49, 53, 57, 58, 100
Checkpoint, 49, 50, 55, 59, 101
Cloning, 50, 59, 102
Cluster, 49, 52, 57, 59, 98
Commitment, 25, 82
Communication domain, 50, 58, 101
Community, 13, 15, 20, 21, 22, 29, 72, 73, 111
Community object, 13, 15, 21, 22, 72
Compliance, 68
Computational interface, 39, 43, 45, 47, 91
Computational language, 7, 36, 40, 41, 46, 47
Computational metamodel, 39
Computational object, 7, 29, 36, 39, 41, 45, 47, 91, 97
Computational signature, 43, 48
Computational specification, 7, 29, 36, 39, 41, 47, 91
Conformance, 6, 30, 67, 68, 108
Contract, 21, 28, 31, 36, 42, 72, 94
Conventions, 3
Correspondence, 35, 47, 66
Correspondence endpoint, 66
Correspondence link, 65, 66
Correspondence metamodel, 66
Correspondence rule, 65, 66
Correspondences, 12, 13, 29, 35, 36, 47, 62, 65, 67, 70, 90, 95, 105, 106, 109
Correspondences specification, 65
Deactivation, 50, 59, 101
Declaration, 25, 82
Delegation, 15, 25, 82, 113
Deontic concepts, 14, 19, 23, 111
Distribution transparency, 6
Domain, 49, 50, 54, 57, 59, 104
Dynamic schema, 29, 31, 34, 85, 88
Embargo, 14, 23
Engineering interface, 47, 48, 58
Engineering interface reference, 48, 50, 59, 103
Engineering language, 7, 48, 51, 52, 60, 96
Engineering metamodel, 51
Engineering object, 49, 51, 56, 62
Engineering specification, 8, 29, 48, 56, 62, 96, 105
Enterprise language, 6, 13, 15, 25, 29, 70, 80, 111
Enterprise metamodel, 15
Enterprise object, 6, 13, 14, 20, 29, 35, 72, 78
Enterprise specification, 6, 13, 19, 28, 29, 35, 70, 72
structure of, 15
Environment contract, 36, 42, 94
Evaluation, 25, 82
Field of application, 20, 72
Flow, 37, 38, 45
Identifier, 48, 50, 55, 59
Implementable standard, 8, 63, 64, 107
Implementation, 63, 64, 67, 108, 109
Information language, 7, 30, 32, 34
Information metamodel, 32
Information object, 7, 29, 30, 32, 47, 84
class, 30
template, 31
type, 30
Information specification, 7, 30, 31, 32, 83, 90
Interaction, 7, 21, 22, 29, 31, 33, 36, 37, 49, 77, 88, 112
Interceptor, 49, 50, 54, 58, 98
Interface, 7, 29, 31, 36, 37, 39, 43, 44, 47, 49, 50, 52, 58, 91, 103, 104
Interface role, 22
Interrogation, 37, 38, 42, 43
Invariant schema, 29, 31, 33, 35, 86
Invocation, 38, 42
IXIT, 63, 64, 108
Metamodel
computational, 39
ing engineering, 51
enterprise, 15
information, 32
technology, 63
Migration, 51, 59, 102
Node, 49, 52, 58, 96, 97, 107
structure, 52, 98, 108
Nucleus, 49, 52, 58, 98, 103
Object modelling, 4
Object type, 20, 32, 42, 57, 63
Objective, 13, 15, 21, 72
ODP functions, 51, 55, 59
ODP system, 13, 19, 64, 70
Operation, 30, 37, 38, 42, 43, 48, 91
Operation interface, 37, 41, 43, 58
Party, 15, 25, 82
Permit, 14, 23, 111
Policy, 14, 24, 29, 80
Prescription, 25, 82
Principal, 15, 25, 82
Protocol object, 50, 58, 98
Reactivation, 50, 59, 102
Recovery, 50, 59, 102
Resource, 13, 23, 47
Scope, 1, 6, 20, 70
Signal, 37, 38, 41, 42, 43, 48, 85
Signal interface, 37, 38, 43, 58
Signature, 38, 41, 43, 45, 49, 58, 91
Static schema, 31, 34, 87
Stream, 38, 43
Stream interface, 37, 38, 43, 44, 45, 58
Stub, 49, 58, 98
System, 4, 6, 11, 13, 14
Technology language, 8, 63, 64
Technology metamodel, 63
Technology object, 8, 62, 63, 107
Technology specification, 8, 63, 64, 107
Template, 8, 20, 31, 32, 37, 41, 42, 56, 57, 63
Term, 12, 65, 66
Termination, 37, 42, 43
Transparency, 6, 38, 48
Transparency schema, 38, 45
UML, 8, 10, 11, 13, 20
Extension mechanisms, 9
UML profile, 11, 19, 32, 41, 56, 63, 66, 67
Unified Modelling Language. See UML
Universes of discourse, 10, 13, 20
Viewpoint specification, 5, 11, 12
Violation, 22
SERIES OF ITU-T RECOMMENDATIONS

Series A  Organization of the work of ITU-T
Series D  General tariff principles
Series E  Overall network operation, telephone service, service operation and human factors
Series F  Non-telephone telecommunication services
Series G  Transmission systems and media, digital systems and networks
Series H  Audiovisual and multimedia systems
Series I  Integrated services digital network
Series J  Cable networks and transmission of television, sound programme and other multimedia signals
Series K  Protection against interference
Series L  Construction, installation and protection of cables and other elements of outside plant
Series M  Telecommunication management, including TMN and network maintenance
Series N  Maintenance: international sound programme and television transmission circuits
Series O  Specifications of measuring equipment
Series P  Terminals and subjective and objective assessment methods
Series Q  Switching and signalling
Series R  Telegraph transmission
Series S  Telegraph services terminal equipment
Series T  Terminals for telematic services
Series U  Telegraph switching
Series V  Data communication over the telephone network

**Series X**  Data networks, open system communications and security
Series Y  Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z  Languages and general software aspects for telecommunication systems