INTERNATIONAL STANDARD

ISO/IEC 13235-1

First edition
1998-12-15

Information technology — Open Distributed Processing — Trading function: Specification

Technologies de l'information — Traitement distribué ouvert — Fonction commerciale: Spécifications

Reference number
ISO/IEC 13235-1:1998(E)

Contents

Foreword

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

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

International Standard ISO/IEC 13235-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 33, *Distributed application services*, in collaboration with ITU-T. The identical text is published as ITU-T Recommendation X.950.

ISO/IEC 13235 consists of the following parts, under the general title *Information technology — Open Distributed Processing — Trading function*:

— *Part 1: Specification*

— *Part 2: (TBD)*

— *Part 3: Provision of trading function using OSI Directory service*

Annexes A to D form an integral part of this part of ISO/IEC 13235.

Introduction

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

One of the components of the architecture (described in RM-ODP Part 3: Architecture) (ITU-T Rec. X.903 | ISO/IEC 10746-3) is the ODP Trading function. The trading function provides the means to offer a service and the means to discover services that have been offered. This Recommendation | International Standard provides an architecture for systems implementing the trading function and the specification of interfaces within the architecture.

NOTE – The specification of computational interfaces in this Recommendation | International Standard is technically aligned with the OMG Trading Object Service.

The goals of this Recommendation | International Standard are:

- to provide a standard which is independent of any implementation;
- to ensure implementations are capable of being made to interoperate (i.e. can be federated);
- to provide sufficient detail to allow conformance claims to be assessed.

Annex A is a normative ODP-IDL specification of the trading function interface signatures.

Annex B is a normative specification of the ODP trading function constraint language.

Annex C is a normative specification of the ODP trading function constraint recipe language.

Annex D is an informative description of a Service Type Repository.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

INFORMATION TECHNOLOGY – OPEN DISTRIBUTED PROCESSING – TRADING FUNCTION: SPECIFICATION

1 Scope and field of application

The scope of this Recommendation | International Standard is:

- an enterprise specification for the trading function;
- an information specification for the trading function;
- a computational specification for traders (i.e. objects providing the trading function);
- conformance requirements in terms of conformance points.

It is not a goal of this Recommendation | International Standard to state how the trading function should be realized. Therefore this Recommendation | International Standard does not include an engineering specification.

The field of application for this Recommendation | Intenational Standard is any ODP system in which it is required to introduce and discover services incrementally, dynamically and openly.

2 Normative References

The following Recommendations and International Standards contain provisions which, trough 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.

- ITU-T Recommendation X.901 (1997) | ISO/IEC 10746-1:1998, *Information technology – Open distributed processing – Reference Model: Overview.*
- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, *Information technology – Open Distributed Processing – Reference Model: Foundations.*
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, *Information technology – Open Distributed Processing – Reference Model: Architecture.*
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1998, *Information technology – Open Distributed Processing – Interface Definition Language.*
- ISO/IEC 13568[1)], *Information technology – The Z Specification Language.*

3 Notations

The information specification of the trading function is described using the Z formal description language. The signature of the computational interface for the trading function is described using ODP Interface Definition Language, in clause 8 and in Annex A.

4 Definitions

4.1 Definitions from ITU-T Rec. X.902 | ISO/IEC 10746-2

This Specification is based on the framework of abstractions and concepts developed in RM-ODP and makes use of the following definitions from RM-ODP Part 2: Foundations (see ITU-T Rec. X.902 | ISO/IEC 10746-2).

a) action;
b) activity;
c) behaviour;
d) behavioural compatibility;
e) binding;
f) client object;
g) conformance point;
h) contract;
i) domain;
j) establishing behaviour;
k) failure;
l) identifier;
m) initiating object;
n) instance;
o) interaction;
p) interface;
q) interface signature;
r) name;
s) object;
t) obligation;
u) ODP system;
v) permission;
w) policy;
x) prohibition;
y) quality of service;
z) reference point;
aa) responding object;
bb) role;
cc) server object;
dd) subtype;
ee) supertype;
ff) template;
gg) template type;
hh) trading;
ii) transparency;
jj) type;
kk) viewpoint.

4.2 Definitions from ITU-T X.903 | ISO/IEC 10746-3

This Specification is based on the framework of abstractions and concepts developed in RM-ODP and makes use of the following definitions from RM-ODP Part 3: Architecture (see ITU-T Rec. X.903 | ISO/IEC 10746-3).

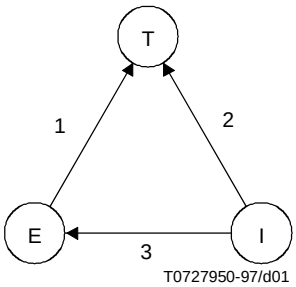
a) community;
b) computational interface template;
c) computational viewpoint;
d) dynamic schema;
e) engineering viewpoint;
f) enterprise viewpoint;
g) exporter;
h) information viewpoint;
i) invariant schema;
j) schema;
k) service export;
l) service import;
m) service offer;
n) static schema;
o) technology viewpoint;
p) <X> federation.

5 Overview of the ODP Trading Function

In the context of the ODP goal of providing distribution transparent utilization of services over heterogeneous platforms and networks, the role of the Trading Function is to allow users to find potential services. It is a corollary of distribution that the finding of services will occur dynamically.

The ODP trading function facilitates the offering and the discovery of instances of interfaces which provide services of particular types. A trader is an object that supports the Trading Function in a distributed environment. It can be viewed as an object through which other objects can advertise their capabilities and match their needs against advertised capabilities. Advertising a capability or offering a service is called "export". Matching against needs or discovering services is called "import". Export and import facilitate dynamic discovery of and late binding to services.

To export, an object gives the trader a description of a service together with the location of an interface at which that service is available. To import, an object asks the trader for a service having certain characteristics. The trader checks against the descriptions of services and responds to the importer with the location(s) of matched service interface(s). The importer is then able to interact with a matched service. These interactions are shown in Figure 1.



Sequence of interactions:
1. Export
2. Import
3. Service Interaction

Figure 1 – Interaction between the trader and its clients

The service interaction could be decoupled from the trading interactions (export and import) by modelling a service provider object and a service user object explicitly. This would imply interactions between service provider and exporter and between importer and service user that are trading actions, as defined in ITU-T Rec. X.902 | ISO/IEC 10746-2. However, these implied interactions need not conform to this Specification.

Due to the sheer number of service offers that will be offered worldwide, and the differing requirements that users of the trading service will have, it is inevitable that the trading service will be split up and that the service offers will be partitioned.

Each partition will, in the first instance, meet the trading needs of a community of clients (exporters and importers). Where a client needs a scope for its trading activities that is wider than that provided by one partition, it will access other partitions either directly or indirectly. Directly means that the client interacts with the traders handling those partitions. Indirectly, means that the client interacts with one trader only and this trader interacts with other traders responsible for other partitions. The latter possibility is referred to as federation of traders. In some cases, interceptors may be required between federated traders.

A user of a trader that interoperates with other traders, may associate with only one trader, and can transparently access the service offers of other traders with which that trader can interoperate .

Thus, the trading function in an ODP environment allows:

- objects to export (advertise) services;
- objects to import information about one or more exported services, according to some criteria;
- federation of traders.

5.1 Diversity and scalability

The concept of trading to discover new services is one that is applicable in a wide range of scenarios. A trader may contain a large number of offers of service and its implementation may be inclined to be based upon a database. Or, a trader may contain a few offers only and so be implementable as a memory resident trader. These two cases exhibit different qualities: availability and integrity in the first case and performance in the second. The variation in these scenarios illustrates the need for scalability, both upwards for very big systems and downwards for small, fast systems.

To discover any arbitrary offer of service, a trader needs all offers to be, in some sense, visible to it. One partition cannot hold every offer, many will necessarily be held at other partitions, therefore, in addition to a number of offers, a trader must possess information about other partitions. However, there is no need for a trader to know about all other partitions. Some of this knowledge can be obtained indirectly via other traders.

The partitioning of the offer space and the limited knowledge held with one partition about other partitions is the basis for meeting requirements for both distribution and contextualisation of the trading function.

5.2 Linking traders

The requirements to contextualise the offer space and to distribute the trading function are both met by linking traders together. By linking to other traders, each trader makes the offer space of those traders implicitly available to its own clients.

Each trader has a horizon limited to those other traders to which it is explicitly linked. As those traders are in turn linked to yet more traders, a large number of traders are reachable from a given starting trader. The traders are linked to form a directed graph with the information describing the graph distributed among the traders. This graph is called the trading graph.

Links may cross domain boundaries: administrative, technological or whatever. Trading may thus be a federated system, i.e. one that spans many domains.

5.3 Policy

To meet the diverse requirements likely to be placed upon the trading function, some degrees of freedom are necessary when specifying the behaviour of a trader object. To accomplish this, and yet still meet the goals of this Specification, the concept of policy is used to provide a framework for describing the behaviour of any ODP conformant trading system.

This Specification identifies a number of policies and gives them semantics. Each policy partly determines the behaviour of a trader. When claiming conformance, an implementation may need to state which combination of policies will ensure conformant behaviour.

Policies may be communicated during interaction, in which case they relate to an expectation on subsequent behaviour.

6 Enterprise specification of the Trading Function

The scope of an enterprise specification is defined in RM-ODP Part 3: Architecture (see ITU-T Rec. X.903 | ISO/IEC 10746-3). This enterprise specification identifies the objectives and the policy statements that govern the activities of a trading function.

The objective of the trading function is to provide the means to offer and to discover instances of a particular type of service, with particular characteristics.

A trading community is comprised by members that have different roles, for example, trader, exporter and importer. An object can have several roles within the same community. For example, an object can both be an importer and an exporter.

The trading activities of the community are service exports and service imports. These activities are governed by a set of policies of the trading community. A service import activity may propagate from one trading community to another. In such a case the domains associated with these two traders are federated. These trader domain boundaries may coincide with other domain boundaries (e.g. type domain or security policy domain).

A policy is a set of rules with a particular objective. Each rule constrains some aspects of a trader's behaviour consistent with the common objective. Members of the trading community are obliged to obey the rules of the policies. These rules provide the guidelines for decisions to satisfy the community's objectives. The rules are not prescribed in this Specification. The enterprise specification identifies the set of policies that limit the trader in certain type of behaviour. The policies identified provide a framework within which the trader object's behaviour may be implemented or configured.

6.1 Communities

6.1.1 **trading community**: A community of objects established for the purpose of trading and governed by a trading policy. The objects perform roles listed in 6.2.

A single trading community (at one level of abstraction) may be refined into a number of interworking trading communities at a second, more detailed level of abstraction. Subject to community policy, the interworking of trading communities at the detailed level is able to maintain the impression of the single abstract community, allowing objects with trader, importer or exporter roles in one subcommunity to interact with objects in any other subcommunity.

6.2 Roles

Objects may play the following roles within a trading community.

6.2.1 **trader**: A role which registers service offers from exporter objects and returns service offers upon request to importer objects according to some criteria.

6.2.2 **exporter**: A role which registers service offers with the trader object.

6.2.3 **importer**: A role which obtains service offers, satisfying some criteria, from the trader object.

6.2.4 **trader administrator**: A role which defines, manages, and enforces the trader policy of the trader object. The trader administrator is the controlling object of a trader domain (the trader and its set of service offers).

6.2.5 **service offer**: A role which maintains a description of a service.

NOTE – The description may be the basis of a future contract.

6.3 Activities

The following activities are relevant to a trading community.

6.3.1 **service export**: A chain of actions by an exporter object and the trader object which establish and terminate a liaison in which the trader object is permitted to provide the exporter object's service offer to a group of importer objects.

6.3.2 **service import**: A chain of actions between an importer object and the trader object in which the importer object obtains a number of service offers which meet some criteria.

6.4 Policies

The behaviours of enterprise objects within a trading community are governed by the policies of the trading community. Some policies govern trading activities, and some policies place constraints on other aspects of behaviour of a trader and other roles in the trading community, consistent with the common objective of the community. Where an activity involves interactions between objects, the resulting policy will be a compromise between the policies of the interacting objects. The compromise will be reached via a form of arbitration.

NOTE – For example, a trader object may be governed by policies such that it is obliged to propagate a search to a depth of 2 links, but it is also permitted to terminate a search after propagating a search to a depth of 1 link. If the trader object is permitted (or obliged) to meet an importer's requirement regarding depth of links to be traversed for a search, then there is a need for some rules to arbitrate between conflicting policies.

6.4.1 **Export activity policy**: A set of rules related to the service export activity (i.e. the offering of services so that they might subsequently be discovered by other objects).

The policy may include, amongst other things:

- an obligation for a service offer to be described in a specific way;
- a prohibition of specified service import activities from discovering the service offer;
- an obligation for a service offer providing rules to be evaluated as part of a service import activity.

Each exporter may have its own export policy. This would describe the exporter's expectation of a service export. Therefore, the service export activity is governed by both the trader's export policy and the exporter's export policy.

6.4.2 **Import activity policy**: A set of rules related to the service import activity (i.e. the attempt to discover offered services that meet a specified requirement.

The policy may include, amongst other things:

- an obligation to limit the use of resources, including duration of activity;
- permissions to propagate the service import to one or more interworking trading communities.

6.4.3 **Arbitration policy**: A set of rules to arbitrate on conflicting rules arising during trading activities.

The policy may include, amongst other things, an obligation to arbitrate in favour of the trader object's rules on:

- use of resources during service import;
- propagating service import activities.

6.4.4 **Service offer acceptance policy**: A set of rules restricting the set of service offers that will be accepted by the trader.

6.4.5 **Type management policy**: A set of rules related to the specification of types and the relationship between types.

NOTE 1 – The policy may be to defer to a type repository function with respect to either or both of these aspects.

NOTE 2 – Examples would be to use name equivalence or to use signature subtyping in type matching.

6.4.6 **Search policy**: A set of rules guiding the search for suitable service offers through the trading system.

6.5 Structuring rules

6.5.1 Community rules

In a trading community there must be an object which assumes a trader role (a trader object). Becoming a member of a trading community enables an object to interact with the trader object in an importer or an exporter role. An object may assume the exporter role, the importer role, or both the exporter and importer roles.

An "enterprise" may include multiple trading communities. An object can be a member of multiple trading communities. The trader object of one community may assume an importer or exporter role within another community of which it is a member.

The community may span several domains with respect to security, types, management, remuneration, etc.

Each trader, along with its set of service offers, is a trader domain. Thus, a set of trader domains which interoperate within a trading community is a federation of traders.

NOTE – Federated traders domains do not always require interceptors placed at their boundary in the engineering viewpoint.

6.5.2 Transfer rules

Exporter objects can export offers for services which they provide at their own interfaces, or may export offers for services provided by a distinct service provider object.

Importer objects can import service offers for their own use, or for use by distinct service user objects.

6.5.3 Delineation of authority rules

Each trader administrator object of a trading community has complete control over its own trader object.

The exporter object is responsible for the accuracy of its service offers.

For traders to be a member of an established federation of traders:

- one trader is not obliged to perform an activity initiated by another trader;
- each trader must have complete autonomy with respect to its own trader policies.

In particular, each trader determines its own trader search policies over the group of interworking traders.

6.5.4 Quality of service rules

The trader object is neither accountable nor responsible for the quality of services described in service offers.

A trader object may be obliged to ensure the timely removal of service offers.

NOTE – Two examples for achieving this are:

1) A trader service offer acceptance policy may oblige service offers to have an expiry date. The trader object is permitted to remove expired service offers.
2) A trader import policy may prohibit the trader object from returning service offers which have expired at the time of an import.

6.5.5 Matching rules

A service import requires computational interface signature type checking. It can, in addition, involve further levels of checking for subtype or supertype relationships, behavioural compatibility and environment constraints. Further checking on enterprise, information, engineering, and technology aspects may also be provided.

7 Information specification of the Trading Function

7.1 Overview

The scope of an information specification is defined in RM-ODP Part 3: Architecture (see ITU-T Rec. X.903 | ISO/IEC 10746-3). This information specification describes the types of information and the relationships between them which are required to define the ODP trading function. It uses the information language defined in RM-ODP and where appropriate interprets the language in terms of the formal specification notation Z. Paragraphs of formal notation are interspersed with English text in the usual Z style.

The information specification in this clause defines:

- basic concepts for information used in this Specification;
- static, invariant and dynamic schemata for this Specification.

7.2 Basic concepts

7.2.1 Interfaces

A service is offered at an interface. There is a need to for a service offer to identify the interface signature type and the interface identifier of the service interface.

7.2.1.1 Interface signature type

The interface signature type identifies the signature of interfaces of objects.

In Z, interface signature types are formally defined by introducing a given set to represent the values they can take:

[InterfaceSignatureType]

7.2.1.2 Interface identifier

An interface identifier identifies an interface at which a service is available or required.

In Z, interface identifiers are formally defined by a given set.

7.2.2 Service type

A service is a set of capabilities provided by an object at a computational interface. A service is an instance of a service type.

A service type definition consists of an interface signature type, a set of service property definitions, and a set of rules about the modes of the service properties.

Service property definitions are explicitly described in the formal specification in terms of names, value types and modes. The valid modes are:

- normal (read and write but optional presence);
- read-only (read but optional presence);
- mandatory (read and write mandatory presence); and
- read-only and mandatory (read only, mandatory presence).

A value type is a set of values.
[Name, Value]

$ValueType == \mathbb{P} Value$

$Mode ::= \{ normal, readonly\_mandatory, readonly\_mandatory \}$

Service properties contain information about computational aspects (such as the behaviour and the environment of an interface) as well as describing the technology, engineering, information, and enterprise aspects of the service.

The formal definition of service type is given in the Z schema *ServiceType* . It groups together an interface signature type and a set of service property definitions.

ServiceType

$signature : InterfaceSignatureType$
$prop\_defs : Name \pfun (ValueType \times Mode)$

In Z, functions are used to extract the set of property names which must be present (i.e. they are *mandatory* or *readonly_ mandatory*) and the set of property names which cannot be modified (i.e. they are *readonly*, or *readonly_ mandatory*). These two functions are formally defined as follows:

$$\textbf{mandatory\_props} : ServiceType \rightarrow \mathbb{P}\ Name$$
$$\textbf{readonly\_props} : ServiceType \rightarrow \mathbb{P}\ Name$$

$$\forall\ s: ServiceType \bullet \textbf{mandatory\_props}\ s =$$
$$\{\ n : Name \mid second\ (s.prop\_defs\ n) \in \{mandatory, readonly\_mandatory\}\}$$
$$\forall\ s: ServiceType \bullet \textbf{readonly\_props}\ s =$$
$$\{n : Name \mid second\ (s.prop\_defs\ n) \in \{readonly, readonly\_mandatory\}\}$$

7.2.3 Service type subtyping rules

The general rules for service subtyping for a conformant trader are as follows:

In the most general case, a service type **b** is a subtype of service type **a**, if, and only if:

- the interface signature type of **b** is a subtype of the interface signature type of **a**;
- all the named properties of **a** are in **b**;
- all of the named properties of **a** have a value type which is a supertype of the identically named property in **b**;
- all of the named properties of **a** have a mode which is a supertype of the mode of the identically named property in **b**.

NOTE – The above rules are equivalent to the normal ODP interface subtyping rules, if the properties are viewed as operations, with the type and mode as return arguments of the operations.

The Z representation requires that three relations are defined to represent interface signature subtyping, value supertyping and mode supertyping. The interface signature subtyping rules are those given in ITU-T

Rec. X.903 | ISO/IEC 10746-3 and are not further defined here. Formal definitions are given for supertyping relations across the modes and value types.

$$\_\ \textbf{is\_sig\_subtype\_of}\ \_ : InterfaceSignatureType \leftrightarrow InterfaceSignatureType$$
$$\_\ \textbf{is\_value\_supertype\_of}\ \_ : ValueType \leftrightarrow ValueType$$
$$\_\ \textbf{is\_mode\_supertype\_of}\ \_ : Mode \leftrightarrow Mode$$

$$\forall\ a,b:Mode \bullet a\ \textbf{is\_mode\_supertype\_of}\ b \Leftrightarrow$$
$$(a,b) \in \{(normal,readonly),(normal,mandatory),(normal, readonly\_mandatory),$$
$$(readonly,readonly\_mandatory), (mandatory,readonly\_mandatory)\}$$
$$\forall\ a,b:ValueType \bullet a\ \textbf{is\_value\_supertype\_of}\ b \Leftrightarrow b \subseteq a$$

$$\_\_\ \textbf{is\_subtype\_of}\ \_\_ : ServiceType \leftrightarrow ServiceType$$

$$\forall\ a,b : ServiceType \bullet b\ \textbf{is\_subtype\_of}\ a \Leftrightarrow$$
$$b.signature\ \textbf{is\_sig\_subtype\_of}\ a.signature\ \wedge$$
$$dom\ a.prop\_defs \subseteq dom\ b.prop\_defs\ \wedge$$
$$(\forall\ n: dom\ a.prop\_defs \bullet$$
$$first\ (a.prop\_defs\ n)\ \textbf{is\_value\_supertype\_of}\ first\ (b.prop\_defs\ n) \wedge$$
$$second\ (a.prop\_defs\ n)\ \textbf{is\_mode\_supertype\_of}\ second\ (b.prop\_defs\ n))$$

7.2.4 Service offer

A service offer advertises a service. It is an assertion made by an exporter about a service being offered for use by other objects at a computational interface. It consists of an instantiation of a service type, an identifier for the service offer and an identifier for the interface through which the service may be used. A service offer may also include a set of service offer property values.

For Z, a given set is introduced to represent the service offer identifiers that are unambiguous within a trading community.

$[ServiceOfferIdentifier]$

The formal Z definition of a service offer is given by the schema *ServiceOffer*. Each service offer must satisfy it's service type: i.e. the service offer must have a value for all the properties defined by the service type as mandatory or readonly-mandatory; and all those properties which are properties of the service type must have values drawn from the sets defined in the service type.

ServiceOffer

$$
\begin{array}{|l}
service\_type : ServiceType \\
prop\_vals : Name \pfun Value \\
interface\_identifier : InterfaceIdentifier \\
service\_offer\_identifier : ServiceOfferIdentifier \\
\hline
\textbf{mandatory\_props}\ service\_type \subseteq \mathrm{dom}\ prop\_vals \\
\forall n : \mathrm{dom}\ service\_type.prop\_defs \cap \mathrm{dom}\ prop\_vals \bullet \\
\quad prop\_vals\ n \in first\ (service\_type.prop\_defs\ n) \\
\hline
\end{array}
$$

7.2.5 Criteria and constraints

Policies in the enterprise viewpoint are represented by criteria and constraints in the information viewpoint.

There are three aspects to specifying the set of service offers which are acceptable results of a search or select action. Two of these are filtering relations over the service properties and service offer properties. The first defines the essential properties which cannot be violated in order to match. The second is a preference which acts as a selection filter when there are many offers which match the essential properties. The third aspect is a scoping relation which restricts the set of service offers which are to be compared with the matching rules. These specifications may be provided by both the importer and the trading system, with the final specification being a combination of the two. The trading system is defined in 7.3.

7.2.5.1 Matching criteria

Rules which are applied to the total set of service offers to yield a smaller set of acceptable service offers.

In Z , the matching criteria may be expressed as the set of all possible service offers which would satisfy the matching rules. The process of applying these rules to a set of service offers may then be represented by taking the intersection of two sets.

$MatchingCriteria == \mathbb{P} ServiceOffer$

7.2.5.2 Preference criteria

Rules which are applied to the set of acceptable service offers to yield an ordered sequence of service offers.

In Z, the preference criteria may be expressed as a total function from sets of service offers to a sequence of service offers. The application of these rules is formally described in 7.5.10.

$PreferenceCriteria == \mathbb{P} ServiceOffer \rightarrow \textbf{seq}\ ServiceOffer$

7.2.5.3 Scope criteria

Rules which restrict the set of service offers which are to be compared with the matching rules.

In Z, scope criteria may be expressed as a set of service offers.

$ScopeCriteria == \mathbb{P} ServiceOffer$

7.2.5.4 Edge criteria

Rules which restrict the set of nodes reachable from a given node.

In Z, edge criteria can be expressed as associations:

$EdgeCriteria == Node \times Node$

7.2.5.5 Trader matching constraint

A constraint on the matching criteria imposed by policy of the trading system.

In Z, matching constraints can be specified in the same manner as matching criteria. Applying these constraints may be represented by set intersection.

7.2.5.6 Trader preference constraint

A constraint on the preference criteria imposed by policy of the trading system.

In Z, preference constraints are represented by a total function between two sets of preference criteria. The application of this function is described in 7.5.10.

$PreferenceConstraint == PreferenceCriteria \rightarrow PreferenceCriteria$

7.2.5.7 Trader scope constraint

A constraint on the scope criteria imposed by trader policy.

In Z, scope constraints are represented by a set of service offers as for scope criteria.

7.2.5.8 Trading system constraints

For the formal Z specification it is convenient to collect the constraints of the trading system together into the schema *TradingSystemConstraints*. This definition is used in 7.5.9.

TradingSystemConstraints

$trader\_matching : MatchingCriteria$
$trader\_scope : ScopeCriteria$
$trader\_preference : PreferenceConstraint$

7.2.6 Search request

A search request is a specification of the importer policy applying to a particular search action.

In Z, a search request may be modelled as a schema consisting of the importer's matching, scope and preference criteria and the service type of the desired service. All offers which satisfy the importers matching requirements must have a service type which is a subtype of the specified service type. This definition is used in 7.5.9.

Search Request

importer_matching : *MatchingCriteria*
importer_scope : *ScopeCriteria*
importer_preference : *PreferenceCriteria*
importer_service_type : *ServiceType*

$\forall s{:}importer\_matching \bullet s.service\_type$ **is_subtype_of** *importer_service_type*

7.3 Invariant schema

The service offer space will be partitioned and associated with any one partition will be knowledge of a limited number of other partitions. This pattern is repeated for focusing on any given partition. In the information specification, this is modelled as a directed graph, where the nodes represent the partitions and the edges represent the knowledge relating partitions. This graph is referred to as the trading graph.

There may be any number of bases upon which to partition the offer space, over and above distribution. For example the partitioning may be based upon:

- properties of the location (e.g. machine architecture);
- properties of the service offers (e.g. a security classification);
- properties of the service (e.g. it's availability).

An edge may have properties that describe the perception of one partition when viewed from another partition.

The components of the trading system are:

- A set (*offers*) of service offers which are available for import;
- A set (*nodes*) of nodes into which these service offers are partitioned;
- A relationship (*edges*) between nodes to represent the edges of the trading graph, which governs the propagation of searches;
- Sets (*edge_properties*) of properties which are associated with the edges;
- Service offers are distinguished by their service offer identifiers, which are unique and unambiguous. This is captured by the invariants of the *TradingSystem* schema in Z.

Individual nodes are formally defined in Z by introducing a given set.

[Node]

All service offers must be allocated to one, and only one node. No overlapping or sub-set relations between the nodes are permitted. This restriction is captured by the partial function *partition* which maps each service offer in the trading system to a single node. An exporter may export the same service to more than one node, by creating another service offer with a different service offer identifier.

The computational viewpoint maps nodes into traders. This mapping is one-to-one (each node in the information viewpoint is a single trader object). Hence in the computational viewpoint an edge corresponds to a computational interface between traders.

The following Z schema represents the state of the trading system.

TradingSystem

$$
\begin{aligned}
&offers : \mathbb{P}\ ServiceOffer \\
&nodes : \mathbb{P}\ Node \\
&partition : ServiceOffer \nrightarrow Node \\
&edges : Node \leftrightarrow Node \\
&edge\_properties : (Node \times Node) \nrightarrow \mathbb{P}\ Property \\
\hline
&dom\ partition = offers \\
&ran\ partition \subseteq nodes \\
&dom\ edges \cup \text{ran}\ edges \subseteq nodes \\
&dom\ edge\_properties = edges \\
&\forall p,q : ServiceOffer \bullet p.service\_offer\_identifier = q.service\_offer\_identifier \Leftrightarrow p = q
\end{aligned}
$$

7.4 Static schema

Static schema applies the state of the trading system at a particular location in time.

For the consistency and completeness of the formal Z specification, the action to initialise a trading system is included.

When a trading system is created it has the null value for each state component:

Initialize

$$
\begin{aligned}
&TradingSystem' \\
\hline
&offers' = \varnothing \\
&nodes' = \varnothing \\
&partition' = \varnothing \\
&edges' = \varnothing \\
&edge\_properties' = \varnothing
\end{aligned}
$$

7.5 Dynamic schemata

This subclause presents dynamic information schemata which describe changes of state associated with the following actions:

- Export – Add a service offer to the service offer space of the trading system.
- Withdraw – Withdraw a service offer from the service offer space of the trading system.
- Modify Offer – Change the service property and service offer property values associated with a service offer whilst preserving the service offer identifier.
- Add Edge – Add an edge to the trading system's set of edges.
- Remove Edge – Remove an edge from the trading system's set of edges.
- Modify Edge – Change the properties of an edge.
- Add Node – Add a node to the trading system's set of nodes.
- Remove Node – Remove a node from the trading system's set of nodes.
- Search – Search for the subset of service offers which satisfy some matching criteria and scoping criteria.
- Select – A useful specialisation of search which returns a sequence of service offers ordered according to some preference constraint.

7.5.1 Export

The dynamic schema Export describes the behaviour of adding a service offer to the trading system.

Successful export of a service offer is as follows. The new service offer is added to the existing set of service offers and is associated with a single node. The source of the service offer identifier component of the service offer is not identified: it is generated by the trader, rather than the exporter. The service offer identifier is passed back to the exporter. The edge relationships, and the properties associated with edges, are not changed.

The pre-condition for this action is that the new service offer identifier is not currently used in the trading system. Another pre-condition requires that the node with which the offer is to be associated exists in the trading system.

In Z, the *Export* schema represents the corresponding behaviour.

ExportOK

$$
\begin{array}{l}
\Delta TradingSystem \\
new\_offer? : ServiceOffer \\
node? : Node \\
service\_offer\_identifier! : ServiceOfferIdentifier \\
\hline
\forall\, s : offers \bullet s.service\_offer\_identifier \neq new\_offer?.service\_offer\_identifier \\
node? \in nodes \\
offers' = offers \cup \{new\_offer?\} \\
partition' = partition \cup \{new\_offer? \mapsto node?\} \\
service\_offer\_identifier! = new\_offer?.service\_offer\_identifier \\
nodes' = nodes \\
edge\_properties' = edge\_properties \\
edges' = edges
\end{array}
$$

If the pre-conditions of the *ExportOK* schema are not met, the state of the trading system remains unchanged. In Z. this is captured in the following error schema, the pre-condition for which is the negation of the pre-condition of *ExportOK*.

ExportError

$$
\begin{array}{l}
\Xi TradingSystem \\
new\_offer? : ServiceOffer \\
node? : Node \\
\hline
\exists\, s : offers \bullet s.service\_offer\_identifier = new\_offer?.service\_offer\_identifier \lor \\
node? \notin nodes
\end{array}
$$

Export will succeed or fail depending on the pre-conditions of the above Z schemata.

$$Export \triangleq ExportOK \lor ExportError$$

7.5.2 Withdraw

The dynamic schema Withdraw removes a service offer from the trading system.

The offer is withdrawn from the set of offers. The edges and edge properties are unchanged. Note that this behaviour does not specify how the action was initiated. Thus the behaviour applies whenever an offer is withdrawn by the trader, the exporter or some other authorized agent.

The pre-condition for this action is that the service offer must exist in the trading system.

In Z, the *Withdraw* schema represents the corresponding behaviour.

WithdrawOfferOK

$\Delta TradingSystem$
$old\_offer? : ServiceOffer$

$old\_offer? \in offers$
$offers' = offers \setminus \{old\_offer?\}$
$partition' = \{old\_offer?\} \lhd partition$
$nodes' = nodes$
$edges' = edges$
$edge\_properties' = edge\_properties$

If the pre-conditions of *WithdrawOfferOK* are not met, the state of the trading system remains unchanged.

WithdrawOfferError

$\Xi TradingSystem$
$old\_offer? : ServiceOffer$

$old\_offer? \notin offers$

WithdrawOffer will succeed or fail depending on the pre-conditions of the above Z schemata.

$WithdrawOffer \triangleq WithdrawOfferOK \lor WithdrawOfferError$

7.5.3 Modify Offer

The dynamic schema ModifyOffer defines the behaviour of modifying the service properties and service properties associated with a service offer. This behaviour is not simply Withdraw followed by Export, because it also guarantees that the service offer identifier is preserved.

A service offer may be modified in three ways:

- properties may be deleted, as long as they are not mandatory properties;
- new properties may be added, provided that they are not already assigned a value in the service offer;
- existing properties may be updated, provided that they are not readonly.

In Z, the *ModifyServiceOffer* schema represents the corresponding behaviour. The definition is given in several steps. First, a definition is given to modify a service offer by replacing the service property values, provided that the above constraints hold. The other components of the offer are not affected: in particular the service offer identifier is preserved. Note that this behaviour does not specify how the action was initiated. Thus the behaviour applies whenever an offer is modified by the trader, the exporter or some other authorized agent.

ModifyServiceOfferOK

$\Delta ServiceOffer$
$delete? : \mathbb{P}\ Name$
$new? : Name \pfun Value$
$update? : Name \pfun Value$

$delete? \cap \mathbf{mandatory\_props}\ service\_type = \varnothing$
$dom\ update? \cap \mathbf{readonly\_props}\ service\_type = \varnothing$
$delete? \cup dom\ update? \subseteq dom\ prop\_vals$
$dom\ new? \cap dom\ prop\_vals = \varnothing$
$prop\_vals' = (delete? \lhd prop\_vals) \oplus update? \cup new?$
$service\_type'.signature = service\_type.signature$
$interface\_identifier' = interface\_identifier$
$service\_offer\_identifier' = service\_offer\_identifier$

Error conditions for this behaviour arise when an attempt is made to:

- update or modify a non-existant property;
- modify a property which is either read-only or mandatory read-only; or
- add a new property which already exists in the service properties.

If this happens, all properties of the service offer remain unchanged, and the interface signature is also unchanged.

ModifyServiceOfferError

$$
\begin{array}{|l}
\hline
\Xi ServiceOffer \\
delete? : \mathbb{P}\ Name \\
new? : Name \pfun Value \\
update? : Name \pfun Value \\
\hline
(\exists\, n : delete? \cup dom\ update? \bullet n \notin dom\ prop\_vals) \\
\lor\ delete? \cap \textbf{\textit{mandatory\_props}}\ service\_type \neq \emptyset \\
\lor\ dom\ update? \cap \textbf{\textit{readonly\_props}}\ service\_type \neq \emptyset \\
\lor\ dom\ new? \cap dom\ prop\_vals \neq \emptyset \\
\hline
\end{array}
$$

The Z technique of promotion is now employed to promote the *ModifyServiceOffer* schema so that it is applied to a specific offer in the trading system. The offer to be modified is identified by the definition of a framing schema, Φ*ModifyOffer*. The framing schema states that:

- the identifier of the offer to be changed is known to the trading system;
- after the action, the selected offer has been changed to the new value. All other offers are unchanged and the offer remains in the original node;
- the edges and edge properties are unchanged.

Φ*ModifyOffer*

$$
\begin{array}{|l}
\hline
\Delta TradingSystem \\
\Delta ServiceOffer \\
modified\_offer? : ServiceOffer \\
\hline
modified\_offer? \in offers \\
\theta ServiceOffer = modified\_offer? \\
offers' = (offers \setminus \{\theta ServiceOffer\}) \cup \{\theta ServiceOffer'\} \\
partition' = \\
\quad (\{\theta ServiceOffer\} \ndres partition) \cup \{\theta ServiceOffer' \mapsto partition\ \theta ServiceOffer\} \\
edges' = edges \\
edge\_properties' = edge\_properties \\
nodes' = nodes \\
\hline
\end{array}
$$

Error conditions for this behaviour arise when the pre-condition of the Φ*ModifyOffer* does not hold. This is when an attempt is made to modify an offer which is not present in the trading system. The trading system remains in the same state.

ModifyOfferError

$$
\begin{array}{|l}
\hline
\Xi TradingSystem \\
modified\_offer? : ServiceOffer \\
\hline
modified\_offer? \notin offers \\
\hline
\end{array}
$$

Finally, the *ModifyOffer* behaviour is defined in terms of *ModifyServiceOffer*, the framing schema Φ*ModifyOffer* (which identifies a particular offer to be modified) and the error conditions defined in *ModifyOfferError*. The components of Δ *ServiceOffer* are hidden so that they do not appear in the declaration part of *ModifyOffer*, in line with the usual Z conventions.

$$ModifyOffer \triangleq ((ModifyServiceOfferOK \land \Phi ModifyOffer) \lor ModifyOfferError) \setminus (service\_type, prop\_vals, service\_offer\_identifier, interface\_identifier, service\_type', prop\_vals', service\_offer\_identifier', interface\_identifier')$$

7.5.4 Add Edge

The dynamic schema AddEdge defines the behaviour associated with adding an edge to the trading graph. An edge is added between the two nodes supplied, and the new edge properties are associated with this edge. The set of service offers and the nodes within the trading system remain unchanged.

The pre-conditions of this dynamic schema are that both nodes exist in the trading system, and that no edge already exists between these two nodes in the same direction.

In Z, the *AddEdge* schema represents the corresponding behaviour.

AddEdgeOK

$$\Delta TradingSystem$$
$$node1?, node2? : Node$$
$$new\_edge\_properties? : \mathbb{P} Property$$

$$\{node1?, node2?\} \subseteq nodes$$
$$(node1?, node2?) \notin edges$$
$$edges' = edges \cup \{node1? \mapsto node2?\}$$
$$edge\_properties' = edge\_properties \cup \{(node1?, node2?) \mapsto new\_edge\_properties?\}$$
$$offers' = offers$$
$$nodes' = nodes$$
$$partition' = partition$$

If the pre-conditions of *AddEdgeOK* are not met, the state of the trading system remains unchanged.

AddEdgeError

$$\Xi TradingSystem$$
$$node1?, node2? : Node$$

$$node1? \notin nodes \lor$$
$$node2? \notin nodes \lor$$
$$(node1?, node2?) \in edges$$

AddEdge will succeed or fail, depending on the pre-conditions of the above Z schemata.

$$AddEdge \triangleq AddEdgeOK \lor AddEdgeError$$

7.5.5 Remove Edge

The dynamic schema RemoveEdge removes an edge from the trading graph. The set of service offers and nodes within the trading system remain unchanged by this dynamic schema. The properties associated with the edge are removed from the *edge_properties*.

The pre-conditions for this action are that the edge supplied is in the current set of edges.

In Z, the *RemoveEdge* schema represents the corresponding behaviour.

RemoveEdgeOK

$\Delta TradingSystem$
$old\_edge? : Node \times Node$

$old\_edge? \in edges$
$edges' = edges \setminus \{old\_edge?\}$
$edge\_properties' = \{old\_edge?\} \lhd edge\_properties$
$offers' = offers$
$nodes' = nodes$
$partition' = partition$

If the pre-conditions of *RemoveEdgeOK* are not met, the state of the trading system remains unchanged.

RemoveEdgeError

$\Xi TradingSystem$
$old\_edge? : Node \times Node$

$old\_edge? \notin edges$

RemoveEdge will succeed or fail depending on the pre-conditions of the above Z schemata.

$RemoveEdge \triangleq RemoveEdgeOK \lor RemoveEdgeError$

7.5.6 Modify Edge

The dynamic schema ModifyEdge modifies the properties associated with an edge. The old properties associated with the edge are replaced by the new properties. The set of service offers and set of nodes remains unchanged, as do the properties associated with all other edges.

The pre-condition for this action is that the supplied edge must exist.

In Z, the *ModifyEdge* schema represents the corresponding behaviour.

ModifyEdgeOK

$\Delta TradingSystem$
$edge? : Node \times Node$
$new\_edge\_properties? : \mathbb{P} Property$

$edge? \in edges$
$edge\_properties' = edge\_properties \oplus \{edge? \mapsto new\_edge\_properties?\}$
$edges' = edges$
$offers' = offers$
$nodes' = nodes$
$partition' = partition$

If the pre-conditions of the *ModifyEdgeOK* are not met, the state of the trading system remains unchanged.

ModifyEdgeError

$\Xi TradingSystem$
$edge? : Node \times Node$

$edge? \notin edges$

ModifyEdge will succeed or fail depending on the pre-conditions of the above Z schemata.

$ModifyEdge \triangleq ModifyEdgeOK \lor ModifyEdgeError$

7.5.7 Add Node

The dynamic schema AddNode describes the behaviour of adding a node to the trading system.

The pre-condition of this action is that the node to be added must not already exist within the trading system. Since the new node was not an element of *nodes*, and hence not in the range of the *partition* function, no existing offers are mapped into the new node. This implies that new nodes do not contain offers.

In Z, the *AddNode* schema represents the corresponding behaviour.

AddNodeOK

$\Delta TradingSystem$
$new\_node? : Node$

$new\_node? \notin nodes$
$offers' = offers$
$nodes' = nodes \cup \{ new\_node? \}$
$edges' = edges$
$edge\_properties' = edge\_properties$
$partition' = partition$

If the pre-conditions of *AddNodeOK* do not hold, the state of the trading system is unchanged.

AddNodeError

$\Xi TradingSystem$
$new\_node? : Node$

$new\_node? \in nodes$

AddNode will succeed or fail depending on the pre-conditions of the above Z schemata.

$AddNode \triangleq AddNodeOK \lor AddNodeError$

7.5.8 Remove Node

The dynamic schema RemoveNode defines the behaviour of removing a node from the trading system. The pre-conditions are that the node to be removed is present in the trading system, that it contains no service offers and that it no longer has edges to other nodes.

The computational behaviour may combine this dynamic schema with other dynamic schemata for removing service offers and edges.

In Z, the *RemoveNode* schema represents the corresponding behaviour.

RemoveNodeOK

$\Delta TradingSystem$
$old\_node? : Node$

$old\_node? \in nodes$
$old\_node? \notin ran\ partition$
$old\_node? \notin dom\ edges \cup ran\ edges$
$offers' = offers$
$nodes' = nodes \setminus \{old\_node?\}$
$edge\_properties' = edge\_properties$
$edges' = edges$
$partition' = partition$

If the pre-conditions of *RemoveNodeOK* do not hold, the state of the trading system is unchanged.

RemoveNodeError

$$\Xi TradingSystem$$
$$old\_node? : Node$$

$$old\_node? \notin nodes \lor$$
$$old\_node? \in ran\ partition \lor$$
$$old\_node? \in dom\ edges \cup ran\ edges$$

RemoveNode will succeed or fail depending on the pre-conditions of the above Z schemata.

$$RemoveNode \triangleq RemoveNodeOK \lor RemoveNodeError$$

7.5.9 Search

The dynamic schema Search is the behaviour which searches the trading system, restricted by some scoping criteria, and returns a set of service offers which satisfy some matching criteria. The action does not change the state of the trading system. Its output is a set of service offers. Its input is a search request.

Offers which satisfy the result of the search must satisfy the following conditions:

- they must have the correct service type or a subtype thereof;
- they must be contained in a node which is acceptable to both the requirements of the importer and the trading system's constraints, and is reachable from the starting point;
- they must match the importer's matching criteria;
- they must satisfy the trading system's matching constraints.

In Z, the *Search* schema represents the corresponding behaviour. The formal specification of Search represents the effect of matching criteria and search criteria as sets of service offers. The result of the search is obtained by simple set intersection. All offers which are returned must be reachable from the original starting point of the search. This property is expressed formally by stating that all such offers must be contained in a node which appears in the transitive closure of the *edges* relation.

SearchOK

$$\Xi TradingSystem$$
$$\Xi TradingSystemConstraints$$
$$SearchRequest?$$
$$starting\_point? : Node$$
$$search\_result! : \mathbb{P} ServiceOffer$$

$$starting\_point? \in nodes$$
$$let\ e: EdgeCriteria \bullet e \subseteq edges$$
$$partition\ (\!|search\_result!|\!) \subseteq$$
$$\{x : Node \mid (starting\_point?, x) \in e^{+}\}$$
$$search\_result! \subseteq$$
$$importer\_matching? \cap trader\_matching \cap importer\_scope? \cap trader\_scope$$

If an attempt is made to search starting from a node which does not exist, no service offers are returned and the state of the trading system is not changed.

SearchError

$$\Xi TradingSystem$$
$$\Xi TradingSystemConstraints$$
$$SearchRequest?$$
$$starting\_point? : Node$$
$$search\_result! : \mathbb{P} ServiceOffer$$

$$starting\_point? \notin nodes$$
$$search\_result! = \varnothing$$

The behaviour of Search is fully described by combining the above Z schemata.

$Search \triangleq SearchOK \vee SearchError$

7.5.10 Select

The dynamic schema Select is the behaviour which orders a set of service offers according to some preference criteria. This criteria is a combination of the importer's preference and the trading system's constraints. These constraints include aspects of the arbitration policy of the trading community. The trading system's constraints modify the preference expressed by the importer. The action does not change the state of the trader.

Selection

Search

selection! : **seq** *ServiceOffer*

---

selection! = trader_preference(*importer_preference*) (*search_result!*)

The formal Z specification uses the schema *Selection* to describe the application of preference criteria and constraints to the result of a search. *Select* removes the search result from the signature of *Selection*.

$Select \triangleq Selection \setminus (search\_result!)$

NOTE – In the computational viewpoint, the *Search* and *Select* behaviours are combined into the query operation.

8 Computational specification of the Trading Function

The scope of the computational specification is defined in RM-ODP Part 3: Architecture (see ITU-T Rec. X903 | ISO/IEC 10746-3).

In the computational viewpoint, the interfaces of a trader with its environment are visible. The computational specification of this Specification defines interface templates for computational interfces (client and server) that can be instantiated by a trader object.

To enable implementors to constrict conformance classes of traders (see clause 9) with different combinations of interfaces (and thus different functionality), the intefaces in this Specification are grouped into two categories:

- Functional interfaces, for grouping of operations based on the providion of functionality. The five functional interfaces to a trader are: Lookup; Register; Proxy; Link; and Admin. In addition, two auxilliary interfaces , Offer Iterator and Offer Id Iterator, are also specified.
- Abstract interfaces, for grouping of read-only attributes based on the required support for a functional interface. This Specification specifies the abstract interfaces: Support Attributes; Import Attributes; Link Attributes; and Trader Components.

The signatures for the operations of the Admin and Link interfaces are defined to support portability of Traders. Behaviour is specified for link management operations.

NOTE 1 – Internal administrative operations such as creation and deletion of trading interfaces are considered to be operations belonging to the generic Object Management Function and are not defined by this Specification.

A trader may be a client to several RM-ODP generic functions which are the subject of future standardisation. For operations at the interfaces of such server functions, neither signatures nor behaviour are specified by this Specification.

ODP-IDL (see ITU-T Rec. X.920 | ISO/IEC 14750) is used in this specification to express computational operation interface signatures. Use of this notation does not imply use of specific supporting mechanisms and protocols.

In addition to this computational specification, Annex A includes an ODP-IDL specification of the operational signatures in the CosTrading module and the CosTradingDynamic module, and Annex D includes an ODP-IDL specification of the operational signatures in the CosTradingRepos module.

NOTE 2 – The computational intefaces in this Specification are technically aligned with the OMG Trading Object Service.

8.1 Viewpoint correspondences

8.1.1 Correspondence with enterprise viewpoint

The policies expressed in the enterprise viewpoint are expressed in the computational viewpoint as operation parameters or trader attributes. Some of these operation parameters are expressed as constraints.

Each constraint can be expressed as a proposition that:

- a named property exists;
- a named property has a specified relationship to a stated value;
- the values of two named properties have a specified relationship.

Any constraint can be set to a default value of true. A constraint may also be a conjunction, a disjunction or a negation of other constraints.

Two action templates are identified for handling policies:

- Consult action – Determines what constraints for governing behaviour apply in order to adhere to the trader policy.
- Arbitrate action – Produces a resultant constraint for performing a given operation by combining the client's policy (expressed as a constraint or input parameter) and the trader policy (represented by a constraint and some attribute or property values).

8.1.2 Correspondence with information viewpoint

The information specification defines a directed graph where partitions of offers are placed at the nodes of the graph. The only constraint discernible from the information viewpoint is that, for a given direction, only one edge is allowed between any two nodes. Two edges, with opposite directions, are permitted between the same two nodes.

The computational specification defines trader objects, i.e. objects that provide a trading service at an interface. A trader object may use a trading service of another trader, i.e. it is linked to that other trader.

The relationship between these two viewpoint specifications is prescribed in this Specification. An information partition shall correspond to one and only one trader object. The possibility of many partitions corresponding to a single trader object is not allowed. Each trading service interface has a single partition, from which all offers associated with that trading service interface can be reached in an import search.

An edge in the information viewpoint graph connects partitions that correspond to different trader objects, and thus must itself correspond to a link between those two traders. The target partition is associated with the trading service interface pointed to by that link. All partitions corresponding to a given trader must either be associated with the trading service interface of that trader or be reachable, in the information graph, from such an associated partition.

8.2 Concepts and data types

8.2.1 Exporter and importer

8.2.1.1 Exporter

An exporter advertises a service with a trader. An exporter can be the service provider or it can advertise a service on behalf of another.

8.2.1.2 Importer

An importer uses a trader to search for services matching some criteria. An importer can be the potential service client or it can import a service on behalf of another.

8.2.2 Architecturaly neutral types and values

8.2.2.1 TypeCode

ODP-IDL does not include a primitive type for representing type descriptions. The ODP-IDL templates specified in this Specification use the term "TypeCode" to represent type descriptions.

When implementing this ODP Function using a particular ODP infrastructure, the term "TypeCode" must be defined using an appropriate type for representing type descriptions in that ODP infrastructure.

NOTE – For CORBA infrastructure, the term "TypeCode" should be defined as "CORBA::TypeCode".

8.2.2.2 Nil interface reference value

In this Specification, the term "nil" is used to denote the interface reference value of a non-existing interface instance.

8.2.3 Service types

8.2.3.1 Service type information

Associated with each traded service is a service type which represents the information needed to describe a service. It comprises:

- an interface type which defines the computational signature of the service interface; and
- zero or more named property types. Typically these represent behavioural, non-functional, and non-computational aspects that are not captured by the computational signature.

The property type defines the property value type, whether a property is mandatory, and whether a property is readonly. That is, associated with a property type is the triple of <name, type, mode>, where the modes are:

```
enum PropertyMode {
    PROP_NORMAL, PROP_READONLY,
    PROP_MANDATORY, PROP_MANDATORY_READONLY
};
```

A service type repository is used to hold the type information.

```
typedef Object TypeRepository;
```

A trader has an associated service type repository. However, this specification does not require use of any particular service type repository interface. The type repository interface may or may not be part of the trading object itself. A suitable interface is specified in Annex D.

Each service type in a repository is identified by a unique ServiceTypeName.

```
typedef Istring ServiceTypeName;
```

NOTE – The Istring typedef denotes the intention to support an international character set (not limited to Latin-1) in its use.

An exporter specifies the service type of the service it is advertising; an importer specifies the service type it is seeking.

Service types can be related in a hierarchy that reflects interface subtyping (e.g. by inheritance) and property type aggregation. This hierarchy provides the basis for deciding if a service of one type may be substituted for a service of another type. These considerations are described more fully in the following service type model.

8.2.3.2 Service type model

This subclause corresponds to the information viewpoint specification of service type subtyping rules in 7.2.3. These rules are expresed in this subclause using terms from the compuational language.

The service type model is illustrated by the following BNF:

```
service <ServiceTypeName> [: <BaseServiceTypeName> [, <BaseServiceTypeName>]* ] {
    interface <InterfaceTypeName>;
    [[mandatory] [readonly] property <IDLType> <PropertyName>;]*
};
```

The keyword “service” introduces a new ServiceTypeName. As the service type is visible to end users and not just to programmers, it is internationalizable.

The list of BaseServiceTypeNames lists those service types from which this service type is derived, which in turn defines where services of this service type can substitute for other service.

The “interface” keyword introduces the InterfaceTypeName for this service. It is related by equivalence or by derivation to the InterfaceTypeNames in each of the BaseServiceTypeNames.

The properties clause is a list of property declarations. Each property declaration is marked by the keyword “property” and may be preceded by mode attributes “mandatory” and/or “readonly”. A property declaration is completed by an IDLType and a PropertyName. A service must support all the properties of each of its base service types, they must have identical property value types, and they must not lose any property mode attributes.

The property mode attributes have the following connotations:

- Mandatory – An instance of this service type must provide an appropriate value for this property when exporting its service offer.
- Readonly – If an instance of this service type provides an appropriate value for this property when exporting its service offer, the value for this property may not be changed by a subsequent invocation of the Register::modify() operation.

The property strength graph is shown in Figure 2.

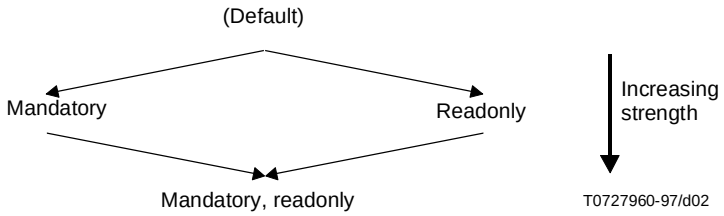


Figure 2 – Service property mode strength graph

Summarising, if a property is defined without any modifiers, it is optional (i.e. an offer of that service type is not required to provide a value for that property name, but if it does, it must be of the type specified in the service type), and the property value may be subsequently modified. The "mandatory" modifier indicates that a value must be provided, but that it may be subsequently modified. The "readonly" modifier indicates that the property is optional, but that once given a value, it may not be subsequently modified. Specifying both modifiers indicates that a value must be provided and that it may not be subsequently modified.

From the above discussion, one can state the rules for service type conformance; a service type β is a subtype of service type α, if, and only if:

- the interface type associated with β is either the same as, or is a subtype of, the interface type associated with α;
- all the properties defined in α are also defined in β;
- for all properties defined in both α and β, the mode of the property in β must be the same as, or stronger than, the mode of the property in α;
- all properties defined in β that are also defined in α shall have, in β, the same property value type as in α, or a subtype of the property value type in α.

8.2.4 Properties

Properties are <name, value> pairs. An exporter asserts values for properties of the service it is advertising. An importer can obtain these values about a service and constrain its search for appropriate offers based on the property values associated with such offers.

```
typedef Istring PropertyName;
typedef sequence<PropertyName> PropertyNameSeq;
typedef any PropertyValue;
struct Property {
        PropertyName name;
        PropertyValue value;
};
typedef sequence<Property> PropertySeq;

enum HowManyProps { none, some, all };
union SpecifiedProps switch ( HowManyProps ) {
        case some: PropertyNameSeq prop_names;
};
```

8.2.5 Service offers

A service offer is the information asserted by an exporter about the service it is advertising. It contains:

- the service type name;
- a reference to the interface that provides the service; and
- zero or more property values for the service.

An exporter must specify a value for all mandatory properties specified in the associated service type. In addition, an exporter can also nominate values for named properties that are not specified in the service type. In such case, the trader is not obliged to do property type checking.

```
struct Offer {
        Object reference;
        PropertySeq properties;
};
typedef sequence<Offer> OfferSeq;

struct OfferInfo {
        Object reference;
        ServiceTypeName type;
        PropertySeq properties;
};
```

8.2.5.1 Modifiable properties

The value of a property in a service offer can optionally be modified, if:

- the property mode is not readonly, whether optional or mandatory; and
- the trader supports the modify property functionality.

Such property values can be updated by explicit modify operations to the trader. An exporter can control a service offer to be non-modifiable by exporting services with service types that have read only properties. The modify operation will return a NotImplemented exception if a trader does not support the modify property functionality. An importer can also specify whether or not a trader should consider offers with modifiable properties during matching.

8.2.5.2 Dynamic properties

A service offer can optionally contain dynamic properties. The value for a dynamic property is not held within a trader, it is obtained on-demand from the interface of a dynamic property evaluator nominated by the exporter of the service. That is, a level of indirection is required to obtain the value for a dynamic property. The structure of a dynamic property value is:

```
exception DPEvalFailure {
        CosTrading::PropertyName name;
        TypeCode returned_type;
        any extra_info;
};

interface DynamicPropEval {

        any evalDP (
                in CosTrading::PropertyName name,
                in TypeCode returned_type,
                in any extra_info
        ) raises (
                DPEvalFailure
        );
};

struct DynamicProp {
        DynamicPropEval eval_if;
        TypeCode returned_type;
        any extra_info;
};
```

It contains the interface to the dynamic property evaluator, the data type of the returned dynamic property, and any extra implementation dependent information. The trader recognizes this structure and, when the value of the property is required, invokes the evalDP operation from the appropriate DynamicPropEval interface. The dynamic property evaluator interface has only one operation, whose signature is defined in this Specification for portability but its behaviour is not specified. The only restrictions imposed are that the property must not be readonly and that the trader must support the dynamic property functionality.

The use of such Properties has implications on the performance of a trader. An importer can specify whether or not a trader should consider offers with dynamic properties during matching.

8.2.6 Offer identifier

An offer identifier is returned to an exporter when a service offer is advertised in a trader. It identifies the exported service offer and is quoted by the exporter when withdrawing and modifying the offer (where supported). It only has meaning to the trader with which the service offer is registered.

```
typedef string OfferId;
typedef sequence<OfferId> OfferIdSeq;
```

8.2.7 Offer selection

The total service offer space for an offer selection is potentially very large, including offers from all linked traders. Logically the trader uses policies to identify the set S1 of service offers to examine. The service type and constraint is then applied to S1 to produce the set S2 that satisfy the service type and constraint. This is then ordered using preferences before returning the offers to the importer.

8.2.7.1 Standard constraint language

Importers select the set of service offers in which they have interest by use of service type and a constraint. The constraint is a well formed expression conforming to a constraint language.

This Recommendation | International Standard defines the standard, mandatory language which is necessary for interworking between traders. Annex B defines the syntax and the expressive power of the constraint language. This constraint language is used to write standard constraint expressions.

```
typedef Istring Constraint;
```

Its main features are:

Property Value Types	manipulation are restricted to int, float, fixed, boolean, Istring/string, Ichar/char types and sequences thereof. The character based types are ordered using the collating sequence in effect for the given character set. Types outside of this range can only be the subject of the “exists” operator.
Literals	in the Constraint are dynamically coerced as required for the Properties they are working with. Literals can contain Istring.
Operators	are comparison, boolean connective, “in_set”, substring, arithmetic operators, property existence.

If a proprietary constraint language (outside the scope of this Specification) is used, then the name and version of the constraint language used is placed between << >> at the start of the constraint expression, The remainder of the string is not interpreted by a trader that does not support the quoted proprietary constraint language.

8.2.7.2 Preferences

Preferences are logically applied to the set of offers matched by application of the service type, constraint expression, and various policies. The application of the preferences can be considered as the determination of the order to return matched offers to the importer.

```
typedef Istring Preference;
```

The preference string can be considered as being composed of two portions. The first portion can be any of the following (note that these keywords are case sensitive):

```
max  min  with  random  first
```

The interpretation for the second portion is dependent on the first portion; it may be empty. The following describes the preferences:

Preference	Description
max expression	The expression is numeric. The matched offers are returned in a descending order of the expression.
min expression	The expression is numeric. The matched offers are returned in an ascending order of the expression.
with expression	The expression is a Constraint expression. The matched offers are ordered such that those that are TRUE precede those that are FALSE.
random	The order of returning the matched offers is according to the following algorithm: select an offer at random from the set of matched offers, select another offer at random from the remaining set of matched offers, ..., select the single remaining offer.
first	The order of returned matched offers is at the offers are discovered.

If no preference is specified, then the default preference of first applies. No combinations of the preferences are permitted.

The expression associated with max, min, and with can refer to properties associated with the matching offers. When applying a preference expression to the set of offers that match the service type and constraint expression, the offer set is partitioned in two:

a) a group of offers for which the preference expression could be evaluated (ordered according to min, max, with); and

b) a group of offers for which the preference expression could not be evaluated (e.g. the preference expression refers to a property name that is optional for that service type).

The offers are returned to the importer in the order of first group in their preference order, followed by those in the second group.

If a proprietary preference language (outside the scope of this Specification) is used, then the name and version of the preference language used is placed between << >> at the start of the preference. The remainder of the string is not interpreted by a trader that does not support the quoted proprietary language.

8.2.7.3 Links

Links represent paths for propagation of queries from a source trader to a target trader. Each link corresponds to an edge in a trading graph, in which the vertices are traders. A link describes the knowledge that one trader has of another trading service that it uses. It also includes information of when to propagate or forward an operation to the target trader. A link has the following information associated with it:

– a Lookup interface provided by the target trader, which supports the query operation;

– a Register interface provided by the target trader, which supports the resolve operation;

– the link's default follow behaviour, which may be used and is passed on when an importer does not specify a link_follow_rule policy;

– the link's limiting follow behaviour, which overrides an importer's link_follow_rule if the importer's request exceeds the limit set by the link.

The OMG IDL for FollowOption is defined as:

```
enum FollowOption {
    local_only,
    if_no_local,
    always
};
```

where:

– "local_only" indicates that the link is never followed unless explicitly named in an operation;

– "if_no_local" indicates that the link is followed only if there are no local offers that satisfy the query; and

– "always" indicates that the link is always followed except when overriden by some policy.

These values are ordered as follows:

local_only < if_no_local < always

The OMG IDL for LinkInfo is defined as:

```
struct LinkInfo {
    Lookup target;
    Register target_reg;
    FollowOption def_pass_on_follow_rule ;
    FollowOption limiting_follow_rule;
};
```

The above information is set for each link when it is created. A link name is given to the link when it is created. The name uniquely identifies a link in a trader.

```
typedef Istring LinkName;
typedef sequence<LinkName> LinkNameSeq;
```

A link is unidirectional. Only the source trader is directly aware of a link; it is the source trader that supports the Link interface.

Additional information may be kept with a link to describe characteristics of the target trading service as perceived by the source trader.

8.2.7.4 Policies

Policies provide information to affect trader behaviour at run time. Policies are represented as name value pairs.

```
typedef string PolicyName; // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
    PolicyName name;
    PolicyValue value;
};
typedef sequence<Policy> PolicySeq;
```

Some policies cannot be overridden while other policies apply in the absence of further information and can be overridden. Policies can be grouped into two categories:

– those that scope the extent of a search;

– those that determine the functionality applied to an operation.

Different policies are associated with different roles in the performance of the trading function. These roles, which are used in the "Where" column of the following tables, are:

T = Trader

L = Link

I = Importer

These policies are further discussed in 8.2.7.5 through 8.2.7.9.

8.2.7.4.1 Standardized scoping policies

The following lists the standardized scoping policies:

Name	Where	IDLType	Description
def_search_card	T	unsigned long	Default upper bound of offers to be searched; used if no search_card is specified..
max_search_card	T	unsigned long	Maximum upper bound of offers to be searched.
search_card	I	unsigned long	Nominated upper bound of offers to be searched; will be overridden by max_search_card.
def_match_card	T	unsigned long	Default upper bound of matched offers to be ordered; used if no match_card is specified.
max_match_card	T	unsigned long	Maximum upper bound of matched offers to be ordered.
match_card	I	unsigned long	Nominated upper bound of offers to be ordered; will be overridden by max_match_card.
def_return_card	T	unsigned long	Default upper bound of ordered offers to be returned; used if no return_card is specified.
max_return_card	T	unsigned long	Maximun upper bound of ordered offers to be returned.
return_card	I	unsigned long	Nominated upper bound of ordered offers to be returned; will be overridden by max_return_card.
def_hop_count	T	unsigned long	Default upper bound of depth of links to be traversed if hop_count is not specified.
max_hop_count	T	unsigned long	Maximum upper bound of depth of links to be traversed.
hop_count	I	unsigned long	Nominated upper bound of depth of links to be traversed; will be overridden by the trader's max_hop_count.
def_pass_on_follow_rule	L	FollowOption	Default link follow behaviour to be passed on for a particular link if an importer does not specify its link_follow_rule. It must not exceed limiting_follow_rule.
limiting_follow_rule	L	FollowOption	Limiting link follow behaviour for a particular link.
max_link_follow_policy	T	FollowOption	Upper bound on the value of a link's limiting follow rule at the time of creation or modification of a link.
def_follow_policy	T	FollowOption	Default link follow behaviour for a particular trader.
max_follow_policy	T	FollowOption	Limiting link follow policy for all links of the trader – overrides both link and importer policies.
link_follow_rule	I	FollowOption	Nominated link follow behaviour; it will be overridden by the trader's max_follow_policy and the link's limiting_follow_rule.
starting_trader	I	TraderName	An importer scopes its search by nominating that the query operation starts at a remote trader; a trader is obliged to forward the request down a link even if the link behaviour is local_only.
request_id	I	OctetSeq	An identifier for a Query operation initiated by a source trader acting as importer on a link; a trader is not obliged to generate an id, but is oblidged to pass it down a link.
exact_type_match	I	boolean	If TRUE, only offers of exactly the service type specified by the importer are considered; if FALSE (or if unspecified), offers of any service type that conforms to the importer's service type are considered.

The ODP-IDL types for TraderName and OctetSeq are:

```
typedef LinkNameSeq TraderName;
typedef sequence<octet> OctetSeq;
```

The results received by an importer are affected by the scoping policies. The hop_count and link follow policies set the scope of the traders to visit. N1 is the total service offer space of those traders. Those offers that have conformant service type are gathered into the set N2; the actual size of N2 may be further restricted by the search cardinality policies. Constraints are applied to N2 to produce a set N3 of offers which satisfy both the service type and the constraints; N3 may be further restricted by the match cardinality policies. The set N3 is then ordered using preferences to produce the set N4. The final set of offers returned to the importer, N5, may be further reduced by the returned cardinality policies.

This is illustrated by the diagram shown in Figure 3, where |N1| >= |N2| >= |N3| = |N4| >= |N5|.

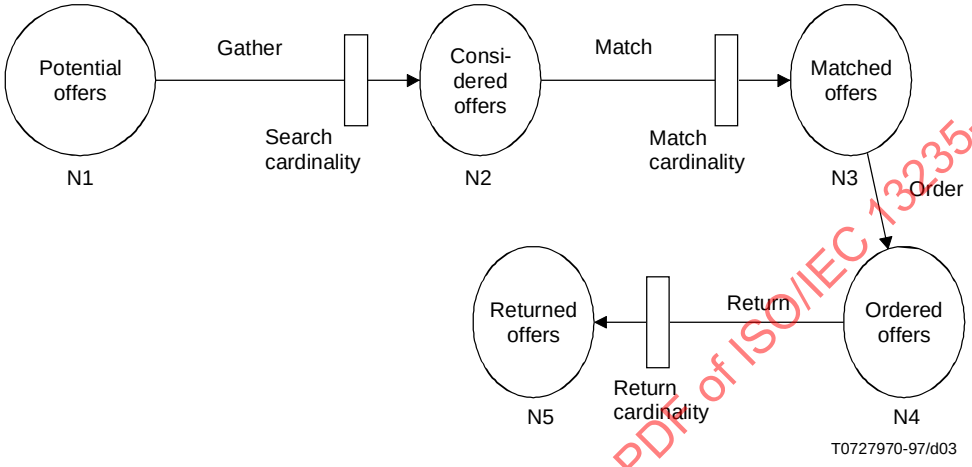


Figure 3 – Pipeline view of trader query steps and cardinality constraint application

8.2.7.4.2 Standardised capability supported policies

There are three capabilities: proxy offer, dynamic properties, and modify offers, that a trader may or may not wish to support. If a trader does not support a capability, then an importer can not override that capability with its policy parameter. However, if a trader supports a capability and an importer does not wish to consider offers that require such functionality, then the trader must respect the importer's wish.

The following lists the standardised policies related to functionality supported:

Name	Where	IDLType	Description
supports_modifiable_properties	T	boolean	Whether the trader supports property modification.
use_modifiable_properties	I	boolean	Whether to consider offers with modifiable properties in the search.
supports_dynamic_properties	T	boolean	Whether the trader supports dynamic properties.
use_dynamic_properties	I	boolean	Whether to consider offers with dynamic properties in the searchs.
supports_proxy_offers	T	boolean	Whether the trader supports proxy offers.
use_ proxy_offers	I	boolean	Whether to consider proxy offers in the search.

8.2.7.5 Trader policies

Policies can be set for a trader as a whole. Trader policies are defined as attributes of the trader object. They are initially specified when the trader is created, and can be modified/interrogated via the Admin interface. An importer can interrogate these trader policies via its Lookup interface. An exporter can interrogate a trader's functionality supported policies via its Register interface.

8.2.7.6 Link follow behaviour

Each link in a trader has its own follow behaviour policies. A trader has a limiting follow policy, max_follow_policy, that overrides all the links of that trader for any given query. Follow behaviour policies are specified for each link when a link is created. These policies, def_pass_on_follow_rule and limiting_follow_rule, can be interrogated/modified via the Link interface. The values they can have are limited by another trader policy, max_link_follow_policy, at the time of creation or modification. An importer can specify a link_follow_rule in a query. In the absence of an importer's link_follow_rule the trader's def_follow_policy is used.

After searching its local offers in response to a query, a trader must decide whether to propagate the query along its links and, if so, what value for the link_follow_rule to pass on in the policies argument.

The ODP-IDL for FollowOption is specified in 8.2.7.3.

The follow policy for a particular link is, therefore:

```
if the importer specified a link_follow_rule policy
    min(trader.max_follow_policy, link.limiting_follow_rule,
        query.link_follow_rule)

else
    min(trader.max_follow_policy, link.limiting_follow_rule,
        trader.def_follow_policy)
```

i.e. if this value is "if_no_local" and there were no local offers that match the query, the nested query is performed; if this value is "always", the nested query is performed.

If a nested query is permitted by the above rule, then the following logic determines the value for the "link_follow_rule" policy to be passed on to the linked trader.

```
if the importer specified a link_follow_rule policy
    pass on min(query.link_follow_rule, link.limiting_follow_rule,
                trader.max_follow_policy)

else
    pass on min(link.def_pass_on_follow_rule , trader.max_follow_policy)
```

8.2.7.7 Importer policies

An importer can specify zero or more importer policies in its policy parameter. If an importer policy is not specified, then the trader uses its default policy. If an importer policy exceeds the limiting policy values set by the trader, then the trader overrides the importer expectations with its limiting policy value.

If a starting_trader policy parameter is used, trader implementations shall place this policy parameter as the first element of the sequence when forwarding the query request to linked traders.

8.2.7.8 Exporter policies

There are no exporter policies specified in this Specification.

8.2.7.9 Link creation policies

At the time that a link is created, the default and limiting follow rules associated with the link are specified. These rules can be constrained by the max_link_follow_policy of the trader.

The trader first checks to see that the default rule is less than or equal to the limiting rule. If not, then an exception is raised. It then compar es the limiting rule against the trader's max_link_follow_policy, again raising an exception if the limiting rule is greater than the trader's max_link_follow_policy.

8.2.8 Interworking mechanisms

8.2.8.1 Link traversal control

The flexible nature of trader linkage allows arbitrary directed graphs of traders to be produced. This can introduce two types of problem:

- A single trader can be visited more than once during a search due to it appearing on more than one path (i.e. distinct set of connected edges) leading from a trader.
- Loops can occur – The most trivial example of this is where two previously disjoint trader spaces decide to join by exchanging links. This can result in the first trader propagating a query to the second and then having it returned immediately via the reverse link.

To ensure that a search does not enter into an infinite loop, a hop_count is used to limit the depth of links to propagate a search. The hop_count is decremented by one before propagating a query to other traders. The search propagation terminates at the trader when the hop_count reaches zero.

To avoid the unproductive revisiting of a particular trader while performing a query, a RequestId can be generated by a source trader for each query operation that it initiates for propagation to a target trader. The trader attribute of request_id_stem is used to form RequestId.

```
typedef sequence<octet> OctetSeq;
attribute OctetSeq request_id_stem;
```

A trader may wish to remember the RequestId of all recent interworking query operations that it has been asked to perform. When a query operation is received, such a trader checks this history and only processes the query if it is the operation's first appearance.

In order for this to work, the administrator for a set of federated traders must have initialized the respective request_id_stems to non-overlapping values.

The RequestId is passed in an importer's policy parameter on the query operation to the target trader. If the target trader does not support the use of the RequestId policy, the target trader need not process the RequestId but it must pass the RequestId onto the next linked trader if the search propagates further.

8.2.8.2 Federated query example

To propagate a query request in a trading graph, each source trader acts as a client to the Lookup interface of the target trader and passes its client's query operation to its target trader.

Figure 4 uses a sequential search example, to illustrate the modification of hop count parameter as a query request passes through a set of linked traders in a trading graph. We assume that the link follow policies in the traders will result in "always" follow behaviour:

a) A query request is invoked at the trading interface of T1 with an importer's hop count policy expressed as hop_count = 4. The trader scoping policy for T1 includes max_hop_count = 5. The resultant hop_count applied for the search (after the arbitration action that combines the trader policy and the importer policy) is hop_count = 4.

b) We assume that no match is found in T1 and the resulting follow policy is always. That is, T1 is to pass the request to T3. A modified importer hop_count policy of hop_count = 3 is used. The local trader scoping policy for T3 includes max_hop_count = 1 and the generation of T3_Request_id to avoid repeat or cyclic searches of the same traders. The resultant scoping policy applied for the search at T3 is hop_count = 1 and the T3_Request_id is stored.

c) Assuming that no match is found in T3 and that the resulting follow policy is always, the modified scoping parameter for the query request at T4 is: hop_count = 0 and request_id = T3_Request_id.

d) Assuming that no match is found in T4. Even though the max_hop_count = 4 for T4, the search is not propagated further. An unsuccessful search result will be passed back to T3, to T1, and finally to the user at T1.

Of course, if a query request is successfully completed at any of the traders on the linked search path, then the list of matched service offers will be returned to the original user; whether the query request is propagated through the remaining trading graph depends upon the link follow policies; in this case, where it is assumed to be always, the query will still visit all of the traders commensurate with the hop count policy.

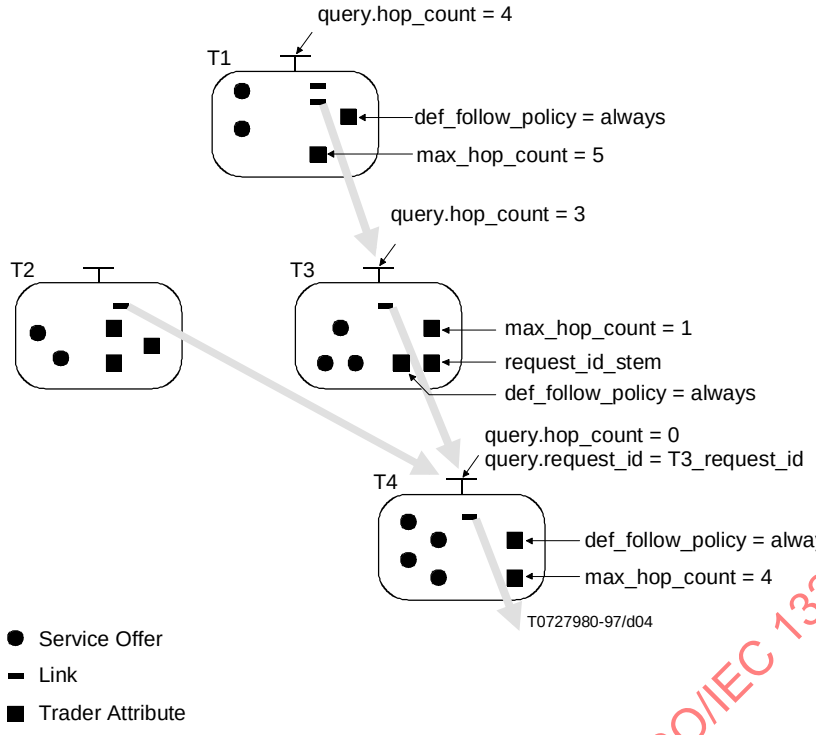


Figure 4 – Flow of a query through a trader graph

8.2.8.3 Proxy offers

A proxy offer is a cross between a service offer and a form of restricted link. It includes the service type and properties of a service offer and, as such, is matched in the same way. However, if the proxy offer matches the importer's requirements, rather than returning details of the offer, the query request (modified) is forwarded to the Lookup interface associated with the proxy offer.

```
typedef Istring ConstraintRecipe;

struct ProxyInfo {
        ServiceTypeName type;
        Lookup target;
        PropertySeq properties;
        boolean if_match_all;
        ConstraintRecipe recipe;
        PolicySeq policies_to_pass_on;
};
```

If an importer's query results in a match to a proxy offer, the trader holding the proxy offer performs a nested query on the trader hiding behind the proxy offer with the following parameters:

– The original type parameter is passed on unchanged.

– A new constraint parameter is constructed following the ConstraintRecipe associated with the proxy offer.

– The original preference parameter is passed on unchanged.

– A new policies parameter is constructed by appending the policies_to_pass_on associated with the proxy offer to the original policies parameter.

– The original desired_props parameter is passed on unchanged.

– The calling trader supplies a value of how_many that makes sense given its resource constraints.

Proxy offers are a convenient way to package the encapsulation of a legacy system of “objects” into the trading system. It permits clients to lookup these “objects” by matching the proxy offer; the nested call to the proxy trader, together with the rewritten constraint expression and the additional policies appended to the original policy parameter, permits the dynamic creation of a service instance which encapsulates the legacy object. Another possible use of proxies is for a service factory to be advertised as a proxy offer; the nested call to the factory causes a new instance of the particular service to be manufactured.

A query may have matched a proxy offer due to a particular value of a property associated with the proxy offer. It is mandatory that any offer returned by the proxy trader as a result of the nested query have the same value for that property so as not to violate the client’s expectations regarding the constraint.

A trader does not have to support the proxy offer functionality. However, if a trader supports such functionality, it must provide the Proxy interface for the export, withdraw, and describe of proxy offers. An importer can specify whether or not a trader should consider proxy offers during matching.

8.2.9 Trader attributes

Each trader has its own characteristics, policies for functionalities supported, and policies for scoping the extent of search. These characteristics and policies are defined as attributes to the trader. These attributes are:

Name	ODP-IDL Type	Description
def_search_card	unsigned long	Default upper bound of offers to be searched for a Query operation.
Max_search_card	unsigned long	Maximun upper bound of offers to be searched for a Query operation.
def_match_card	unsigned long	Default upper bound of matched offers to be ordered in applying a Preference criteria.
max_match_card	unsigned long	Maximun upper bound of matched offers to be ordered in applying a Preference criteria.
def_return_card	unsigned long	Default upper bound of ordered offers to be returned to an importer.
max_return_card	unsigned long	Maximun upper bound of ordered offers to be returned to an importer.
def_hop_count	unsigned long	Default upper bound of depth of links to be traversed.
max_hop_count	unsigned long	Maximum upper bound of depth of links to be traversed.
def_follow_policy	FollowOption	Default link-follow behaviour for a particular trader.
max_follow_policy	FollowOption	Limiting link follow policy for all links of the trader – overrides both link and importer policies.
max_link_follow_policy	FollowOption	Most permissive follow policy allowed when creating new links.
supports_modifiable_properties	boolean	Whether the trader supports property modification.
supports_dynamic_properties	boolean	Whether the trader supports dynamic properties.
supports _proxy_offers	boolean	Whether the trader supports proxy offers.
max_list	unsigned long	The max_list attribute determines the upper bound of any list returned by the trader, namely: the returned offers parameter in query, and the next-n operation in offer-iterator and offer-id iterator.
Type_repos	Repository	Interface to trader’s service repository.
request_id_stem	OctetSeq	Identification of the trader, to be used as the stem for the production of an id for a query request from one trader to another.

These attributes are initially specified when a trader is created, and can be modified/interrogated via the Admin Interface.

8.3 Exceptions

This Specification defines the exceptions raised by operations. Exceptions are parameterised to indicate the source of the error. The ODP-IDL segments below refer to some of the typedef's defined in clause 2.

When multiple exception conditions arise, only one exception is raised. The choice of exception to raise is implementation-dependent.

8.3.1 For CosTrading module

8.3.1.1 Exceptions used in more than one interface

```
exception UnknownMaxLeft {};

exception NotImplemented {};

exception IllegalServiceType {
        ServiceTypeName type;
};

exception UnknownServiceType {
        ServiceTypeName type;
};

exception IllegalPropertyName {
        PropertyName name;
};

exception DuplicatePropertyName {
        PropertyName name;
};

exception PropertyTypeMismatch {
        ServiceTypeName type;
        Property prop;
};

exception MissingMandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception IllegalConstraint {
        Constraint constr;
};

exception InvalidLookupRef {
        Lookup target;
};

exception IllegalOfferId {
        OfferId id;
};

exception UnknownOfferId {
        OfferId id;
};

exception ReadonlyDynamicProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception DuplicatePolicyName {
        PolicyName name;
};
```

8.3.1.2 Additional exceptions for Lookup interface

```
exception IllegalPreference {
        Preference pref;
};
```

```
exception IllegalPolicyName {
        PolicyName name;
};

exception PolicyTypeMismatch {
        Policy the_policy;
};

exception InvalidPolicyValue {
        Policy the_policy;
};
```

8.3.1.3 Additional exceptions for Register interface

```
exception InvalidObjectRef {
        Object ref;
};

exception UnknownPropertyName {
        PropertyName name;
};

exception InterfaceTypeMismatch {
        ServiceTypeName type;
        Object reference;
};

exception ProxyOfferId {
        OfferId id;
};

exception MandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception ReadonlyProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception NoMatchingOffers {
        Constraint constr;
};

exception IllegalTraderName {
        TraderName name;
};

exception UnknownTraderName {
        TraderName name;
};

exception RegisterNotSupported {
        TraderName name;
};
```

8.3.1.4 Additional exceptions for Link interface

```
exception IllegalLinkName {
        LinkName name;
};

exception UnknownLinkName {
        LinkName name;
};

exception DuplicateLinkName {
        LinkName name;
};

exception DefaultFollowTooPermissive {
        FollowOption def_pass_on_follow_rule ;
        FollowOption limiting_follow_rule;
};
```

```
exception LimitingFollowTooPermissive {
        FollowOption limiting_follow_rule;
        FollowOption max_link_follow_policy;
};
```

8.3.1.5 Additional exceptions for Proxy Offer interface

```
exception IllegalRecipe {
        ConstraintRecipe recipe;
};

exception NotProxyOfferId {
        OfferId id;
};
```

8.3.2 For CosTradingDynamic module

There is only a DynamicPropEval interface in this module, and the interface has only one operation which raises the exception:

```
exception DPEvalFailure {
        CosTrading::PropertyName name;
        TypeCode returned_type;
        any extra_info;
};
```

8.3.3 For CosTradingRepos module

There is only the ServiceTypeRepository interface in this module. The following interface-specific exceptions can be raised:

```
exception ServiceTypeExists {
        CosTrading::ServiceTypeName name;
};

exception InterfaceTypeMismatch {
        CosTrading::ServiceTypeName base_service;
        Identifier base_if;
        CosTrading::ServiceTypeName derived_service;
        Identifier derived_if;
};

exception HasSubTypes {
        CosTrading::ServiceTypeName the_type;
        CosTrading::ServiceTypeName sub_type;
};

exception AlreadyMasked {
        CosTrading::ServiceTypeName name;
};

exception NotMasked {
        CosTrading::ServiceTypeName name;
};

exception ValueTypeRedefinition {
        CosTrading::ServiceTypeName type_1;
        PropStruct definition_1;
        CosTrading::ServiceTypeName type_2;
        PropStruct definition_2;
};

exception DuplicateServiceTypeName {
        CosTrading::ServiceTypeName name;
};
```

8.4 Abstract interfaces

In order to enable the construction of traders with varying support for the different trader interfaces, this Specification defines several abstract interfaces from which each of the trading object service functional interfaces (Lookup, Register, Link, Proxy, and Admin) are derived. Each of these abstract interfaces are documented below.

8.4.1 TraderComponents

```
interface TraderComponents {

    readonly attribute Lookup lookup_if;
    readonly attribute Register register_if;
    readonly attribute Link link_if;
    readonly attribute Proxy proxy_if;
    readonly attribute Admin admin_if;
};
```

The functionality of a trader can be configured by composing the defined interfaces in one of a number of prescribed combinations. The composition is not modelled through inheritance but rather by multiple interfaces to an object. Given one of these interfaces a way of finding the other associated interfaces is needed. To facilitate this each trader functional interface is derived from the TraderComponents interface.

The TraderComponents interface contains five readonly attributes that each provide a way to get a specific object reference.

Access to these attributes must return a nil object reference if the trading service in question does not support that particular interface.

8.4.2 SupportAttributes

```
interface SupportAttributes {

    readonly attribute boolean supports_modifiable_properties;
    readonly attribute boolean supports_dynamic_properties;
    readonly attribute boolean supports_proxy_offers;
    readonly attribute TypeRepository type_repos;
};
```

In addition to the ability of a trader implementation to selectively choose which functional interfaces to support, a trader implementation may also choose not to support modifiable properties, dynamic properties, and/or proxy offers. The functionality supported by a trader implementation can be determined by querying the readonly attributes in this interface.

The type repository used by the trader implementation can also be obtained from this interface.

8.4.3 ImportAttributes

```
interface ImportAttributes {

    readonly attribute unsigned long def_search_card;
    readonly attribute unsigned long max_search_card;
    readonly attribute unsigned long def_match_card;
    readonly attribute unsigned long max_match_card;
    readonly attribute unsigned long def_return_card;
    readonly attribute unsigned long max_return_card;
    readonly attribute unsigned long max_list;
    readonly attribute unsigned long def_hop_count;
    readonly attribute unsigned long max_hop_count;
    readonly attribute FollowOption def_follow_policy;
    readonly attribute FollowOption max_follow_policy;
};
```

Each trader is configured with default and maximum values of certain cardinality and link follow constraints that apply to queries. The values for these constraints can be obtained by querying the attributes in this interface.

8.4.4 LinkAttributes

```
interface LinkAttributes {

    readonly attribute FollowOption max_link_follow_policy;
};
```

When a trader creates a new link or modifies an existing link the max_link_follow_policy attribute will determine the most permissive behaviour that the link will be allowed. The value for this constraint on link creation and modification can be obtained from this interface.

8.5 Functional interfaces

This subclause describes the five functional interfaces to a trading object service: Lookup, Register, Link, Admin, and Proxy. The two iterator interfaces needed for these functional interface are also described.

8.5.1 Lookup

```
interface Lookup : TraderComponents, SupportAttributes, ImportAttributes {

    typedef Istring Preference;

    enum HowManyProps { none, some, all };

    union SpecifiedProps switch ( HowManyProps ) {
        case some: PropertyNameSeq prop_names;
    };

    exception IllegalPreference {
        Preference pref;
    };

    exception IllegalPolicyName {
        PolicyName name;
    };

    exception PolicyTypeMismatch {
        Policy the_policy;
    };

    exception InvalidPolicyValue {
        Policy the_policy;
    };

    void query (
        in ServiceTypeName type,
        in Constraint constr,
        in Preference pref,
        in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        IllegalPreference,
        IllegalPolicyName,
        PolicyTypeMismatch,
        InvalidPolicyValue,
        IllegalPropertyName,
        DuplicatePropertyName,
        DuplicatePolicyName
    );
};
```

8.5.1.1 Query operation

8.5.1.1.1 Signature

```
void query (
        in ServiceTypeName type,
        in Constraint constr,
        in Preference pref,
        in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
```

```
) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        IllegalPreference,
        IllegalPolicyName,
        PolicyTypeMismatch,
        InvalidPolicyValue,
        IllegalPropertyName,
        DuplicatePropertyName,
        DuplicatePolicyName
);
```

8.5.1.1.2 Function

The query operation is the means by which an object can obtain references to other objects that provide services meeting its requirements.

The “type” parameter conveys the required service type. It is key to the central purpose of trading: to perform an introduction for future type safe interactions between importer and exporter. By stating a service type the importer implies the desired interface type and a domain of discourse for talking about properties of the service. If the string representation of the “type” does not obey the rules for service type identifiers, then an IllegalServiceType exception is raised. If the “type” is syntactically correct but is not recognized as a service type within the trading scope then an UnknownServiceType exception is raised.

The trader may return a service offer of a subtype of the “type” requested. Subtyping of service types is discussed in 8.2.3. A service subtype has all the mandatory properties of its supertypes. This ensures that what is a well-formed query for the “type” is also a well-formed query with respect to any subtypes. However, if the importer specifies the policy of exact_type_match = TRUE, then only offers with the exact (no subtype) service type requested are returned.

The constraint “constr” is the means by which the importer states those requirements of a service that are not captured in the signature of the interface. These requirements deal with the computational behaviour of the desired service, non-functional aspects, and non-computational aspects (such as the organization owning the objects that provide the service). An importer is always guaranteed that any returned offer satisfies the matching constraint at the time of import. If the “constr” does not obey the syntax rules for a legal constraint expression, as defined in Annex B, then an IllegalConstraint exception is raised.

The “pref” parameter is used to order those offers that match the “constr” so that the offers returned by the trader are in the order of greatest interest to the importer. If “pref” does not obey the syntax rules for a legal preference expression, then an IllegalPreference exception is raised.

The “policies” parameter allows the importer to specify how the search should be performed as opposed to what sort of services should be found in the course of the search. This can be viewed as parameterising the algorithms within the trader implementation. The “policies” are a sequence of name-value pairs. The names available to an importer depend on the implementation of the trader. However, some names are standardised where they effect the interpretation of other parameters or where they may impact linking and federation of traders. If a policy name in this parameter does not obey the syntactic rules for legal PolicyName’s, then an IllegalPolicyName exception is raised. If the type of the value associated with a policy differs from that specified in this Specification, then a PolicyTypeMismatch exception is raised. If subsequent processing of a PolicyValue yields any errors (e.g. the starting_trader policy value is malformed), then an InvalidPolicyValue exception is raised. If the same policy name is included two or more times in this parameter, the DuplicatePolicyName exception is raised.

The “desired_props” parameter defines the set of properties describing returned offers that are to be returned with the object reference. There are three possibilities: the importer wants none of the properties, all of the properties but without having to name them, and thirdly, some properties the names of which are provided. If any of the “desired_props” names do not obey the rules for identifiers, then an IllegalPropertyName exception is raised. If the same property name is included two or more times in this parameter, the DuplicatePropertyName exception is raised. The desired_props parameter may name properties which are not mandatory for the requested service type. If the named property is present in the matched service offer, it shall be returned. The desired_props parameter does not affect whether or not a service offer is returned. To avoid “missing” desired properties, the importer should specify “prop exists” in the constraint.

The returned offers are passed back in one of two ways (or a combination of both). The “offers” return result conveys a list of offers and the “offer_itr” is a reference to an interface at which offers can be obtained. The “how_many” parameter states how many offers are to be returned via the “offers” result, any remaining offers are available via the iterator interface. If the “how_many” exceeds the number of offers to be returned, then the “offer_itr” will be nil.

If any cardinality or other limits were applied by one or more traders in responding to a particular query, then the "limits_applied" parameter will contain the names of the policies which limited the query. The sequence of names returned in "limits_applied" from any federated or proxy queries must be concatenated onto the names of limits applied locally and returned.

8.5.1.1.3 Importer policy specifications

```
struct LookupPolicies {
        unsigned long search_card;
        unsigned long match_card;
        unsigned long return_card;
        boolean use_modifiable_properties;
        boolean use_dynamic_properties;
        boolean use_proxy_offers;
        TraderName starting_trader;
        FollowOption link_follow_rule;
        unsigned long hop_count;
        boolean exact_type_match;
        OctetSeq request_id
};
```

The "search_card" policy indicates to the trader the maximum number of offers it should consider when looking for type conformance and constraint expression match. The lesser of this value and the trader's max_search_card attribute is used by the trader. If this policy is not specified, then the value of the trader's def_search_card attribute is used.

The "match_card" policy indicates to the trader the maximum number of matching offers to which the preference specification should be applied. The lesser of this value and the trader's max_match_card attribute is used by the trader. If this policy is not specified, then the value of the trader's def_match_card attribute is used.

The "return_card" policy indicates to the trader the maximum number of matching offers to return as a result of this query. The lesser of this value and the trader's max_return_card attribute is used by the trader. If this policy is not specified, then the value of the trader's def_return_card attribute is used.

The "use_modifiable_properties" policy indicates whether the trader should consider offers which have modifiable properties when constructing the set of offers to which type conformance and constraint processing should be applied. If the value of this policy is TRUE, then such offers will be included; if FALSE, they will not. If this policy is not specified, such offers will be included.

The "use_dynamic_properties" policy indicates whether the trader should consider offers which have dynamic properties when constructing the set of offers to which type conformance and constraint processing should be applied. If the value of this policy is TRUE, then such offers will be included; if FALSE, they will not. If this policy is not specified, such offers will be included.

The "use_proxy_offers" policy indicates whether the trader should consider proxy offers when constructing the set of offers to which type conformance and constraint processing should be applied. If the value of this policy is TRUE, then such offers will be included; if FALSE, they will not. If this policy is not specified, such offers will be included.

The "starting_trader" policy facilitates the distribution of the trading service itself. It allows an importer to scope a search by choosing to explicitly navigate the links of the trading graph. If the policy is used in a query invocation, it is recommended that it be the first policy-value pair; this facilitates an optimal forwarding of the query operation. A "policies" parameter need not include a value for the "starting_trader" policy. Where this policy is present, the first name component is compared against the name held in each link. If no match is found the InvalidPolicyValue exception is raised. Otherwise, the trader invokes query() on the Lookup interface held by the named link, but passing the "starting_trader" policy with the first component removed.

The "link_follow_rule" policy indicates how the client wishes links to be followed in the resolution of its query. See the discussion in 8.2.7 for details.

The "hop_count" policy indicates to the trader the maximum depth of hops across federation links that should be tolerated in the resolution of this query. The hop_count at the current trader is determined by taking the minimum of the trader's max_hop_count attribute and the importer's hop_count policy, if provided, or the trader's def_hop_count attribute if it is not. If the resulting value is zero, then no federated queries are permitted. If it is greater than zero, then it must be decremented before passing on to a federated trader.

The "exact_type_match" policy indicates to the trader whether the importer's service type must exactly match an offer's service type; if not (and by default), then any offer of a type conformant to the importer's service type is considered.

The "request_id" policy indicates to the trader the identifier of a Query operation that is initiated by a source trader acting as importer on a link. The id is generated using the trader attribute request_id_stem. A trader is not obliged to generate such an id for a query operation for another trader, but it is obliged to pass it down a link to another trader.

8.5.2 Offer Iterator

8.5.2.1 Signature

```
interface OfferIterator {

    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );

    boolean next_n (
        in unsigned long n,
        out OfferSeq offers
    );

    void destroy ();
};
```

8.5.2.2 Functions of offer iterator operations

The OfferIterator interface is used to return a set of service offers from the query operation by enabling the service offers to be extracted by successive operations on the OfferIterator interface.

The next_n operation returns a set of service offers in the output parameter “offers”. The operation returns n service offers if there are at least n service offers remaining in the iterator. If there are fewer than n service offers in the iterator, then all remaining service offers are returned. The actual number of service offers returned can be determined from the length of the “offers” sequence. The next_n operation returns TRUE if there are further service offers to be extracted from the iterator. It returns FALSE if there are no further service offers to be extracted.

The max_left operation returns the number of service offers remaining in the iterator. The exception UnknownMaxLeft is raised if the iterator cannot determine the remaining number of service offers (e.g. if the iterator determines its set of service offers through lazy evaluation).

The destroy operation destroys the iterator. No further operations can be invoked on an iterator after it has been destroyed.

8.5.3 Register

```
interface Register : TraderComponents, SupportAttributes {

    struct OfferInfo {
        Object reference;
        ServiceTypeName type;
        PropertySeq properties;
    };

    exception InvalidObjectRef {
        Object ref;
    };

    exception UnknownPropertyName {
        PropertyName name;
    };

    exception InterfaceTypeMismatch {
        ServiceTypeName type;
        Object reference;
    };

    exception ProxyOfferId {
        OfferId id;
    };

    exception MandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
    };

    exception ReadonlyProperty {
        ServiceTypeName type;
        PropertyName name;
    };
```

```
exception NoMatchingOffers {
        Constraint constr;
};

exception IllegalTraderName {
        TraderName name;
};

exception UnknownTraderName {
        TraderName name;
};

exception RegisterNotSupported {
        TraderName name;
};

OfferId export (
        in Object reference,
        in ServiceTypeName type,
        in PropertySeq properties
) raises (
        InvalidObjectRef,
        IllegalServiceType,
        UnknownServiceType,
        InterfaceTypeMismatch,
        IllegalPropertyName, // e.g. prop_name = "<foo-bar"
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        DuplicatePropertyName
);

void withdraw (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);

OfferInfo describe (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);

void modify (
        in OfferId id,
        in PropertyNameSeq del_list,
        in PropertySeq modify_list
) raises (
        NotImplemented,
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId,
        IllegalPropertyName,
        UnknownPropertyName,
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MandatoryProperty,
        ReadonlyProperty,
        DuplicatePropertyName
);
```

```
        void withdraw_using_constraint (
                in ServiceTypeName type,
                in Constraint constr
        ) raises (
                IllegalServiceType,
                UnknownServiceType,
                IllegalConstraint,
                NoMatchingOffers
        );

        Register resolve (
                in TraderName name
        ) raises (
                IllegalTraderName,
                UnknownTraderName,
                RegisterNotSupported
        );
};
```

8.5.3.1 Export operation

8.5.3.1.1 Signature

```
        OfferId export (
                in Object reference,
                in ServiceTypeName type,
                in PropertySeq properties
        ) raises (
                InvalidObjectRef,
                IllegalServiceType,
                UnknownServiceType,
                InterfaceTypeMismatch,
                IllegalPropertyName, // e.g. prop_name = "<foo-bar"
                PropertyTypeMismatch,
                ReadonlyDynamicProperty,
                MissingMandatoryProperty,
                DuplicatePropertyName
        );
```

8.5.3.1.2 Function

The export operation is the means by which a service is advertised, via a trader, to a community of potential importers. The OfferId returned is the handle with which the exporter can identify the exported offer when attempting to access it via other operations. The OfferId is only meaningful in the context of the trader that generated it.

The “reference” parameter is the information that enables a client to interact with a remote server. If a trader implementation chooses to consider certain types of object references (e.g. a nil object reference) to be un-exportable, then it may return the InvalidObjectRef exception in such cases.

The “type” parameter identifies the service type, which contains the interface type of the “reference” and a set of named property types that may be used in further describing this offer, i.e. it restricts what is acceptable in the properties parameter. If the string representation of the “type” does not obey the rules for identifiers, then an IllegalServiceType exception is raised. If the “type” is syntactically correct but a trader is able to unambiguously determine that it is not a recognized service type, then an UnknownServiceType exception is raised. If the trader can determine that the interface type of the “reference” parameter is not a subtype of the interface type specified in “type”, then an InterfaceTypeMismatch exception is raised.

The “properties” parameter is a list of named values that conform to the property value types defined for those names. They describe the service being offered. This description typically covers behavioural, non-functional and non-computational aspects of the service. If any of the property names do not obey the syntax rules for PropertyNames, then an IllegalPropertyName exception is raised. If the type of any of the property values is not the same as the declared type (declared in the service type), then a PropertyTypeMismatch exception is raised. If an attempt is made to assign a dynamic property value to a readonly property, then the ReadonlyDynamicProperty exception is raised. If the

"properties" parameter omits any property declared in the service type with a mode of mandatory, then a MissingMandatoryProperty exception is raised. If two or more properties with the same property name are included in this parameter, the DuplicatePropertyName exception is raised.

8.5.3.2 Withdraw operation

8.5.3.2.1 Signature

```
void withdraw (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);
```

8.5.3.2.2 Function

The withdraw operation removes the service offer from the trader, i.e. after withdraw, the offer can no longer be returned as the result of a query. The offer is identified by the "id" parameter which was originally returned by export. If the string representation of "id" does not obey the rules for offer identifiers, then an IllegalOfferId exception is raised. If the "id" is legal but there is no offer within the trader with that "id", then an UnknownOfferId exception is raised. If the "id" identifies a proxy offer rather than an ordinary offer, then a ProxyOfferId exception is raised.

8.5.3.3 Describe operation

8.5.3.3.1 Signature

```
OfferInfo describe (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);
```

8.5.3.3.2 Function

The describe operation returns the information about an offered service that is held by the trader. It comprises the "reference" of the offered service, the "type" of the service offer and the "properties" that describe this offer of service. The offer is identified by the "id" parameter which was originally returned by export. If the string representation of "id" does not obey the rules for object identifiers, then an IllegalOfferId exception is raised. If the "id" is legal but there is no offer within the trader with that "id", then an UnknownOfferId exception is raised. If the "id" identifies a proxy offer rather than an ordinary offer, then a ProxyOfferId exception is raised.

8.5.3.4 Modify operation

8.5.3.4.1 Signature

```
void modify (
        in OfferId id,
        in PropertyNameSeq del_list,
        in PropertySeq modify_list
) raises (
        NotImplemented,
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId,
        IllegalPropertyName,
        UnknownPropertyName,
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MandatoryProperty,
        ReadonlyProperty,
        DuplicatePropertyName  );
```

8.5.3.4.2 Function

The modify operation is used to change the description of a service as held within a service offer. The object reference and the service type associated with the offer cannot be changed. This operation may:

a) add new (non-mandatory) properties to describe an offer;

b) change the values of some existing (not readonly) properties;

c) delete existing (neither mandatory nor readonly) properties.

The modify operation either succeeds completely or it fails completely.

The offer is identified by the "id" parameter which was originally returned by export. If the string representation of "id" does not obey the rules for offer identifiers, then an IllegalOfferId exception is raised. If the "id" is legal but there is no offer within the trader with that "id", then an UnknownOfferId exception is raised. If the "id" identifies a proxy offer rather than an ordinary offer, then a ProxyOfferId exception is raised.

The "del_list" parameter gives the names of the properties that are no longer to be recorded for the identified offer. Future query and describe operations will not see these properties. If any of the names within the "del_list" do not obey the rules for PropertyName's, then an IllegalPropertyName exception is raised. If a "name" is legal but there is no property for the offer with that "name", then an UnknownPropertyName exception is raised. If the list includes a property that has a mandatory mode, then the MandatoryProperty exception is raised. If the same property name is included two or more times in this parameter, the DuplicatePropertyName exception is raised.

The "modify_list" parameter gives the names and values of properties to be changed. If the property is not in the offer, then the modify operation adds it. The modified (or added) property values are returned in future query and describe operations in place of the original values. If any of the names within the "modify_list" do not obey the rules for PropertyName's, then an IllegalPropertyName exception is raised. If the list includes a property that has a readonly mode, then the ReadonlyProperty exception is raised unless that readonly property is not currently recorded for the offer. The ReadonlyDynamicProperty exception is raised if an attempt is made to assign a dynamic property value to a readonly property. If the value of any modified property is of a type that is not the same as the type expected, then the PropertyTypeMismatch exception is raised. If two or more properties with the same property name are included in this argument, the DuplicatePropertyName exception is raised.

The NotImplemented exception shall be raised if, and only if, the supports_modifiable_properties attribute yields FALSE.

It is not possible to change the service type of an offer or the object reference of the service. This has to be achieved by withdrawing and then re-exporting. The purpose of modify is to change the description of the offered service whilst preserving the OfferId. This might be important where the OfferId has been propagated around a community of objects.

8.5.3.5 Withdraw Using Constraint operation

8.5.3.5.1 Signature

```
void withdraw_using_constraint (
        in ServiceTypeName type,
        in Constraint constr
) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        NoMatchingOffers
);
```

8.5.3.5.2 Function

The withdraw_using_constraint operation withdraws a set of offers from within a single trader. This set is identified in the same way as a query operation identifies a set of offers to be returned to an importer.

The "type" parameter conveys the required service type. Each offer of the specified type will have the constraint expression applied to it. If it matches the constraint expression, then the offer will be withdrawn.

If "type" does not obey the rules for service types, then an IllegalServiceType exception is raised. If the "type" is syntactically correct but is not recognized as a service type by the trader, then an UnknownServiceType exception is raised.

The constraint "constr" is the means by which the client restricts the set of offers to those that are intended for withdrawal. If "constr" does not obey the syntax rules for a constraint, then an IllegalConstraint exception is raised. If the constraint fails to match with any offer of the specified service type, then a NoMatchingOffers exception is raised.

8.5.3.6 Resolve operation

8.5.3.6.1 Signature

```
Register resolve (
        in TraderName name
) raises (
        IllegalTraderName,
        UnknownTraderName,
        RegisterNotSupported
);
```

8.5.3.6.2 Function

This operation is used to resolve a context relative name for another trader. In particular, it is used when exporting to a trader that is known by a name rather than by an interface reference. The client provides the name, which will be a sequence of name components. If the content of the parameter cannot yield legal syntax for the first component, then the IllegalTraderName exception is raised. Otherwise, the first name component is compared against the name held in each link. If no match is found, or the trader does not support links, the UnknownTraderName exception is raised. Otherwise, the trader obtains the register_if, held as part of the matched link. If the Register interface reference is not nil, then the trader binds to the Register interface and invokes resolve but passes the TraderName with the first component removed; if it is nil, then the RegisterNotSupported exception is raised. When a trader is able to match the first name component leaving no residual name, then that trader returns the reference for the Register interface for that linked trader. In unwinding the recursion, intermediate traders return the Register interface reference to their client (another trader).

8.5.4 Offer Id Iterator

8.5.4.1 Signature

```
interface OfferIdIterator {

        unsigned long max_left (
        ) raises (
                UnknownMaxLeft
        );

        boolean next_n (
                in unsigned long n,
                out OfferIdSeq ids
        );

        void destroy ();
};
```

8.5.4.2 Functions of Offer Id Iterator operations

The OfferIdIterator interface is used to return a set of offer identifiers from the list_offers operation and the list_proxies operation in the Admin interface by enabling the offer identifiers to be extracted by successive operations on the OfferIdIterator interface.

The next_n operation returns a set of offer identifiers in the output parameter “ids”. The operation returns n offer identifiers if there are at least n offer identifiers remaining in the iterator. If there are fewer than n offer identifiers in the iterator, then all remaining offer identifiers are returned. The actual number of offer identifiers returned can be determined from the length of the “ids” sequence. The next_n operation returns true if there are further offer identifiers to be extracted from the iterator. It returns false if there are no further offer identifiers to be extracted.

The max_left operation returns the number of offer identifiers remaining in the iterator. The exception UnknownMaxLeft is raised if the iterator cannot determine the remaining number of offer identifiers (e.g. if the iterator determines its set of offer identifiers through lazy evaluation).

The destroy operation destroys the iterator. No further operations can be invoked on an iterator after it has been destroyed.

8.5.5 Admin

```
interface Admin : TraderComponents, SupportAttributes, ImportAttributes,
                  LinkAttributes {

        typedef sequence<octet> OctetSeq;

        readonly attribute OctetSeq request_id_stem;
```

```
        unsigned long set_def_search_card (in unsigned long value);
        unsigned long set_max_search_card (in unsigned long value);

        unsigned long set_def_match_card (in unsigned long value);
        unsigned long set_max_match_card (in unsigned long value);

        unsigned long set_def_return_card (in unsigned long value);
        unsigned long set_max_return_card (in unsigned long value);

        unsigned long set_max_list (in unsigned long value);

        boolean set_supports_modifiable_properties (in boolean value);
        boolean set_supports_dynamic_properties (in boolean value);
        boolean set_supports_proxy_offers (in boolean value);

        unsigned long set_def_hop_count (in unsigned long value);
        unsigned long set_max_hop_count (in unsigned long value);

        FollowOption set_max_follow_policy (in FollowOption policy);
        FollowOption set_def_follow_policy (in FollowOption policy);

        FollowOption set_max_link_follow_policy (in FollowOption policy);

        TypeRepository set_type_repos (in TypeRepository repository);

        OctetSeq set_request_id_stem (in OctetSeq stem);

        void list_offers (
                in unsigned long how_many,
                out OfferIdSeq ids,
                out OfferIdIterator id_itr
        ) raises (
                NotImplemented
        );

        void list_proxies (
                in unsigned long how_many,
                out OfferIdSeq ids,
                out OfferIdIterator id_itr
        ) raises (
                NotImplemented
        );
};
```

8.5.5.1 Attributes and Set operations

The admin interface enables the values of the trader attributes to be read and written. All attributes are defined as readonly in either SupportAttributes, ImportAttributes, LinkAttributes, or Admin. To set the trader “attribute” to a new value, set_<attribute_name> operations are defined in Admin. Each of these set operations returns the previous value of the attribute as its function value.

If the admin interface operation set_support_proxy_offers is invoked with a value set to FALSE, for a trader which supports the proxy_interface, the set_support_proxy_offer value does not affect the function of operations in the proxy interface. However, in this case, the effect of the support_proxy_offers value being set to FALSE has the effect of making any proxy offers exported via the proxy intrface for that trader not being available to satisfy queries on that trader’s lookup interface.

8.5.5.2 List Offers operation

8.5.5.2.1 Signature

```
        void list_offers (
                in unsigned long how_many,
                out OfferIdSeq ids,
                out OfferIdIterator id_itr
        ) raises (
                NotImplemented
        );
```

8.5.5.2.2 Function

The list_offers operation allows the administrator of a trader to perform housekeeping by obtaining a handle on each of the offers within a trader, e.g. for garbage collection, etc. Only the identifiers of ordinary offers are returned, identifiers of proxy offers are not returned via this operation. If the trader does not support the Register interface, the NotImplemented exception is raised.

The returned identifiers are passed back in one of two ways (or a combination of both). The “ids” return result conveys a list of offer identifiers and the “id_itr” is a reference to an interface at which additional offer identities can be obtained. The “how_many” parameter states how many identifiers are to be returned via the “ids” result; any remaining are available via the iterator interface. If the “how_many” exceeds to the number of offers held in the trader, then the “id_itr” is nil.

8.5.5.3 List Proxies operation

8.5.5.3.1 Signature

```
void list_proxies (
        in unsigned long how_many,
        out OfferIdSeq ids,
        out OfferIdIterator id_itr
) raises (
        NotImplemented
);
```

8.5.5.3.2 Function

The list_proxies operation returns the set of offer identifiers for proxy offers held by a trader. At most “how_many” offer identifiers are returned via “ids”. If there are more than “how_many” offer identifiers, the remainder are returned via the “id_itr” iterator. If there are only “how_many” or fewer offer identifiers, the id_itr is nil. If the trader does not support the Proxy interface, the NotImplemented exception is raised.

8.5.6 Link

```
interface Link : TraderComponents, SupportAttributes,
                                LinkAttributes {

    struct LinkInfo {
        Lookup target;
        Register target_reg;
        FollowOption def_pass_on_follow_rule ;
        FollowOption limiting_follow_rule;
    };

    exception IllegalLinkName {
        LinkName name;
    };

    exception UnknownLinkName {
        LinkName name;
    };

    exception DuplicateLinkName {
        LinkName name;
    };

    exception DefaultFollowTooPermissive {
        FollowOption def_pass_on_follow_rule ;
        FollowOption limiting_follow_rule;
    };

    exception LimitingFollowTooPermissive {
        FollowOption limiting_follow_rule;
        FollowOption max_link_follow_policy;
    };

    void add_link (
        in LinkName name,
        in Lookup target,
        in FollowOption def_pass_on_follow_rule ,
        in FollowOption limiting_follow_rule
```

```
        ) raises (
            IllegalLinkName,
            DuplicateLinkName,
            InvalidLookupRef, // e.g. nil
            DefaultFollowTooPermissive,
            LimitingFollowTooPermissive
        );

        void remove_link (
            in LinkName name
        ) raises (
            IllegalLinkName,
            UnknownLinkName
        );

        LinkInfo describe_link (
            in LinkName name
        ) raises (
            IllegalLinkName,
            UnknownLinkName
        );

        LinkNameSeq list_links ();

        void modify_link (
            in LinkName name,
            in FollowOption def_pass_on_follow_rule ,
            in FollowOption limiting_follow_rule
        ) raises (
            IllegalLinkName,
            UnknownLinkName,
            DefaultFollowTooPermissive,
            LimitingFollowTooPermissive
        );
    };
```

8.5.6.1 Add_Link operation

8.5.6.1.1 Signature

```
        void add_link (
            in LinkName name,
            in Lookup target,
            in FollowOption def_pass_on_follow_rule ,
            in FollowOption limiting_follow_rule
        ) raises (
            IllegalLinkName,
            DuplicateLinkName,
            InvalidLookupRef, // e.g. nil
            DefaultFollowTooPermissive,
            LimitingFollowTooPermissive
        );
```

8.5.6.1.2 Function

The add_link operation allows a trader subsequently to use the service of another trader in the performance of its own trading service operations.

The “name” parameter is used in subsequent link management operations to identify the intended link. If the parameter is not legally formed, then the IllegalLinkName exception is raised. An exception of DuplicateLinkName is raised if the link name already exists. The link name is also used as a component in a sequence of name components in naming a trader for resolving or forwarding operations. The sequence of context relative link names provides a path to a trader.

The “target” parameter identifies the Lookup interface at which the trading service provided by the target trader can be accessed. Should the Lookup interface parameter be nil, then an exception of InvalidLookupRef is raised. The target interface is used to obtain the associated Register interface, which will be subsequently returned as part of a describe_link operation and invoked as part of a resolve operation.

The “def_pass_on_follow_rule” parameter specifies the default link behaviour for the link if no link behaviour is specified on an importer’s query request. If the “def_pass_on_follow_rule” exceeds the “limiting_follow_rule” specified in the next parameter, then a DefaultFollowTooPermissive exception is raised.

The "limiting_follow_rule" parameter specifies the most permissive link follow behaviour that the link is willing to tolerate. The exception LimitingFollowTooPermissive is raised if this parameter exceeds the trader's attribute of "max_link_follow_policy" at the time of the link's creation.

NOTE – It is possible for a link's "limiting_follow_rule" to exceed the trader's "max_link_follow_policy" later in the life of a link, as it is possible that the trader could set its "max_link_follow_policy" to a more restrictive value after the creation of the link.

8.5.6.2 Remove Link operation

8.5.6.2.1 Signature

```
void remove_link (
        in LinkName name
) raises (
        IllegalLinkName,
        UnknownLinkName
);
```

8.5.6.2.2 Function

The remove_link operation removes all knowledge of the target trader. The target trader cannot subsequently be used to resolve, forward, or propagate trading operations, from this trader.

The "name" parameter identifies the link to be removed. The exception IllegalLinkName is raised if the link is poorly formed and the UnknownLinkName exception is raised if the named link is not in the trader.

8.5.6.3 Describe Link operation

8.5.6.3.1 Signature

```
LinkInfo describe_link (
        in LinkName name
) raises (
        IllegalLinkName,
        UnknownLinkName
);
```

8.5.6.3.2 Function

The describe_link operation returns information on a link held in the trader.

The "name" parameter identifies the link whose description is required. For a malformed link name, the exception IllegalLinkName is raised. An UnknownLinkName exception is raised if the named link is not found in the trader.

The operation returns a LinkInfo structure comprising: the Lookup interface of the target trading service, the Register interface of the target trading service, and the default, as well as, the limiting follow behaviour of the named link. If the target service does not support the Register interface, then that field of the LinkInfo structure is nil.

NOTE – Given the description of the Register::resolve() operation in 8.5.3.6, implementations may opt for determining the Register interface when add_link is called and storing that information statically with the rest of the link state.

8.5.6.4 List Links operation

8.5.6.4.1 Signature

```
LinkNameSeq list_links ();
```

8.5.6.4.2 Function

The list_links operation returns a list of the names of all trading links within the trader. The names can subsequently be used for other management operations, such as describe_link or remove_link.

8.5.6.5 Modify Link operation

8.5.6.5.1 Signature

```
void modify_link (
        in LinkName name,
        in FollowOption def_pass_on_follow_rule ,
        in FollowOption limiting_follow_rule
```

```
) raises (
        IllegalLinkName,
        UnknownLinkName,
        DefaultFollowTooPermissive,
        LimitingFollowTooPermissive
);
```

8.5.6.5.2 Function

The modify_link operation is used to change the existing link follow behaviours of an identified link. The Lookup interface reference of the target trader and the name of the link can not be changed.

The “name” parameter identifies the link whose follow behaviours are to be changed. A poorly formed “name” raises the IllegalLinkName exception. An UnknownLinkName exception is raised if the link name is not known to the trader.

The “def_pass_on_follow_rule” parameter specifies the new default link behaviour for this link. If the “def_pass_on_follow_rule” exceeds the “limiting_follow_rule” specified in the next parameter, then a DefaultFollowTooPermissive exception is raised.

The “limiting_follow_rule” parameter specifies the new limit for the follow behaviour of this link. The exception LimitingFollowTooPermissive is raised if the value exceeds the current “max_link_follow_policy” of the trader.

8.5.7 Proxy

```
interface Proxy : TraderComponents, SupportAttributes {

    typedef Istring ConstraintRecipe;

    struct ProxyInfo {
        ServiceTypeName type;
        Lookup target;
        PropertySeq properties;
        boolean if_match_all;
        ConstraintRecipe recipe;
        PolicySeq policies_to_pass_on;
    };

    exception IllegalRecipe {
        ConstraintRecipe recipe;
    };

    exception NotProxyOfferId {
        OfferId id;
    };

    OfferId export_proxy (
        in Lookup target,
        in ServiceTypeName type,
        in PropertySeq properties,
        in boolean if_match_all,
        in ConstraintRecipe recipe,
        in PolicySeq policies_to_pass_on
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        InvalidLookupRef, // e.g. nil
        IllegalPropertyName,
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        IllegalRecipe,
        DuplicatePropertyName,
        DuplicatePolicyName
    );

    void withdraw_proxy (
        in OfferId id
    ) raises (
        IllegalOfferId,
        UnknownOfferId,
        NotProxyOfferId
    );
```

```
        ProxyInfo describe_proxy (
            in OfferId id
        ) raises (
            IllegalOfferId,
            UnknownOfferId,
            NotProxyOfferId
        );
    };
```

8.5.7.1 Export Proxy operation

8.5.7.1.1 Signature

```
        OfferId export_proxy (
            in Lookup target,
            in ServiceTypeName type,
            in PropertySeq properties,
            in boolean if_match_all,
            in ConstraintRecipe recipe,
            in PolicySeq policies_to_pass_on
        ) raises (
            IllegalServiceType,
            UnknownServiceType,
            InvalidLookupRef, // e.g. nil
            IllegalPropertyName,
            PropertyTypeMismatch,
            ReadonlyDynamicProperty,
            MissingMandatoryProperty,
            IllegalRecipe,
            DuplicatePropertyName,
            DuplicatePolicyName
        );
```

8.5.7.1.2 Function

The Proxy interface enables the export and subsequent manipulation of proxy offers. Proxy offers enable run-time determination of the interface at which a service is provided. The export_proxy operation adds a proxy offer to the trader's set of service offers.

Like normal service offers, proxy offers have a service type "type" and named property values "properties". However, a proxy offer does not include an object reference at which the offered service is provided. Instead, this object reference is obtained when it is needed for a query operation; it is obtained by invoking another query operation upon the "target" Lookup interface held in the proxy offer.

The "if_match_all" parameter, if TRUE, indicates that the trader should consider this proxy offer as a match to an importers query based upon type conformance alone, i.e. it does not match the importer's constraint expression against the properties associated with the proxy offer. This is most often useful when the constraint expression supplied by the importer is simply passed along in the secondary query operation.

The "recipe" parameter tells the trader how to construct the constraint expression for the secondary query operation to "target". The recipe language is described in Annex C; it permits the secondary constraint expression to be made up of literals, values of properties of the proxy offer, and the primary constraint expression.

The "policies_to_pass_on" parameter provides a static set of <name, value> pairs for relaying on to the "target" trader. The table below describes how the secondary policy parameter is generated from the primary policy parameter and the "policies_to_pass_on".

If a query operation matches the proxy offer (using the normal service type matching and property matching and preference algorithms), this primary query operation invokes a secondary query operation on the Lookup interface nominated in the proxy offer. Although the proxy offer nominates a Lookup interface, this interface is only required to conform syntactically to the Lookup interface; it need not conform to the Lookup interface behaviour specified above.

The secondary query operation is invoked as follows:

in ServiceTypeName type	The type is copied from primary query.
in Constraint constr	The recipe in the proxy offer is "evaluated" to provide the constr parameter.
in Preference pref	The preference is copied from the primary query.

n PolicySeq policies	The “policies” (names and values) contained in the policies_to_pass_on field of the proxy offer are appended to the policies of the primary query.
in SpecifiedProps desired_props	The desired_props are copied from the primary query.
in unsigned long how_many	The how_many parameter is set by the trader to reflect the trader implementation’s preference for receiving the resultant offer as a list or through an iterator.
out OfferSeq offers	At most how_many offers are returned from the secondary query operation via offers.
out OfferIterator offer_itr	If the secondary query needs to return more than how_many offers, then the remaining offers can be accessed via the iterator offer_itr. If there are only *how_many* or fewer offers, then offer_itr is nil.
out PolicyNameSeq limits_applied	The names of any policy limits that were applied by the proxy trader.

The IllegalServiceType exception is raised if the service type name (type) is not well-formed. The UnknownServiceType exception is raised if the service type name (type) is not known to the trader. The InvalidLookupRef exception is raised if target is not a valid Lookup interface reference (e.g. if target is a nil object reference). The IllegalPropertyName exception is raised if a property name in “properties” is not well-formed. The PropertyTypeMismatch exception is raised if a property value is not of an appropriate type as determined by the service type. The ReadonlyDynamicProperty exception is raised if a dynamic property value was supplied for a property that was flagged as readonly. The MissingMandatoryProperty exception is raised if “properties” does not contain one of the mandatory properties defined by the service type. The IllegalRecipe exception is raised if the recipe is not well-formed. The DuplicatePropertyName exception is raised if two or more properties, with the same property name, are included in the “properties” parameter. The DuplicatePolicyName exception is raised if two or more policies, with the same policy name, are included in the “policies_to_pass_on” parameter.

Proxy offers cannot be modified; they must be withdrawn and re-exported.

8.5.7.2 Withdraw Proxy operation

8.5.7.2.1 Signature

```
void withdraw_proxy (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        NotProxyOfferId
);
```

8.5.7.2.2 Function

The withdraw_proxy operation removes the proxy offer identified by “id” from the trader.

The IllegalOfferId exception is raised if “id” is not well-formed. The UnknownOfferId exception is raised if “id” does not identify any offer held by the trader. The NotProxyOfferId exception is raised if “id” identifies a normal service offer rather than a proxy offer.

8.5.7.3 Describe Proxy operation

8.5.7.3.1 Signature

```
ProxyInfo describe_proxy (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        NotProxyOfferId
);
```

8.5.7.3.2 Function

The describe_proxy operation returns the information contained in the proxy offer identified by “id” in the trader.

The IllegalOfferId exception is raised if “id” is not well-formed. The UnknownOfferId exception is raised if “id” does not identify any offer held by the trader. The NotProxyOfferId exception is raised if “id” identifies a normal service offer rather than a proxy offer.

8.6 Dynamic Property Evaluation interface

8.6.1 Signature

```
module CosTradingDynamic {

        exception DPEvalFailure {
            CosTrading::PropertyName name;
            TypeCode returned_type;
            any extra_info;
        };

        interface DynamicPropEval {

            any evalDP (
                in CosTrading::PropertyName name,
                in TypeCode returned_type,
                in any extra_info)
            raises (DPEvalFailure);
        };

        struct DynamicProp {
            DynamicPropEval eval_if;
            TypeCode returned_type;
            any extra_info;
        };
};
```

8.6.2 Functions of Dynamic Property Eval interface operations

The DynamicPropEval interface is provided by an exporter who wishes to provide a dynamic property value in a service offer held by the trader.

When exporting a service offer (or proxy offer), the property with the dynamic value has an “any” value which contains a DynamicProp structure rather than the normal property value. A trader which supports dynamic properties accepts this DynamicProp value as containing the information which enables a correctly-typed property value to be obtained during the evaluation of a query. The export (or export_proxy) operation raises the PropertyTypeMismatch if the returned_type is not appropriate for the property name as defined by the service type.

Readonly properties may not have dynamic values. The export and modify operations on the Register interface and the export_proxy operation on the Proxy interface raise the ReadonlyDynamicProperty exception if dynamic values are assigned to readonly properties.

When a query requires a dynamic property value, the evalDP operation is invoked on the eval_if interface in the DynamicProp structure. The property_name parameter is the name of the property whose value is being obtained. The returned_type and extra_info parameters are copied from the DynamicProp structure. The evalDP operation returns any value which should contain a value for that property. The value should be of a type indicated by returned_type.

The DPEvalFailure exception is raised if the value for the property cannot be determined. If the value is required for the evaluation of a constraint or preference, then that evaluation is deemed to have failed on that service offer (or proxy offer).

Other than the preceding rules, additional behaviour of the evalDP operation is not specified by this Specification. In particular, the purpose of the extra_info data in determining the dynamic property value is implementation-specific.

If the trader does not support dynamic properties (indicated by the trader attribute supports_dynamic_properties), the export and export_proxy operations should not be parameterised by dynamic properties. The behaviour of such traders in such circumstances is not specified by this Specification.

If the trader does not support dynamic properties or the importer has requested that dynamic properties are not used (via the policies parameter of the query operation), then dynamic property evaluation is not performed. If the value of a dynamic property is required by the evaluation of a constraint or preference, then that evaluation is deemed to have failed on that service offer (or proxy offer).

The describe operation of the Register interface and the describe_proxy operation of the Proxy interface do not perform dynamic property evaluation, but return the DynamicProp structure as the value of the property. As these interfaces are used to create dynamic properties via the export and export_proxy operations, the other operations on these interfaces must ensure that the dynamic nature of the properties remains visible to the exporters.

The modify operation on the Register interface of a trader which supports dynamic properties must accept the establishment and modification of dynamic properties, consistent with the export operation. There is no restriction on a property value changing from a static value stored by the trader into a dynamic value, and vice versa; however readonly static properties may not be modified to be dynamic.

8.7 Trader object template

A basic computational object template for the trader contains:

- computational interface templates for the interfaces that it can instantiate;
- a behaviour specification;
- an environment contract specification.

8.7.1 Interface templates

This Specification provides the following interface templates:

- trader components interface template;
- support attributes interface template;
- import attributes interface template;
- link attributes interface template;
- lookup interface template;
- register interface template;
- admin interface template;
- link interface template;
- proxy interface template;
- offer iterator interface template;
- offer id iterator interface template;
- dynamic property evaluator interface template.

8.7.2 Behaviour specification

Instantiation of an interface template in the server role requires an object to meet the behaviour specifications associated with the operations contained in the interface.

Instantiation of an interface template in the client role requires an object to be able to receive all possible terminations for the operations it invokes on that interface.

When a trader object instantiates a lookup interface in the client role in order to support services invoked upon a lookup interface in its server role, it shall pass the responses received from that client role interface to the server role interface, in accordance with the behaviour specification of the lookup interface template.

When a trader object instantiates a register interface in the client role in order to support services invoked upon a lookup interface in its server role, it shall pass the responses received from that client role interface to the server role interface, in accordance with the behaviour specification of the lookup interface template.

When a trader object instantiates a register interface in the client role in order to support services invoked upon a register interface in its server role, it shall pass the responses received from that client role interface to the server role interface, in accordance with the behaviour specification of the register interface template.

In addition there are other actions that the trader can perform.

8.7.2.1 Instantiating interface templates

The following templates can be instantiated:

- lookup interface template (server role);
- lookup interface template (client role);
- register interface template (server role);
- admin interface template (server role);
- link interface template (server role);
- proxy interface template (server role);
- offer iterator interface template (server role);
- offer id iterator interface template (server role).

NOTE – There may be many instances of each of lookup (client role), offer iterator, offer id iterator interfaces.

Implementations may instantiate other client role interfaces, for example type repository interface or storage interface. However, these are separate ODP functions and their standardization is beyond the scope of this Specification.

8.7.2.2 Actions regarding state

The trader can access and modify its state. The following components of the state are identified:

- the set of service offers;
- the set of proxy offers;
- the set of links;
- the set of trader attributes.

The register interface updates the service offers.

The proxy interface updates the proxy offers.

The link interface updates the links.

The admin interface updates the trader attributes.

8.7.2.3 Combined lookup-register object behaviour

All of the service offers that are currently in the trader due to use of operations in register interface are available to be returned through the query operation of the lookup interface.

8.7.2.4 Combined lookup-proxy object behaviour

All of the proxy offers that are currently in the trader due to use of operations in proxy interface are available to be returned through the query operation of the lookup interface.

8.7.2.5 Combined admin-lookup object behaviour

The set of service offers that may be returned is affected by the value of the support_dynamic_properties attribute, set through the admin interface operations. If the support_dynamic_properties attribute is set to value of FALSE, then subsequent query operations shall not return offers which include dynamic properties.

The set of proxy offers that may be returned is affected by the value of the support_proxy_offer attribute set through the admin interface operations. If the support_proxy_offer is set to value of FALSE, then subsequent query operations shall not return offers through a proxy offer, even if the query operation import policy parameter has use_proxy_offers value set to TRUE.

The set of service offers that may be returned is affected by the vlue of support_modifiable_properties attribute, set thwough the admin interface operations. If the support_modifiable_properties attribute is set to FALSE, the subsequent query operations shall not return offers which include modifiable properties.

8.7.2.6 Combined admin-register object behaviour

The availablity of the modify operation in the register interface, is affected by the value of the supports_modifiable_properties attribute of the trader.

The ability to export service offers with dynamic properties, is affected by the value of the supports_dynamic_properties attribute of the trader. If the value of the supports_dynamic_properties attribute is set to FALSE, then export requests which include dynamic properties may be rejected.

8.7.2.7 Combined admin-proxy object behaviour

The functions associated with the proxy interface are not affected by the value of the supports_proxy_offer attribute. If the value of the supports_proxy_offer attribute is set to FALSE, any proxy offers within the trader shall not be made available to query operations on the lookup interface.

The ability to export proxy offers with dynamic properties, is affected by the value of the supports_dynamic_properties attribute of the trader. If the value of the supports_dynamic_properties attribute is set to FALSE, then export_proxy requests which include dynamic properties may be rejected.

8.7.2.8 Combined link-lookup object behaviour

All of the service offers that are available through the lookup interfaces of linked traders shall be recursively accessible by a query operation upon an initial lookup interface.

8.7.2.9 Combined link-register object behaviour

The resolve operation on the register is only available if the trader is linked with other traders.

8.7.2.10 Arbitration action

An arbitration action template is a template for actions which combine a criteria argument (provided at an interface) with trader criteria and property values (available from the trader's own state). The action produces a resultant criteria which corresponds to the policy (in enterprise terms) for performing a given operation.

The arbitration action represents some computational algorithm within the trader object. It corresponds to the enterprise specification's arbitration policy.

8.7.2.11 Constraints on the occurence of actions

The behaviour of the export interface modifies the service offer space and offer identities. The behaviour of the link interface modifies the link space. The behaviour of the admin interface modifies the trader attributes. Therefore no constraints on interleaving of actions from operations on one interface with actions from operations on the other interface are necessary to ensure data consistency. The behaviour of the proxy interface modifies the proxy offer space and offer identifiers.

8.7.3 Environment contract

There may be an environment constraint that the first trading service interface that is instantiated, be at a particular location. This would allow other objects to be instantiated with knowledge of a trader.

No further environment constraints are identified.

9 Conformance statements and reference points

The following operational interfaces are programmatic reference points for testing conformance:

- the Lookup interface (as server) provided by the trader implementation under test;
- the Register interface (as server) provided by the trader implementation under test;
- the Link interface (as server) provided by the trader implementation under test;
- the Admin interface (as server) provided by the trader implementation under test;
- the Proxy interface (as server) provided by the trader implementation under test;
- a Lookup interface (as client) of a linked trader, used by the trader implementation under test;
- a Register interface (as client) of a linked trader, used by the trader implementation under test;
- a DynamicPropEval interface (as client) of an object, used by the trader implementation under test during the evaluation of a dynamic property.

The behaviour defined for each of the normative operations in the computational specification shall be exhibited at the conformance points associated with that behaviour.

In order to claim conformance, implementors shall state what engineering interface corresponds to each of the computational interfaces identified as conformance points.

In addition, the implementor shall state which communications mechanism is used and which transparencies are provided over which interfaces.

The following taxonomy is defined for specific implementation conformance classes of trading function object implementations:

- Query trader: Can serve Lookup interface.
- Simple trader: Can serve Lookup and Register interfaces.
- Stand-alone trader: Can serve Lookup, Register and Admin interfaces.
- Linked trader: Can serve Lookup, Register, Admin and Link interfaces; and can be client for Lookup and Register interfaces.
- Proxy trader: Can serve Lookup, Register, Admin and Proxy interfaces; and can be client for Lookup interface.
- Full-servicetrader: Can serve Lookup, Register, Admin, Link and Proxy interfaces; and can be client for Lookup and Register interfaces.

Any of these specific trading object conformance classes may also be a client for the DynamicPropEval interface if it supports dynamic properties.

The above taxonomy results in five specific trading function object types relevant for conformance claims.

9.1 Conformance requirement for trading function interfaces as server

Since the interfaces to a trading object are separable, and support for these interfaces is selectable, subject to the conformance classes defined above, this subclause specifies the conformance requirements on a per-interface basis.

9.1.1 Lookup interface

An implementation claiming conformance to the Lookup interface as server shall implement the complete behaviour associated with all the operations and readonly attributes defined within the scope of the Lookup interface type.

An implementation claiming conformance to the Lookup interface as server shall also support the OfferIterator interface type as server.

9.1.2 Register interface

An implementation claiming conformance to the Register interface as server shall implement the complete behaviour associated with all the operations and readonly attributes defined within the scope of the Register interface type, with the following permitted exceptions:

- An implementation which only allows the value of FALSE for the supports_modifiable_properties attribute is conformant, in which case it may reject a service offer which includes modifiable properties passed in an export operation, and may always respond to modify operation requests with an exception.
- An implementation which only allows the value of FALSE for the supports_dynamic_properties attribute is conformant, in which case it may reject a service offer which includes dynamic properties passed in an export operation.
- An implementation claiming conformance to the Register interface as server, with the value of the supports_dynamic_properties set to TRUE, shall also be able to assume the client role for the DynamicPropEval interface type.
- An implementation claiming conformance to the Register interface as server, with the value of the readonly attribute supports_proxy_offers set to TRUE, shall also support the Proxy interface.

9.1.3 Link interface

An implementation claiming conformance to the Link interface as server shall implement the complete behaviour associated with all the operations and readonly attributes defined within the scope of the Link interface type.

9.1.4 Admin interface

An implementation claiming conformance to the Admin interface as server shall implement the complete behaviour associated with all the operations and readonly attributes defined within the scope of the Admin interface type.

An implementation claiming conformance to the Admin interface as server shall also support the OfferIdIterator interface type as server.

9.1.5 Proxy interface

An implementation claiming conformance to the Proxy interface as server shall implement the complete behaviour associated with all the operations defined within the scope of the Proxy interface type.

An implementation claiming conformance to the Proxy interface as server shall also support the OfferIdIterator interface type as server.

9.1.6 OfferIterator interface

An implementation claiming conformance to the OfferIterator interface as server shall implement the complete behaviour associated with all the operations defined within the scope of the OfferIterator interface type.

9.1.7 OfferIdIterator interface

An implementation claiming conformance to the OfferIdIterator interface as server shall implement the complete behaviour associated with all the operations defined within the scope of the OfferIdIterator interface type.

9.1.8 DynamicPropEval interface

An implementation claiming conformance to the DynamicPropEval interface as server shall implement the required behaviour associated with all the operations defined within the scope of the DynamicPropEval interface type.

NOTE – The behaviour of this interface is limited to support of the interface signature.

9.1.9 Service subtyping rule conformance

ODP-IDL is used in this Specification to express computational operation interface signatures. Use of this notation does not imply use of specific supporting mechanisms and protocols.

For conformance, the architectural neutral service subtyping rules in 8.2.3.2 are the widest interpretation of subtyping for a conformant trader. An implementation of these architectural neutral service subtyping rules in a specific mechanism and protocol cannot be more permissive than these service subtyping rules. A specific mechanism and protocol may not support a definition of subtyping which is not permitted by the ODP trading function.

The most restrictive (lower bound) rules for subtyping of a conformant trader implementation built on a particular supporting mechanism and protocol, allowed for conformance, are:

- A service type b, is a subtype of service type a, if and only if:
  - the interface type of ST b may add operations to the interface type of ST a; and
  - all the named properties of ST a are in ST b; and
  - all of the named properties of ST a have a type which is identical to the type of the identically named property in ST b; and
  - all of the named properties of ST a have mode which is identical to the mode of the identically named property in ST b.

9.2 Conformance requirements for query trader conformance class

A trading system implementation claiming conformance to the query trader conformance class, shall conform to the conformance requirements of the Lookup interface type as server.

Implementations conforming to this conformance class would not claim conformance to the register interface, and would use non-standard means for introducing service offers into the trader.

9.3 Conformance requirements for simple trader conformance class

A trading system implementation claiming conformance to the simple trader conformance class, shall conform to the conformance requirements of the Lookup and Register interface types as server.

A simple trader shall conform to the combined lookup-register object behaviour specified in 8.7.2

9.4 Conformance requirements for stand-alone trader conformance class

A trading system implementaion claiming conformance to the stand-alone Trader conformance class, shall conform to the conformance requirements of the Lookup, Register and Admin interface types as server.

A stand-alone trader shall conform to the combined lookup-register object behaviour specified in 8.7.2.

A stand-alone trader shall conform to the combined admin-register object behaviour specified in 8.7.2.

A stand-alone trader shall conform to the combined admin-lookup object behaviour specified in 8.7.2.

9.5 Conformance requirements for linked trader conformance class

A trading system implementaion claiming conformance to the linked trader conformance class, shall conform to the conformance requirements of the Lookup, Register, Admin and Link interface types as server; and shall be able to assume the client role for invoking operations on the Lookup interface type.

A linked trader shall conform to the combined lookup-register object behaviour specified in 8.7.2.

A linked trader shall conform to the combined admin-register object behaviour specified in 8.7.2.

A linked trader shall conform to the combined admin-lookup object behaviour specified in 8.7.2.

A linked trader shall conform to the combined lookup-link object behaviour specified in 8.7.2.

9.6 Conformance requirements for proxy trader conformance class

A trading system implementaion claiming conformance to the proxy trader conformance class, shall conform to the conformance requirements of the Lookup, Register, Admin, and Proxy interface types as server; and shall be able to assume the client role for invoking operations on the Lookup interface type.

A proxy trader shall conform to the combined lookup-register object behaviour specified in 8.7.2.

A proxy trader shall conform to the combined admin-register object behaviour specified in 8.7.2.

A proxy trader shall conform to the combined admin-lookup object behaviour specified in 8.7.2.

A proxy trader shall conform to the combined lookup-proxy object behaviour specified in 8.7.2.

A proxy trader shall conform to the combined admin-proxy object behaviour specified in 8.7.2.

9.7 Conformance requirements for full-service trader conformance class

A trading system implementaion claiming conformance to the full-service trader conformance class, shall conform to the conformance requirements of the Lookup, Register, Admin, Link and Proxy interface types as server; and shall be able to assume the client role for invoking operations on the Lookup interface type.

A full-service trader shall conform to the combined lookup-register object behaviour specified in 8.7.2.

A full-service trader shall conform to the combined admin-register object behaviour specified in 8.7.2.

A full-service trader shall conform to the combined admin-lookup object behaviour specified in 8.7.2.

A full-service trader shall conform to the combined lookup-link object behaviour specified in 8.7.2.

A full-service trader shall conform to the combined lookup-proxy object behaviour specified in 8.7.2.

A full-service trader shall conform to the combined admin-proxy object behaviour specified in 8.7.2.

9.8 Conformance tests

For each conformance test, the implementation shall identify the programmatic reference points for which conformance is claimed.

For each conformance test, the implementor shall state:

- what policy is in force for the duration of the test;
- what object state is assumed, with respect to the information specification, as a static schema, e.g. service offers held.

An implementation claiming conformance is required to provide the set of policies or circumstances under which an exported offer can be subsequently retrieved by a query operation.

Implementations claiming conformance to either the linked trader or full-service trader conformance classes shall be able to demonstrate the ability to propagate a query operation to a remote trader's lookup interface.

Implementations claiming conformance to either the proxy trader or full-service trader conformance classes shall be able to demonstrate the ability to forward a query operation to a remote object via a proxy offer.

Annex A

ODP-IDL based specification of the Trading Function

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

A.1 Introduction

This annex provides the ODP-IDL (see ITU-T Rec. X.920 | ISO/IEC 14750) specification of the interface signature for the trading function's computational specification. It specifies the signature for each computational operation in ODP-IDL, according to the functional description (signature and semantics) provided in clause 8.

If there are any discrepancies between the ODP-IDL specifications in this annex and those in Clause 8, the specifications in this annex take precedence.

ODP-IDL is used in this Specification to express computational operation interface signatures. Use of this notation does not imply use of specific supporting mechanisms and protocols.

NOTE – The templates in this annex are technically aligned with the templates of the OMG Trading Object Service.

A.2 ODP Trading Function module

```
module CosTrading {

        // forward references to our interfaces

        interface Lookup;
        interface Register;
        interface Link;
        interface Proxy;
        interface Admin;
        interface OfferIterator;
        interface OfferIdIterator;

        // type definitions used in more than one interface

        typedef string Istring;
        typedef Object TypeRepository;

        typedef Istring PropertyName;
        typedef sequence<PropertyName> PropertyNameSeq;
        typedef any PropertyValue;
        struct Property {
                PropertyName name;
                PropertyValue value;
        };
        typedef sequence<Property> PropertySeq;

        struct Offer {
                Object reference;
                PropertySeq properties;
        };
        typedef sequence<Offer> OfferSeq;
        typedef string OfferId;
        typedef sequence<OfferId> OfferIdSeq;

        typedef Istring ServiceTypeName;  // similar structure to IR::Identifier

        typedef Istring Constraint;

        enum FollowOption {
                local_only,
                if_no_local,
                always
};
```

```
typedef Istring LinkName;
typedef sequence<LinkName> LinkNameSeq;
typedef LinkNameSeq TraderName;

typedef string PolicyName;  // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
        PolicyName name;
        PolicyValue value;
};
typedef sequence<Policy> PolicySeq;

// exceptions used in more than one interface

exception UnknownMaxLeft {};

exception NotImplemented {};

exception IllegalServiceType {
        ServiceTypeName type;
};

exception UnknownServiceType {
        ServiceTypeName type;
};

exception IllegalPropertyName {
        PropertyName name;
};

exception DuplicatePropertyName {
        PropertyName name;
};

exception PropertyTypeMismatch {
        ServiceTypeName type;
        Property prop;
};

exception MissingMandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception ReadonlyDynamicProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception IllegalConstraint {
        Constraint constr;
};

exception InvalidLookupRef {
        Lookup target;
};

exception IllegalOfferId {
        OfferId id;
};

exception UnknownOfferId {
        OfferId id;
};

exception DuplicatePolicyName {
        PolicyName name;
};

// the interfaces
```

```
interface TraderComponents {

        readonly attribute Lookup lookup_if;
        readonly attribute Register register_if;
        readonly attribute Link link_if;
        readonly attribute Proxy proxy_if;
        readonly attribute Admin admin_if;
};

interface SupportAttributes {

        readonly attribute boolean supports_modifiable_properties;
        readonly attribute boolean supports_dynamic_properties;
        readonly attribute boolean supports_proxy_offers;
        readonly attribute TypeRepository type_repos;
};

interface ImportAttributes {

        readonly attribute unsigned long def_search_card;
        readonly attribute unsigned long max_search_card;
        readonly attribute unsigned long def_match_card;
        readonly attribute unsigned long max_match_card;
        readonly attribute unsigned long def_return_card;
        readonly attribute unsigned long max_return_card;
        readonly attribute unsigned long max_list;
        readonly attribute unsigned long def_hop_count;
        readonly attribute unsigned long max_hop_count;
        readonly attribute FollowOption def_follow_policy;
        readonly attribute FollowOption max_follow_policy;
};

interface LinkAttributes {

        readonly attribute FollowOption max_link_follow_policy;
};

interface Lookup : TraderComponents, SupportAttributes, ImportAttributes {

        typedef Istring Preference;

        enum HowManyProps { none, some, all };

        union SpecifiedProps switch ( HowManyProps ) {
                case some: PropertyNameSeq prop_names;
        };

        exception IllegalPreference {
                Preference pref;
        };

        exception IllegalPolicyName {
                PolicyName name;
        };

        exception PolicyTypeMismatch {
                Policy the_policy;
        };

        exception InvalidPolicyValue {
                Policy the_policy;
        };

        void query (
                in ServiceTypeName type,
                in Constraint constr,
                in Preference pref,
                in PolicySeq policies,
                in SpecifiedProps desired_props,
                in unsigned long how_many,
                out OfferSeq offers,
                out OfferIterator offer_itr,
                out PolicyNameSeq limits_applied
```