Trading Object Service Specification 16

This chapter provides complete documentation for the Trading Object Service specification.

Contents

This chapter contains the following sections.

16.1 Overview

The OMG trading object service facilitates the offering and the discovery of instances of services of particular types. A trader is an object that supports the trading object service 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 and the location of an interface where that service is available. To import, an object asks the trader for a service having certain characteristics. The trader checks against the service descriptions it holds and responds to the importer with the location of the selected service’s interface. The importer is then able to interact with the service. These interactions are shown in Figure 16-1.

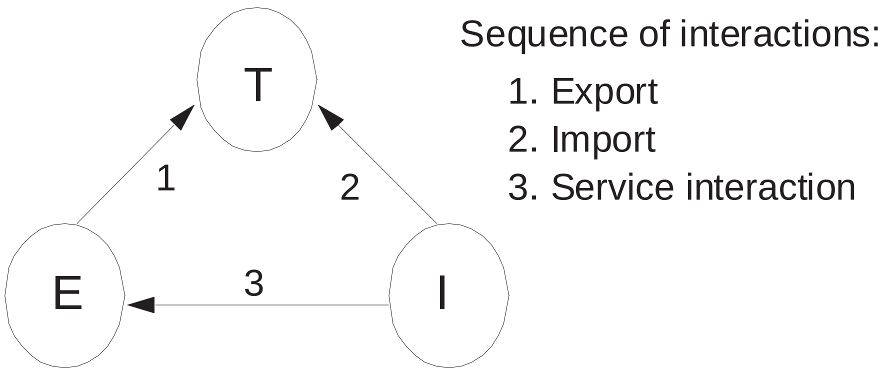


Figure 16-1 Interactions between a trader and its clients

Due to the number of service offers that will be offered worldwide, and the differing requirements that users of a trading service will have, it is inevitable that a trading service will be split up and 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 interworking (or federation) of traders.

The trading object service in an OMG environment allows interworking between traders and objects to:

- export (advertise) services
- import information about one or more exported services, according to some criteria

16.1.1 Diversity and Scalability

The concept of trading to discover new services applies to a wide range of scenarios. A trader may contain numerous offers of service and its implementation may be based upon a database. Or, a trader may contain only a few offers and 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 large systems and downwards for small, fast systems.

To discover any arbitrary offer of service, a trader needs all offers to be visible to it. One partition cannot hold every offer, many are 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 utilized indirectly via other traders.

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

16.1.2 Linking Traders

The requirements to contextualise the offer space and to distribute the trading object service are both met by linking traders together. When a trader links to other traders, it makes the offer spaces 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 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 (e.g., administrative, technological, etc.); therefore, trading is a federated system (i.e., one that spans many domains).

16.1.3 Policy

To meet the diverse requirements likely to be placed upon the trading function, some degree of freedom is necessary when specifying the behavior 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 behavior of any OMG trading object service implementation.

This specification identifies a number of policies and gives them semantics. Each policy partly determines the behavior of a trader.

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

16.1.4 Additional ObjectID

A trading object service may be used by an object to bootstrap itself into operation; as such, this specification mandates an additional ObjectId for use in the resolve_initial_references() operation defined in the ORB Initialization Specification, OMG Document 94-10-24.

The following ObjectId is reserved for finding an initial trading object service:

```
TradingService
```

As described in 94-10-24, a client object wishing to obtain an initial trading object service object reference will invoke the resolve_initial_references() operation, which has the following OMG IDL signature:

```
typedef string ObjectId;
exception InvalidName {};


Object resolve_initial_references (in ObjectId identifier) raises
(InvalidName);
```

The object reference returned as the result of a successful invocation of this operation when “TradingService” is specified as the ObjectId parameter must be narrowed to an object reference of the appropriate type; for the trading object service this type is CosTrading::Lookup.

No other extensions are proposed to OMG IDL, CORBA, and/or the OMG object model.

16.2 Concepts and Data Types

16.2.1 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.

16.2.2 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.

16.2.3 Service Types

A service type, which represents the information needed to describe a service, is associated with each traded 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 behavioral, 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;
```

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

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

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 type 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.

Service Type Model

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. Its structure is similar to that of interface repository identifiers (::First::Second::Third ...). 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 16-2.

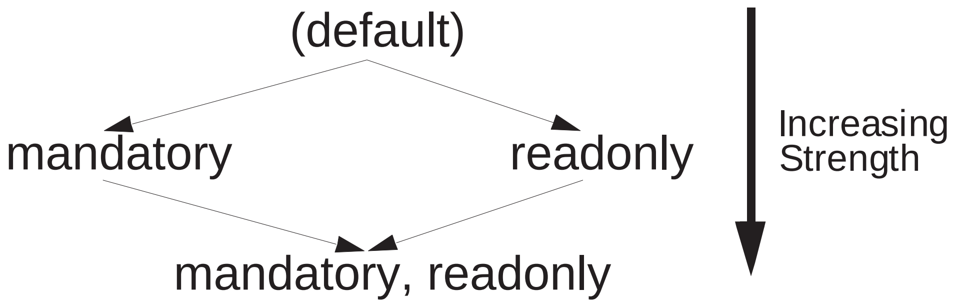


Figure 16-2 Property Strength

Summarizing, 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 subsequently may be modified. The “mandatory” modifier indicates that a value must be provided, but that subsequently it may be modified. The “readonly” modifier indicates that the property is optional, but that once given a value, subsequently it may not be modified. Specifying both modifiers indicates that a value must be provided and that subsequently it may not be 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 derived from, 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 the same property value type in β as their corresponding definitions had in α

16.2.4 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;
};
```

16.2.5 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 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;
    };
```

Modifiable Properties

The value of a property in a service offer can 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 readonly 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.

Dynamic Properties

A service offer can 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;
        CORBA::TypeCode returned_type;
        any extra_info;
    };

    interface DynamicPropEval {

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

```
struct DynamicProp {
     DynamicPropEval eval_if;
     CORBA::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 standard for portability but its behavior 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.

16.2.6 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;
```

16.2.7 Offer Selection

The total service offer space for an offer selection may be 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 applied to S1 to produce the set S2 that satisfies the service type and constraint. Then this is ordered using preferences before returning the offers to the importer.

Standard Constraint Language

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

This document defines the standard, mandatory language which is necessary for interworking between traders. Appendix 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	These manipulations are restricted to int, float, 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 “exist” operator.
Literals	In the constraint, literals are dynamically coerced as required for the properties they are working with. Literals can contain Istring.
Operators	The operators are comparison, boolean connective, set inclusion, substring, arithmetic operators, property existence.

Note – If a proprietary constraint language (outside the scope of this specification) is used, then the name and version of the constraint language 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.

Preferences

Preferences are applied logically to the set of offers matched by application of the service type, constraint expression, and various policies. Application of the preferences can determine the order used to return matched offers to the importer.

```
    typedef Istring Preference;
```

Consider the preference string as being composed of two portions.

- The first portion can be comprised of any of the following case-sensitive keywords:

  ```
  max min with random first
  ```

- The interpretation for the second portion is dependent on the first portion; it may be empty. Table 16-1 describes the preferences.

Table 16-1 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 returned 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 in the order as 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 into a group of offers for which the preference expression

- could be evaluated (ordered according to min, max, with), and
- 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.

Note – If a proprietary preference language (outside the scope of this specification) is used, 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.

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 behavior, which may be used and is passed on when an importer does not specify a link_follow_rule policy.
- The link's limiting follow behavior, which overrides an importer's link_follow_rule if the importer's request exceeds the limit set by the link.

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

Policies

Policies provide information to affect trader behavior 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:

1. Policies that scope the extent of a search.
2. Policies that determine the functionality applied to an operation.

Different policies are associated with different roles in the performance of the trading function. These roles are:

```
T   = Trader
L   = Link
I   = Import
```

Standardized Scoping Policies:

The following table lists the standardized scoping policies.

Table 16-2 Scoping Policies

Name	Where	IDL Type	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	Maximum 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.

Name	Where	IDL Type	Description
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 behavior 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 behavior for a particular link.
def_follow_policy	T	FollowOption	Default link follow behavior 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.
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.
link_follow_rule	I	FollowOption	Nominated link follow behavior; 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 behavior is local_only.
request_id	I	OctetSeq	An identifier for a query operation initiated by a source trader acting as an importer on a link; a trader is not obliged to generate an id, but is obliged to pass one received 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 serviced type that conforms to the importer's service type are considered.

The 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 following diagram, where |N1| >= |N2| >= |N3| = |N4| >= |N5

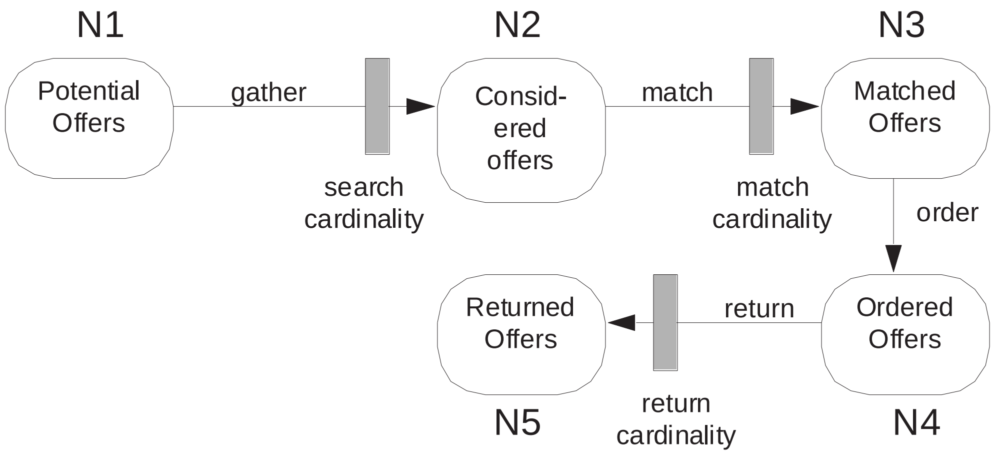


Figure 16-3 Pipeline View of Trader Query Steps and Cardinality Constraint Application

Standardized Capability Supported Policies

There are three optional 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 cannot override it 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 table lists the standardized policies related to supported functionality.

Table 16-3 Capability Supported Policies

Name	Where	IDL Type	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 search.
supports_proxy_offers	T	boolean	Whether the trader supports proxy offers.
use_proxy_offers	I	boolean	Whether to consider proxy offers in the search.

Trader Policies

Policies can be set for a trader as a whole. Trader policies are defined as attributes of the trader object. They are specified initially 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.

Link Follow Behavior

Each link in a trader has its own follow behavior policies. A trader has a limiting follow policy, max_follow_policy, that overrides all the links of that trader for any given query. Follow behavior 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.

Recall that the OMG IDL for FollowOption is:

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

where “local_only” indicates that the link is followed only by explicit navigation (“starting_trader” policy), “if_no_local” indicates that the link is followed only if there are no local offers that satisfy the query, and “always” has the obvious semantics. These values are ordered as follows:

```
local_only < if_no_local < always
```

The follow policy for a particular link is:

```
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)
```

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 the nested query is permitted by the above rule, then the following logic determines the value for the “link_follow_rule” policy to pass 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)
```

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.

Exporter Policies

There are no exporter policies specified in this standard.

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 compares 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.

16.2.8 Interworking Mechanisms

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 remembers the RequestId of all recent interworking query operations that it has been asked to perform. When an interworking query operation is received, the 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 must pass the RequestId onto the next linked trader if the search propagates further.

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.

The following example illustrates 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 behavior.

1. 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.
2. 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.
3. Assuming that no match is found in T3 and 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.
4. 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 completed successfully 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.

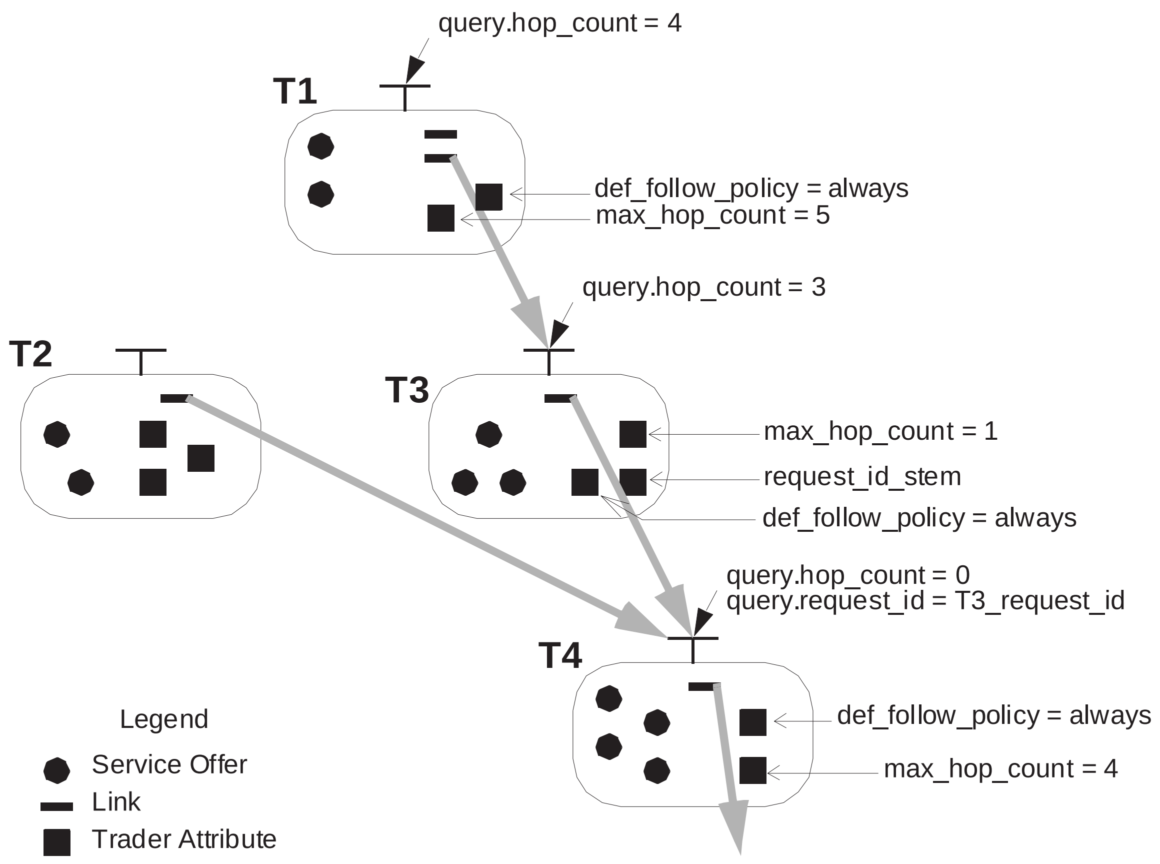


Figure 16-4 Flow of a query through a trader graph

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. Any offer returned by the proxy trader as a result of the nested query must 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. Traders that support such functionality 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.

16.2.9 Trader Attributes

Each trader has its own characteristics, policies for supported functionalities, and policies for scoping the extent of search. These characteristics and policies are defined as attributes to the trader. These attributes are described in Table 16-4.

Table 16-4 Trader Attributes

Name	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	Maximum 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	Maximum 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	Maximum 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
max_list	unsigned long	The upper bound on the size of any list returned by the trader, namely the returned offers parameter in query, and the next_n operations in OfferIterator and OfferIdIterator.
def_follow_policy	FollowOption	Default link follow behavior 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
type_repos	TypeRepository	Interface to trader's service type 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.

16.3 Exceptions

This specification defines the exceptions raised by operations. Exceptions are parameterized to indicate the source of the error. The OMG IDL segments below refer to some of the typedef's defined in Section 16.2 Concepts and Data Types.

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

16.3.1 For CosTrading module

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;
};
```

Additional Exceptions for Lookup Interface

```
exception IllegalPreference {
    Preference pref;
};

exception IllegalPolicyName {
    PolicyName name;
};

exception PolicyTypeMismatch {
    Policy the_policy;
};


exception InvalidPolicyValue {
    Policy the_policy;
};
exception IllegalPreference {
    Preference pref;
};
exception IllegalPolicyName {
    PolicyName name;
};
exception PolicyTypeMismatch {
    Policy policy;
};
```

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;
};
```

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;
};
```

Additional Exceptions for Proxy Offer Interface

```
exception IllegalRecipe {
    ConstraintRecipe recipe;
};

exception NotProxyOfferId {
    OfferId id;
};
```

16.3.2 For CosTradingDynamic module

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

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

16.3.3 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;
    };
```

16.4 Abstract Interfaces

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.

16.4.1 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;
};
```

A trader’s functionality can be configured by composing the defined interfaces in one of several prescribed combinations. The composition is not modeled 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 provide a way to get a specific object reference.

The implementation of the _get_<interface>_if() operation must return a nil object reference if the trading service in question does not support that particular interface.

16.4.2 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.

16.4.3 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.

16.4.4 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 behavior that the link will be allowed. The value for this constraint on link creation and modification can be obtained from this interface.

16.5 Functional Interfaces

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

16.5.1 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
        );
    };
```

Query Operation

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
    );
```

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 correct syntactically 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. Sub-typing of service types is discussed in “Service Types” on page 16-4. A service subtype can be described by the properties of its supertypes. This ensures that 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 behavior 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, then an IllegalConstraint exception is raised.

The “pref” parameter is also 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 parameterizing 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 standardized 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, then 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 one of the properties, all of the properties (but without having to name them), or 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, then 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 “exists prop_name” 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.

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;
};
```

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 “Link Follow Behavior” on page 16-16 for details.

The “hop_count” policy indicates to the trader the maximum number 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.

16.5.2 Offer Iterator

Signature

```
interface OfferIterator {
    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );
    boolean next_n (
            in unsigned long n,
            out OfferSeq offers
        );
        void destroy ();
    };
```

Function

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.

16.5.3 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
        );
    };
```

Export Operation

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
        );
```

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 unexportable, 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 correct syntactically 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 behavioral, 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.

Withdraw Operation

Signature

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

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.

Describe Operation

Signature

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

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.

Modify Operation

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
);
```

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:

- add new (non-mandatory) properties to describe an offer,
- change the values of some existing (not readonly) properties, or

- 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.

Note – 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 while preserving the OfferId. This might be important where the OfferId has been propagated around a community of objects.

Withdraw Using Constraint Operation

Signature

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

Function

The withdraw_using_constraint operation withdraws a set of offers from within a single trader. This set is identified in the same way that 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 correct syntactically 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.

Resolve Operation

Signature

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

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 object 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 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, 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).

16.5.4 Offer Id Iterator

Signature

```
interface OfferIdIterator {

    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );
```

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

        void destroy ();
    };
```

Function

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.

16.5.5 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
        );
    };
```

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 in 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, it does have the effect of making any proxy offers exported via the proxy interface for that trader unavailable to satisfy queries on that trader's lookup interface.

List Offers Operation

Signature

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

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 the number of offers held in the trader, then the "id_itr" is nil.

List Proxies Operation

Signature

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

Function

The list_proxies operation returns the set of offer identifiers for proxy offers held by a trader. 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.
- There are only “how_many” or fewer offer identifiers, the id_itr is nil.
- The trader does not support the Proxy interface, the NotImplemented exception is raised.

16.5.6 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
        );
    };
```

Add_Link Operation

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
        );
```

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 behavior for the link if no link behavior 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 behavior 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.

Remove Link Operation

Signature

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

Function

The remove_link operation removes all knowledge of the target trader. The target trader cannot be used subsequently 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 formed poorly and the UnknownLinkName exception is raised if the named link is not in the trader.

Describe Link Operation

Signature

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

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 behavior of the named link.

If the target service does not support the Register interface, then that field of the LinkInfo structure is nil. Given the description of the Register::resolve() operation in “Resolve Operation” on page 16-45, most implementations will opt for determining the Register interface when add_link is called and storing that information statically with the rest of the link state.

List Links Operation

Signature

```
    LinkNameSeq list_links ();
```

Function

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

Modify Link Operation

Signature

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

Function

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

The “name” parameter identifies the link whose follow behaviors 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 behavior 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 behavior of this link. The exception LimitingFollowTooPermissive is raised if the value exceeds the current “max_link_follow_policy” of the trader.

16.5.7 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
        );
    };
```

Export Proxy Operation

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
    );
```

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 Appendix 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. Table 16-5 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 behavior specified above.

The secondary query operation is detailed in Table 16-5.

Table 16-5 Primary/Secondary Policy Parameters

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.
in 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.

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

Withdraw Proxy Operation

Signature

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

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.

Describe Proxy Operation

Signature

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

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.

16.6 Service Type Repository

```
module CosTradingRepos {

    interface ServiceTypeRepository {

    // local types
        typedef sequence<CosTrading::ServiceTypeName>
                                        ServiceTypeNameSeq;
        enum PropertyMode {
            PROP_NORMAL, PROP_READONLY,
            PROP_MANDATORY, PROP_MANDATORY_READONLY
        };
        struct PropStruct {
            CosTrading::PropertyName name;
            CORBA::TypeCode value_type;
            PropertyMode mode;
        };
        typedef sequence<PropStruct> PropStructSeq;

        typedef CosTrading::Istring Identifier; // IR::Identifier
        struct IncarnationNumber {
            unsigned long high;
            unsigned long low;
        };
        struct TypeStruct {
            Identifier if_name;
            PropStructSeq props;
```

```
                ServiceTypeNameSeq super_types;
                boolean masked;
                IncarnationNumber incarnation;
            };


            enum ListOption { all, since };
            union SpecifiedServiceTypes switch ( ListOption ) {
                case since: IncarnationNumber incarnation;
            };

        // local exceptions
            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;
    };

// attributes
    readonly attribute IncarnationNumber incarnation;

// operation signatures
    IncarnationNumber add_type (
        in CosTrading::ServiceTypeName name,
        in Identifier if_name,
        in PropStructSeq props,
        in ServiceTypeNameSeq super_types
    ) raises (
        CosTrading::IllegalServiceType,
        ServiceTypeExists,
        InterfaceTypeMismatch,
        CosTrading::IllegalPropertyName,
        CosTrading::DuplicatePropertyName,
        ValueTypeRedefinition,
        CosTrading::UnknownServiceType,
        DuplicateServiceTypeName
    );

    void remove_type (
        in CosTrading::ServiceTypeName name
    ) raises (
        CosTrading::IllegalServiceType,
        CosTrading::UnknownServiceType,
        HasSubTypes
    );

    ServiceTypeNameSeq list_types (
        in SpecifiedServiceTypes which_types
    );

    TypeStruct describe_type (
        in CosTrading::ServiceTypeName name
    ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType
        );

        TypeStruct fully_describe_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType
        );

        void mask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            AlreadyMasked
        );

        void unmask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            NotMasked
        );

    };
}; /* end module CosTradingRepos */
```

Add Type Operation

Signature

```
        IncarnationNumber add_type (
            in CosTrading::ServiceTypeName name,
            in Identifier if_name,
            in PropStructSeq props,
            in ServiceTypeNameSeq super_types
```

```
) raises (
    CosTrading::IllegalServiceType,
    ServiceTypeExists,
    InterfaceTypeMismatch,
    CosTrading::IllegalPropertyName,
    CosTrading::DuplicatePropertyName,
    ValueTypeRedefinition,
    CosTrading::UnknownServiceType,
    DuplicateServiceTypeName
);
```

Function

The add_type operation enables the creation of new service types in the service type repository. The caller supplies the “name” for the new type, the identifier for the interface associated with instances of this service type, the properties definitions for this service type, and the service type names of the immediate super-types to this service type.

If the type creation is successful, an incarnation number is returned as the value of the operation. Incarnation numbers are opaque values that are assigned to each modification to the repository’s state. An incarnation number can be quoted when invoking the list_types operation to retrieve all changes to the service repository since a particular logical time. (Note: IncarnationNumber is currently declared as a struct consisting of two unsigned longs; what we really want here is an unsigned hyper [64-bit integer]. A future revision task force should modify this when CORBA systems support IDL 64-bit integers.)

- If the “name” parameter is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If the type already exists, then the ServiceTypeExists exception is raised.
- If the “if_name” parameter is not a sub-type of the interface associated with a service type from which this service type is derived, such that substitutability would be violated, then the InterfaceTypeMismatch exception is raised.
- If a property name supplied in the “props” parameter is malformed, the CosTrading::IllegalPropertyName exception is raised.
- If the same property name appears two or more times in the “props” parameter, the CosTrading::DuplicatePropertyName exception is raised.
- If a property value type associated with this service type illegally modifies the value type of a super-type’s property, or if two super-types incompatibly declare value types for the same property name, then the ValueTypeRedefinition exception is raised.
- If one of the ServiceTypeNames in “super_types” is malformed, then the CosTrading::IllegalServiceType exception is raised.

- If one of the ServiceTypeNames in “super_types” does not exist, then the CosTrading::UnknownServiceType exception is raised.
- If the same service type name is included two or more times in this parameter, the DuplicateServiceTypeName exception is raised.

Remove Type Operation

Signature

```
void remove_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    HasSubTypes
);
```

Function

The remove_type operation removes the named type from the service type repository.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.
- If “name” has a service type which has been derived from it, then the HasSubTypes exception is raised.

List Types Operation

Signature

```
ServiceTypeNameSeq list_types (
    in SpecifiedServiceTypes which_types
);
```

Function

The list_types operation permits a client to obtain the names of service types which are in the repository. The “which_types” parameter permits the client to specify one of two possible values:

- all types known to the repository
- all types added/modified since a particular incarnation number

The names of the requested types are returned by the operation for subsequent querying via the describe_type or the fully_describe_type operation.

Describe Type Operation

Signature

```
TypeStruct describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);
```

Function

The describe_type operation permits a client to obtain the details for a particular service type.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.

Fully Describe Type Operation

Signature

```
TypeStruct fully_describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);
```

Function

The fully_describe_type operation permits a client to obtain the details for a particular service type. The property sequence returned in the TypeStruct includes all properties inherited from the transitive closure of its super types; the sequence of super types in the TypeStruct contains the names of the types in the transitive closure of the super type relation.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.

Mask Type Operation

Signature

```
void mask_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    AlreadyMasked
);
```

Function

The mask_type operation permits the deprecation of a particular type (i.e., after being masked, exporters will no longer be able to advertise offers of that particular type). The type continues to exist in the service repository due to other service types being derived from it.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.
- If the type is currently in the masked state, then the AlreadyMasked exception is raised.

Unmask Type Operation

Signature

```
void unmask_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    NotMasked
);
```

Function

The unmask_type undeprecates a type (i.e., after being unmasked, exporters will be able to resume advertisement of offers of that particular type).

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.

- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.
- If the type is not currently in the masked state, then the NotMasked exception is raised.

16.7 Dynamic Property Evaluation interface

```
module CosTradingDynamic {

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

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 an 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, the behavior of the evalDP operation is not specified by this standard. 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 parameterized by dynamic properties. The behavior of such traders in such circumstances is not specified by this standard.

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.

Note – Readonly static properties may not be modified to be dynamic.

16.8 Conformance Criteria

The following 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 Admin interface (as server) provided by the trader implementation under test
- the Link 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 behavior defined for each of the operations in the interface specifications shall be exhibited at the conformance points associated with that behavior.

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

- query trader - supports the Lookup interface
- simple trader - supports the Lookup and Register interfaces
- stand-alone trader - supports the Lookup, Register, and Admin interfaces
- linked trader - supports the Lookup, Register, Admin, and Link interfaces; is also a client for Lookup and Register interfaces
- proxy trader - supports the Lookup, Register, Admin, and Proxy interfaces; is also a client for Lookup interfaces
- full-service trader - supports the Lookup, Register, Admin, Link, and Proxy interfaces; is also a client for Lookup and Register interfaces

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

16.8.1 Conformance Requirements for Trading Interfaces as Server

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

Lookup Interface

An implementation claiming conformance to the Lookup interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Lookup interface as documented in "Lookup" on page 16-30.

An implementation claiming conformance to the Lookup interface as server shall also support the OfferIterator interface as server as documented in "Offer Iterator" on page 16-35.

Register Interface

An implementation claiming conformance to the Register interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Register interface as documented in "Register" on page 16-36, 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 be able to assume the client role for the DynamicPropEval interface.
- 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.

Admin Interface

An implementation claiming conformance to the Admin interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Admin interface as documented in “Admin” on page 16-46.

An implementation claiming conformance to the Admin interface as server shall also support the OfferIdIterator interface as server as documented in “Offer Id Iterator” on page 16-45.

Link Interface

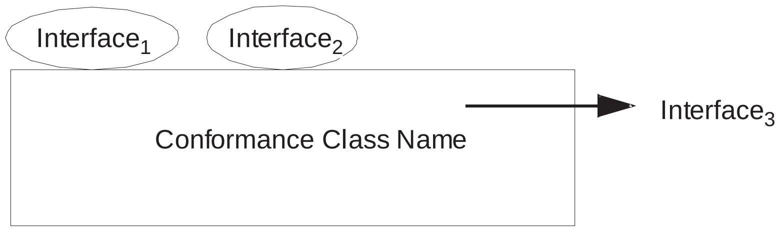
An implementation claiming conformance to the Link interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Link interface as documented in “Link” on page 16-49.

Proxy Interface

An implementation claiming conformance to the Proxy interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Proxy interface as documented in “Proxy” on page 16-54.

16.8.2 Conformance Requirements for Implementation Conformance Classes

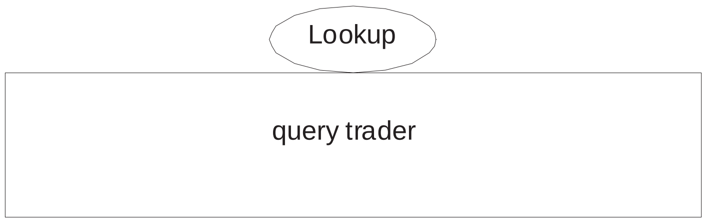
In the sections below, the following graphical notation is used:



The meaning of this notation is as follows:

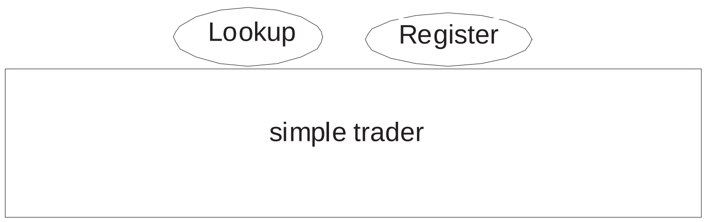
- The rectangle represents an implementation of "Conformance Class Name."
- The ellipses on the surface of the rectangle represent the interfaces supported by this implementation.
- The arrows to the right indicate that traders of this conformance class act as clients to other traders via the named interface.

Query Trader



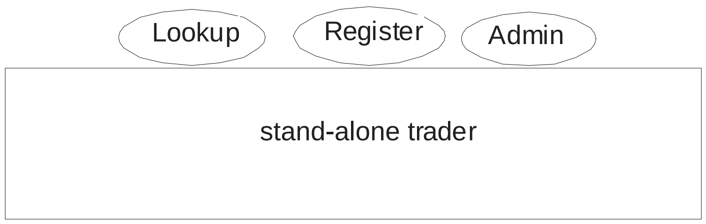
A trading object service implementation claiming conformance to the query trader conformance class shall meet the conformance requirements of the Lookup interface as server.

Simple Trader



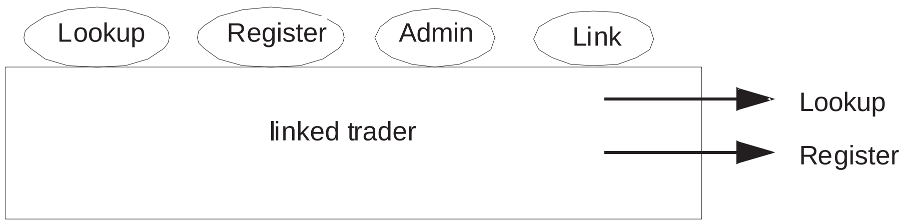
A trading object service implementation claiming conformance to the simple trader conformance class shall meet the conformance requirements of the Lookup and Register interfaces as server.

Stand-alone Trader



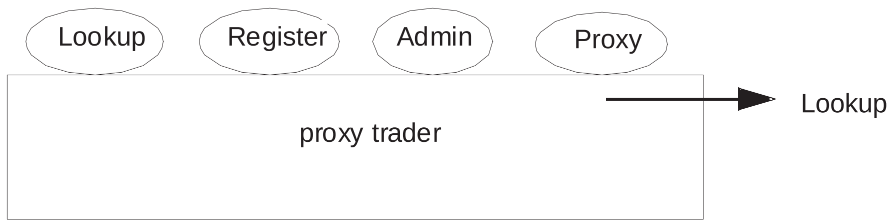
A trading object service implementation claiming conformance to the stand-alone trader conformance class shall meet the conformance requirements of the Lookup, Register, and Admin interfaces as server.

Linked Trader



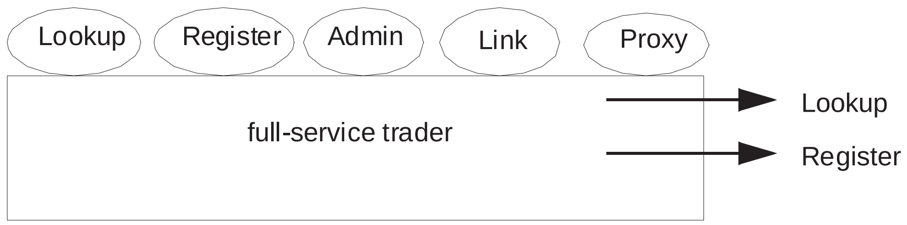
A trading object service implementation claiming conformance to the linked trader conformance class shall meet the conformance requirements of the Lookup, Register, Admin, and Link interfaces as server.

Proxy Trader



A trading object service implementation claiming conformance to the proxy trader conformance class shall meet the conformance requirements of the Lookup, Register, Admin, and Proxy interfaces as server.

Full-service Trader



A trading object service implementation claiming conformance to the full-service trader conformance class shall meet the conformance requirements of the Lookup, Register, Admin, Link, and Proxy interfaces as server.

Appendix A CORBA OMG IDL based Specification of the Trading Function

This appendix provides the CORBA OMG IDL specification of the interface signature for the trading function's computational specification. It specifies the signature for each computational operation in OMG IDL, according to the functional description (signature and semantics) provided in the body of this chapter.

A.1 OMG 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
        ) raises (
            IllegalServiceType,
            UnknownServiceType,
            IllegalConstraint,
            IllegalPreference,
            IllegalPolicyName,
            PolicyTypeMismatch,
            InvalidPolicyValue,
            IllegalPropertyName,
            DuplicatePropertyName,
            DuplicatePolicyName
        );
    };

    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
            );
        };

        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
        );
    };

    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
        );
    };

    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_def_follow_policy (in FollowOption policy);
        FollowOption set_max_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
        );
    };

    interface OfferIterator {

        unsigned long max_left (
        ) raises (
            UnknownMaxLeft
        );

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

        void destroy ();
    };

    interface OfferIdIterator {
        unsigned long max_left (
        ) raises (
            UnknownMaxLeft
        );

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

        void destroy ();
    };

}; /* end module CosTrading */
```

A.2 Dynamic Property Module

```
module CosTradingDynamic {

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

    interface DynamicPropEval {

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

    struct DynamicProp {
         DynamicPropEval eval_if;
         CORBA::TypeCode returned_type;
        any extra_info;
    };
}; /* end module CosTradingDynamic */
```

A.3 Service Type Repository Module

```
module CosTradingRepos {

    interface ServiceTypeRepository {

    // local types
        typedef sequence<CosTrading::ServiceTypeName>
ServiceTypeNameSeq;
        enum PropertyMode {
            PROP_NORMAL, PROP_READONLY,
            PROP_MANDATORY, PROP_MANDATORY_READONLY
        };
        struct PropStruct {
            CosTrading::PropertyName name;
            CORBA::TypeCode value_type;
            PropertyMode mode;
        };
        typedef sequence<PropStruct> PropStructSeq;

        typedef CosTrading::Istring Identifier;  // IR::Identifier
        struct IncarnationNumber {
            unsigned long high;
            unsigned long low;
        };
        struct TypeStruct {
            Identifier if_name;
            PropStructSeq props;
            ServiceTypeNameSeq super_types;
            boolean masked;
            IncarnationNumber incarnation;
        };

        enum ListOption { all, since };
        union SpecifiedServiceTypes switch ( ListOption ) {
            case since: IncarnationNumber incarnation;
        };

    // local exceptions
        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;
        };

    // attributes
        readonly attribute IncarnationNumber incarnation;

    // operation signatures
        IncarnationNumber add_type (
    in CosTrading::ServiceTypeName name,
    in Identifier if_name,
    in PropStructSeq props,
    in ServiceTypeNameSeq super_types
) raises (
    CosTrading::IllegalServiceType,
    ServiceTypeExists,
    InterfaceTypeMismatch,
    CosTrading::IllegalPropertyName,
    CosTrading::DuplicatePropertyName,
    ValueTypeRedefinition,
    CosTrading::UnknownServiceType,
    DuplicateServiceTypeName
);

void remove_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    HasSubTypes
);

ServiceTypeNameSeq list_types (
    in SpecifiedServiceTypes which_types
);

TypeStruct describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);

TypeStruct fully_describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType
        );

        void mask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            AlreadyMasked
        );

        void unmask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            NotMasked
        );

    };
}; /* end module CosTradingRepos */
```

Appendix B OMG Constraint Language BNF

This appendix provides the BNF specification of the CORBA standard constraint language; it is used for specifying both the constraint and preference expression parameters to various operations in the trader interfaces.

A statement in this language is an Istring. Other constraint languages may be supported by a particular trader implementation; the constraint language used by a client of the trader is indicated by embedding “<<Identifier major.minor>>” at the beginning of the string. If such an escape is not used, it is equivalent to embedding “<<OMG 1.0>>” at the beginning of the string.

B.1 Language Basics

B.1.1 Basic Elements

Both the constraint and preference expressions in a query can be constructed from property names of conformant offers and literals. The constraint language in which these expressions are written consists of the following items (examples of these expressions are shown in square brackets below each bulleted item):

- comparative functions: == (equality), != (inequality), >, >=, <, <=, ~ (substring match), in (element in sequence); the result of applying a comparative function is a boolean value
  [“Cost < 5” implies only consider offers with a Cost property value less than 5; “’Visa’ in CreditCards” implies only consider offers in which the CreditCards property, consisting of a set of strings, contains the string ’Visa’]
- boolean connectives: and, or, not
  [“Cost >= 2 and Cost <= 5” implies only consider offers where the value of the Cost property is in the range 2 <= Cost <= 5]
- property existence: exist
- property names
- numeric and string constants
- mathematical operators: +, -, *, /
  [“10 < 12.3 * MemSize + 4.6 * FileSize” implies only consider offers for which the arithmetic function in terms of the value of the MemSize and FileSize properties exceeds 10]
- grouping operators: (,)

Note that the keywords in the language are case sensitive.

B.1.2 Precedence Relations

The following precedence relations hold in the absence of parentheses, in the order of highest to lowest:

```
() exist unary-minus
not
* /
+ -
~
in
== != < <= > >=
and
or
```

B.1.3 Legal Property Value Types

While one can define properties of service types with arbitrarily complex OMG IDL value types, only the following property value types can be manipulated using the constraint language:

- boolean, short, unsigned short, long, unsigned long, float, double, char, Ichar, string, Istring
- sequences of the above types

The "exist" operator can be applied to any property name, regardless of the property's value type.

B.1.4 Operator Restrictions

exist	can be applied to any property
~	can only be applied if left operand and right operand are both strings or both Istrings
in	can only be applied if the left operand is one of the simple types described above and the right operand is a sequence of the same simple type
==	can only be applied if the left and right operands are of the same simple type
!=	can only be applied if the left and right operands are of the same simple type
<	can only be applied if the left and right operands are of the same simple type
<=	can only be applied if the left and right operands are of the same simple type
>	can only be applied if the left and right operands are of the same simple type
>=	can only be applied if the left and right operands are of the same simple type
+	can only be applied to simple numeric operands
-	can only be applied to simple numeric operands
*	can only be applied to simple numeric operands

/	can only be applied to simple numeric operands

<, <=, >, >= comparisons imply use of the appropriate collating sequence for characters and strings; TRUE is greater than FALSE for booleans.

B.1.5 Representation of Literals

boolean	TRUE or FALSE
integers	sequences of digits, with a possible leading + or -
floats	digits with decimal point, with optional exponential notation
characters	char and Ichar are of the form '<char>', string and Istring are of the form '<char><char>+'; to embed an apostrophe in a string, place a backslash (\) in front of it; to embed a backslash in a string, use \\.

B.2 The Constraint Language BNF

B.2.1 The Constraint Language Proper in Terms of Lexical Tokens

```
<constraint>:=/* empty */
    |   <bool>

<preference>:=/* <empty> */
    |   min <bool>
    |   max <bool>
    |   with <bool>
    |   random
    |   first

<bool>:=<bool_or>

<bool_or>:=<bool_or> or <bool_and>
    |   <bool_and>

<bool_and>:=<bool_and> and <bool_compare>
    |   <bool_compare>

<bool_compare>:=<expr_in> == <expr_in>
    |   <expr_in> != <expr_in>
    |   <expr_in> <  <expr_in>
    |   <expr_in> <= <expr_in>
    |   <expr_in> >  <expr_in>
    |   <expr_in> >= <expr_in>
    |   <expr_in>

<expr_in>:=<expr_twiddle> in <Ident>
    |   <expr_twiddle>

<expr_twiddle>:=<expr> ~ <expr>
    |   <expr>

<expr>:=<expr> + <term>
    |   <expr> - <term>
    |   <term>

<term>:=<term> * <factor_not>
    |   <term> / <factor_not>
    |   <factor_not>

<factor_not>:=not <factor>
    |   <factor>

<factor>:=( <bool_or> )
    |   exist <Ident>
    |   <Ident>
    |   <Number>
    |   - <Number>
    |   <String>
    |   TRUE
    |   FALSE
```

B.2.2 "BNF" for Lexical Tokens up to Character Set Issues

```
<Ident>:=<Leader> <FollowSeq>

<FollowSeq>:=/* <empty> */
    |   <FollowSeq> <Follow>

<Number>:=<Mantissa>
    |   <Mantissa> <Exponent>
```

```
<Mantissa>:=<Digits>
    |   <Digits> .
    |   . <Digits>
    |   <Digits> . <Digits>

<Exponent>:=<Exp> <Sign> <Digits>

<Sign>:=+
    |   -

<Exp>:= E
    |   e

<Digits>:=<Digits> <Digit>
    |   <Digit>

<String>:=’ <TextChars> ’

<TextChars>:=/* <empty> */
    |   <TextChars> <TextChar>

<TextChar>:=<Alpha>
    |   <Digit>
    |   <Other>
    |   <Special>

<Special>:=\\
    |   \’
```

B.2.3 Character Set Issues

The previous BNF has been complete up to the non-terminals <Leader>, <Follow>, <Alpha>, <Digit>, and <Other>. For a particular character set, one must define the characters which make up these character classes.

Each character set which the trading service is to support must define these character classes. This appendix defines these character classes for the ASCII character set.

```
<Leader>:=<Alpha>
```

```
<Follow>:=<Alpha>
    |   <Digit>
    |   _


<Alpha> is the set of alphabetic characters [A-Za-z]
<Digit> is the set of digits [0-9]
<Other> is the set of ASCII characters that are not <Alpha>, <Digit>,
or <Special>
```

Appendix C OMG Constraint Recipe Language

This appendix describes the recipe language used to construct the secondary constraint expression when resolving proxy offers; the secondary constraint expression is constructed from the primary constraint expression and the properties associated with the proxy offer.

A statement in this language is an Istring. Other recipe languages may be supported by a particular trader implementation; the recipe language used by a client of the trader is indicated by embedding “<<Identifier major.minor>>” at the beginning of the string. If such an escape is not used, it is equivalent to embedding “<<OMG 1.0>>” at the beginning of the string.

While the nested invocation of the Trader behind the proxy assumes support for the Lookup interface, the secondary constraint expression does not necessarily need to conform to the language described in Appendix B.

C.1 The Recipe Syntax

The rewriting from primary to secondary works similarly to formatted output in a variety of programming languages and systems. It is patterned after the variable replacement syntax of the Bourne and Korn shells on most UNIX systems.

When it is time to construct the secondary constraint expression from the recipe, the algorithm is as follows:

```
while not end of recipe
    fetch the next character from the recipe
    if not a ‘$’ character
        append the character to the secondary constraint
    else
        fetch next character from the recipe
        if a ‘*’ character
            append the entire primary constraint to the secondary
constraint
        else if not a ‘(’ character
            append the character to the secondary constraint
        else
            collect characters up to a ‘)’ character, discarding ‘)’
            lookup property with that name
            append formatted value of that property to secondary
constraint
```

C.2 Example

Assume a proxy offer has been exported to a trader with the following properties:

```
<Name, ‘MyName’>, <Cost, 42>, <Host, ‘x.y.co.uk’>
```

and with the following recipe:

```
“Name == $(Name) and Cost == $$$(Cost)”
```

The above algorithm will generate the following secondary constraint for the nested call to the trader behind the proxy:

```
“Name == ‘MyName’ and Cost == $42”
```