Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

The Property Repository

Gray Girling, Mike Beasley

Abstract

The promotion and advertising of existing and forthcoming electronic services and their efficient location by potential clients are important prerequisites for the wide scale deployment of an electronic services market place.

Trading services provide advertising and location functions for services but current implementations are usable only at runtime, and the information they give about services is not extensible.

This document describes the interface to a property repository that can support an extensible range of property information types, and an implementation using a relational database management system.

APM.1384.01	**Approved**	13th December 1994
	Technical Report	

Distribution:
Supersedes:
Superseded by:

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

The Property Repository

The Property Repository

Gray Girling, Mike Beasley

APM.1384.01

13th December 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

1	**1**	**Introduction**
1	1.1	The business problem
1	1.2	The technical problem
1	1.3	The solution
2	1.4	Overview
5	**2**	**Requirements**
5	2.1	Support ODP-like trader interfaces
5	2.2	Implement links rather than contexts
6	2.3	Support a wider variety of property value type
6	2.4	Support an extensible set of property value types
6	2.5	Allow new property names to be registered dynamically
6	2.6	Provide a scheme to avoid property name clashes
7	2.7	Provide information about dynamically creatable services
7	2.8	Allow dynamically creatable services to be parameterised
7	2.9	Ease of use by unintelligent clients
7	2.10	Flexible use by intelligent clients
7	2.11	Use at development time
7	2.12	Do not constrain interface reference design
8	2.13	Do not constrain the passing of interface references
8	2.14	Use of existing storage technology for trader information
9	**3**	**Specification**
9	3.1	Overview
9	3.1.1	Sub-Profiles, Property Types and Properties
10	3.1.2	Exporting
10	3.1.3	Interrogation and Importing
10	3.1.4	Type Management
11	3.1.5	Link Management
11	3.1.6	Factories
11	3.2	Typical Use
11	3.2.1	Sub-Profiles and Properties
12	3.2.2	Referring to Offers
13	3.2.3	Export and Use of an Offer
13	3.3	Data Types
14	3.3.1	Underlying Information System
14	3.3.2	Property Representation
14	3.3.3	Property naming
15	3.3.4	Monitor Data
15	3.3.5	Basic Model
16	3.4	Interface Specifications
16	3.4.1	Property Repository
16	3.4.2	Monitor Interface

17 3.4.3 Query Manager Interface
18 3.4.4 Export Manager Interface
18 3.4.5 Reference Manager Interface
19 3.4.6 Property Repository Factory Interface
19 3.4.7 Link Manager Interface
19 3.4.8 Traditional Trader Retrieval Interface
20 3.4.9 Properties Sub-Profile Manager Interface
21 3.4.10 Type Manager Interface
22 3.5 Interface Interrelationships
22 3.6 The Basic Properties Sub-profile
22 3.6.1 Purpose
23 3.6.2 Interface References
23 3.6.3 Interface Signatures

25 4 IDL Specifications
25 4.1 Data Types
25 4.1.1 Underlying Information System
25 4.1.2 Property Representation
25 4.1.3 Property naming
26 4.1.4 Monitor Data
26 4.1.5 Basic Model
26 4.2 Interface Specifications
26 4.2.1 Property Repository
26 4.2.2 Monitor Interface
27 4.2.3 Query Manager Interface
27 4.2.4 Export Manager Interface
28 4.2.5 Reference Manager Interface
28 4.2.6 Property Repository Factory Interface
28 4.2.7 Link Manager Interface
29 4.2.8 Traditional Trader Retrieval Interface
29 4.2.9 Properties Sub-Profile Manager Interface
30 4.2.10 Type Manager Interface

33 5 Comparison with Other Traders
33 5.1 ODP Trader
33 5.1.1 Search
33 5.1.2 Select
33 5.1.3 List Offer Details
34 5.1.4 Export
34 5.1.5 Withdraw
34 5.1.6 Replace
34 5.1.7 Add Link
34 5.1.8 Remove Link
34 5.1.9 Modify Link
34 5.1.10 List Link Details
34 5.2 ANSA Trader
34 5.2.1 Register, RegisterMonitor
35 5.2.2 LookUpOne
35 5.2.3 LookUpAll
35 5.2.4 Link
35 5.2.5 UnLink

35 5.2.6 Contexts
35 5.2.7 Select
35 5.2.8 SelectWithPolicy
35 5.2.9 Withdraw
35 5.2.10 Trading context creation
36 5.3 ANSAware Trader
36 5.3.1 Register
36 5.3.2 Lookup
36 5.3.3 Delete
36 5.3.4 AddName
36 5.3.5 ListNames
36 5.3.6 DelName
36 5.3.7 BindContext
36 5.3.8 UnbindContext
36 5.3.9 ProxyExport
37 5.3.10 DeleteProxy
37 5.3.11 AddType
37 5.3.12 MaskType
37 5.3.13 UnmaskType
37 5.3.14 DelType
37 5.3.15 ListTypes

39 6 Design
39 6.1 Overview
39 6.2 Design Issues
39 6.2.1 Behavioural Semantics Specifications
40 6.2.2 Client/Server Symmetry
40 6.2.3 Development Process
40 6.2.4 Interface Signatures in the Model
41 6.2.5 Extensibility
41 6.2.6 Matching Algorithms - the options
43 6.2.7 Matching Algorithms - the decision
43 6.2.8 Design of Interface References
44 6.2.9 Federation
45 6.2.10 Dynamic Services and Orbix
45 6.3 Representation in the Database
45 6.3.1 Property Type Specification
45 6.3.2 Property Representation Strategies
47 6.4 Options for implementing repositories
49 6.5 Use of DBMS and SQL by the various interfaces
49 6.5.1 Repository Factory
49 6.5.2 Query Manager
49 6.5.3 QuerySession
49 6.5.4 Selection
49 6.5.5 Export Manager
50 6.5.6 Reference Manager
50 6.5.7 Link Manager
51 6.5.8 Type Manager
51 6.5.9 Old Trader Retrieval

53	**7**	**Database Tables used by the Repository**
53	7.1	Tables which must be present
53	7.1.1	Sub-Profile information
53	7.1.2	Property Type Information
53	7.1.3	Property Information
53	7.1.4	Offer Information
54	7.1.5	Type Information
54	7.1.6	Link Information
54	7.2	Tables which are normally present
55	7.3	Additional tables containing property values
57	**8**	**Planned Evolution of Implementation**
57	8.1	Properties Sub-profile Support
57	8.1.1	Support for signatures
58	8.2	Repository Support
58	8.2.1	Support for the property repository factory
58	8.3	Traditional Trader Support
58	8.3.1	Constraint Expression Support
58	8.3.2	Search Policy Support
58	8.4	Summary of Timetable
58	8.4.1	Short term
59	8.4.2	Medium term
59	8.4.3	Long term
59	8.5	Future Evolution
61	**9**	**Experience of initial implementation**

1 Introduction

1.1 The business problem

The promotion and advertising of existing and forthcoming electronic services and their efficient location by potential clients are important prerequisites for the wide scale deployment of an electronic services market place.

1.2 The technical problem

Trading services provide advertising and location functions for services but current implementations are only usable at runtime, and the information they give about services is not extensible.

Exploitation of existing database technology (e.g. relational or object) would enable trading services to perform a wider range of information manipulation and would facilitate their integration into existing information systems.

The range of information associated with electronic services continues to increase with the development of infrastructure capable of providing different qualities of service (such as those related to performance and dependability) or distribution transparencies (such as migration, and federation). In addition the use of tools in the development process, leading ultimately to the provision of new services, increasingly uses and generates service-specific information. Such information requires dissemination to both service producers and consumers.

New trading services capable of meeting these changing information needs are required more and more frequently. Nonetheless the functionality of traditional trading interfaces must continue to fulfil existing uses.

1.3 The solution

Trading has been an important component of the ANSA distributed system architecture for some time [APM 1005.1 93], and it is also included in ODP [ISO ODP Trader 93]. The purpose of a trader is to accept *service offers* from potential providers (exporters), so that the information can be given to potential users (importers). An offer contains an *interface reference*, which can be used by a client to invoke operations on the service, together with the *interface type* of the service (the names of its operations, with their parameter and result types), and some *properties*, which allow a potential user to choose between services which are functionally equivalent. In addition, a trader can have *links* to other traders, so that a potential user can ask for them to be searched too.

Existing traders are defined in terms of their ability to provide information about the interface type (e.g. interface signature), location and other properties of service offers; and to use this information to identify desired services with which (type safe) interaction can take place.

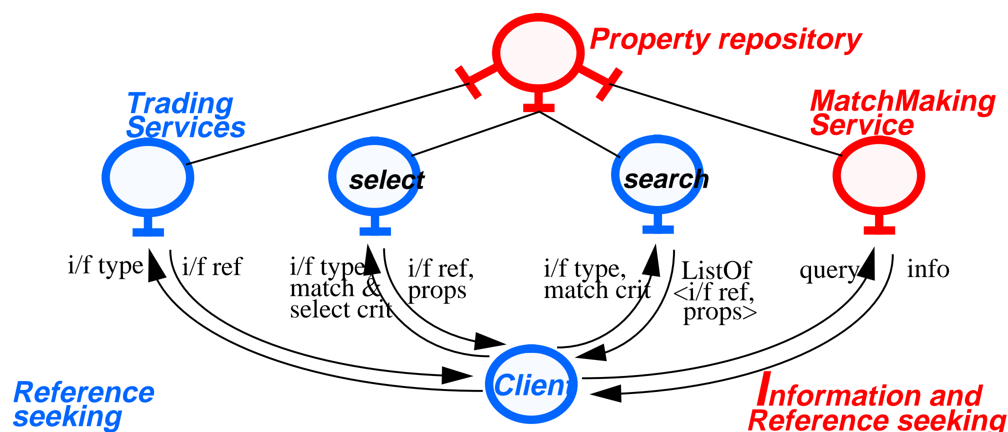
The solution addressed here is the design of a distributed service consistent with previous traders but:

- capable of implementation based on a relational database management system; and,
- able to extend the range of types of properties that it holds and uses.

Section 1.2 gives our reasons for using existing database technology; we chose relational rather than object databases for a number of reasons - it is mature technology, widely available, with which the team have previous experience.

The properties required by existing traders form a basic set that must always be present. However, types of information with purposes other than locating servers and binding to them might additionally be held by the proposed service[1]. In particular we are taking a whole life cycle view of Object Management. Traditional trader functionality is a subset of the functionality of our new service, and so the term *Property Repository* (rather than "trader") therefore more accurately describes its purpose.

Figure 1.1: Trading and MatchMaking



1.4 Overview

This document follows a logical progression from Requirements through Specification to Design.

Requirements:

- Chapter 2 outlines the requirements which the Property Repository must satisfy.

Specification:

- in Chapter 3, a set of interfaces are proposed which are consistent with the requirements
- these interfaces are defined in CORBA IDL in Chapter 4
- the interfaces are compared with those offered by the ODP, ANSA and ANSAware traders in Chapter 5.

1. Such requirements are outlined in [APM 1140.0.6 94].

Design:

- The main design issues are discussed in Chapter 6, which is mainly concerned with how to represent the repository information in database tables, and how to access those tables
- Chapter 7 gives more detail on the use of database tables.

Finally:

- Chapter 8 outlines an evolutionary path by which a prototype can be enhanced, and
- Chapter 9 presents some of the lessons from the initial implementation.

2 Requirements

The following requirements must be met in the specification of the Property Repository.

2.1 Support ODP-like trader interfaces

The functionality of the Property Repository, as will be seen from the following requirements, is more than that of a trader, but it must include the functionality of a trader. Interfaces should be similar to ODP (the emerging standard), with divergences justified. The following figures list the operations of the ODP trader [ISO ODP Trader 93], with the corresponding operations of the ANSA [APM 1005.1 93] and ANSAware [APM ANSAware 93] traders for comparison purposes:

Table 2.1: Import/Export

ODP Trader	ANSA Trader	ANSAware Trader
Export	Register	Register
Search + List Offer	LookUpAll	Lookup
Export	RegisterMonitor	ProxyExport
Select	LookUpOne	Lookup
Withdraw	(Offer.withdraw)	Delete, (ProxyDelete)
Replace = Withdraw+Export		

Table 2.2: Link management

ODP Trader	ANSA Trader	ANSAware Trader
Add Link	Link	BindContext, AddName
Remove Link	Unlink	UnbindContext, DelName
List Link Details	Contexts	ListNames
(Resolve)	Select	
(Resolve)	SelectWithPolicy	
Modify Link = Remove Link+Add Link		

2.2 Implement links rather than contexts

This requirement is a consequence of the requirement to be similar to ODP.

Older traders such as that in ANSAware [APM ANSAware 93] have a notion of “context” to allow the trader’s offer space to be subsetted. A context can be bound to a remote trader, and local and remote lookup environments are therefore treated differently.

In ODP, there is no notion of "context". The subsetting of the offer space is achieved by the use of computationally separate traders, with links between them. This unifies the treatment of local and remote environments.

The Property Repository is required to follow ODP in this respect. In addition, there must be a factory mechanism which allows multiple computational traders to exist in the same engineering capsule (e.g. UNIX process).

2.3 Support a wider variety of property value type

Existing traders allow "properties" to be associated with service offers. ODP [ISO ODP Trader 93] suggests that the values of these properties can be interpreted as numbers or as strings; the ANSAware trader [APM ANSAware 93] also allows a property to be interpreted as a set of strings.

The Property Repository must support structured information as property values, and must therefore support a wider range of property value types than do existing traders.

2.4 Support an extensible set of property value types

Each property associated with a given service offer conforms to some *Property Type* that determines the structure of the information used to represent it.

To achieve extensibility, some means should be provided whereby support for new property types can be added to the service, preferably dynamically during service operation.

2.5 Allow new property names to be registered dynamically

Each property associated with a given service offer has an associated name. The introduction of complex property value types will require that the association between a name and the corresponding value type will need to be pre-registered. Means should be provided whereby this registration can be performed while the service is running.

It would be desirable if there was no need to pre-register property names if the property value has a simple type like number or string (registration would be automatic on first export, and the value type assumed from the value supplied).

2.6 Provide a scheme to avoid property name clashes

It is likely that different people will define groups of related properties for various purposes (e.g. a group of properties related to Quality of Service, or dependability). There should be some mechanism for avoiding name clashes, i.e. avoiding global naming.

Such a naming scheme may also facilitate the selection of a subset of the available properties to be returned from a query operation.

2.7 Provide information about dynamically creatable services

In addition to the registration of information about existing services it must be possible to register the same information about services that can be created or located on demand. Also information required to support the process of service creation must be held (e.g. instantiation parameters).

The property repository must incorporate a means to associate with an offer an export policy controller [ISO ODP Trader 93], (e.g. a factory service) which will be called to create (or locate) the required service when it is requested. The equivalent concepts in the ANSA [APM 1005.1 93] and ANSAware [APM ANSAware 93] traders are a monitor and a proxy service respectively.

2.8 Allow dynamically creatable services to be parameterised

It must be possible for a monitored offer to be associated with a range of services with the same interface type but with different property values. The property values will be chosen according to the requirements placed by the importer. The properties associated with the original export will be those that are common to all the services which can be offered.

Traditional concepts of properties and constraints are not very good at describing the possible values of the variable properties (for example, the monitor may be able to create services with response times of 1, 2, 5 and 10 seconds). This is an area which will undoubtedly need to be addressed further.

2.9 Ease of use by unintelligent clients

It must be straightforward for a Property Repository client to be able to obtain a single interface reference from an offer which satisfies a simple constraint.

2.10 Flexible use by intelligent clients

It must also be possible for an intelligent client of a Property Repository to gather information on available services by making arbitrary enquiries from different sources, and then to make a choice between the offers available.

This process must be possible without the dynamic creation of any services other than that which is finally chosen.

2.11 Use at development time

It must be possible to use the Property Repository at development time to advertise a service which is not actually running. This implies that it must be possible to export offers which do not contain an interface reference, and to perform queries which extract information from such offers.

2.12 Do not constrain interface reference design

The design of the Property Repository must not place any constraints on designers of interface references; the format of interface references must be extensible to allow for encapsulation.

However, it may be assumed that an interface reference can be converted to a string of no more than a few thousand bytes in length.

2.13 Do not constrain the passing of interface references

The design of the Property Repository must not assume that all interface references come from the Property Repository. It must always be possible for interface references to be passed around, and for all the necessary binding information to be extracted from an interface reference without any explicit involvement of the Property Repository.

2.14 Use of existing storage technology for trader information

An extensible range of property types enables many different types of information to be included in the scope of the property repository. This is likely to result in a requirement to store far larger quantities of data than, for example, is the case with the ANSAware trader. With the greater volume of data come greater concerns over its integrity and durability.

The same extensibility also requires support by an underlying data model that is itself extensible.

These concerns must be met through the use of existing technology, not the development of new technology. This requirement could be satisfied by using an RDBMS or an object database, for example.

3 Specification

This specification assumes that the reader has a general grounding in trading, based on previous reading of ANSA [APM 1005.1 93] and ODP [ISO ODP Trader 93] documents.

3.1 Overview

3.1.1 Sub-Profiles, Property Types and Properties

In order to reduce the probability of name clashes between independent property definers, the property name space is partitioned into *sub-profiles*[1], and a property name consists of two parts, a sub-profile name and a name which is unique within that sub-profile. The independent definers now only have to reach agreement on the use of sub-profile names.

There is a set of property types. A property type consists of a name, a description, an indication of how properties of that type are stored in the database[2] and a description of the structure of the property[3]. Currently, a property can be a basic value, or a sequence, structure, sequence of structures or sequence of sequences of basic values. This could be extended.

A property name is associated with a sub-profile, to which it belongs, and a property type, which defines what values the property is allowed to take and how those values are stored in the database.

Suppose, for example, that we are concerned with banking services. We might then have:

- a ‘Banking’ sub-profile
- a property type ‘BankNameType’, which is a string; bank names might be stored in a different database table from other strings
- a property name ‘BankName’ in the ‘Banking’ sub-profile, associated with the ‘BankNameType’ property type.

There is a basic set of property types, to deal with properties whose values are types (signatures), interface (object) references, numbers and strings of various lengths. We make the reasonable assumption that an interface reference can be converted to, and constructed from, a string. By passing interface references into the repository as strings, we avoid the need for the repository to know anything about the structure of references.

1. the reason for the use of the name ‘*sub*-profile’ is that the whole property name space is regarded as the property profile of the repository

2. currently a table name

3. currently an Orbix ‘type code’ string

Facilities are provided for creating, interrogating and deleting sub-profiles, property types and named properties.

3.1.2 Exporting

The 'export' operation takes a list of properties, and returns an offer identifier. Offers can be service offers, proxy (monitored) offers or indeed offers which include no interface (object) reference at all. The distinction between the three cases depends on a conventional use of property names to identify service interface references and monitor interface references.

It would be desirable to distinguish between offers that can be bound to and offers that cannot, based on what properties are present. This should be addressed in the future.

It is also possible to add properties to an offer, remove properties from an offer, list the properties of an offer or withdraw an offer.

3.1.3 Interrogation and Importing

The contents of the repository can be interrogated in three ways:

1. the 'Query' operation allows interrogation of the underlying database using the full facilities of the SQL 'select' statement. This is very powerful and flexible, but requires an understanding of the structure of the database to be used at all. Transactional support is provided so that a set of queries can have a consistent view of the database.
2. the 'Select' operation, similar to that in existing traders, searches for offers of types conformant to a given type. A constraint expression, in the ANSAware constraint language, can be supplied, together with a search policy which determines if and how linked traders are searched. A single service reference is returned, together with its simple properties. If a proxy offer is found, the service reference is obtained from the monitor.
3. the 'Search' operation differs from 'Select' in that multiple offers can be returned, and the monitor mechanism is not invoked. It is also possible to specify a list of properties that are to be returned, in which case offers will only be included if they have values for that set of properties.

Note that the choice of languages for expressing constraints is made entirely for reasons of ease of implementation:

- SQL is chosen because we are using an RDBMS for storage, and an SQL query can be passed straight through to the database
- the ANSAware constraint language is chosen because we can reuse code from the ANSAware trader for parsing it; something similar is obviously envisaged in ODP

There is no suggestion intended that either SQL or the ANSAware constraint language is considered to be an ideal query language for use in traders.

3.1.4 Type Management

The ultimate intention is to use automated conformance checking of interface types; the initial implementation, however, includes an assertion-based type checker along the same lines as that in the ANSAware trader. Types can be added and deleted, and conformance relationships asserted. There are also interrogation functions.

3.1.5 Link Management

Links between traders can be set up, removed and interrogated. There is also support for link properties to control searching and selecting.

Some decisions have to be made about how links are treated in the trader's database.

The first decision is between treating links as (special) offers and treating links as a different sort of object. The options are:

- a link is stored as an offer, with link properties as offer properties, and a special property to indicate that it is a link rather than an offer.
- links are stored completely separately from offers.

It has been decided to keep links and offers distinct. Keeping them together confuses what are really distinct concepts, and makes SQL querying more difficult. Links will have associated properties; only strings need to be supported.

The `ListLinks` operation could be modified to do an arbitrary query on link properties and only return those links which match.

3.1.6 Factories

The ANSAware trader [APM ANSAware 93] has the concept of 'Contexts' to partition the offer space. Following ODP, the Property Repository replaces this with links between computationally separate traders. The *repository factory* supports the creation of more than one computational trader within the same engineering capsule (e.g. UNIX process).

As a result of using the factory, followed by deletion of links, unreferenced trading objects might exist. Detection of this situation might be difficult; having done so, a distributed garbage collection algorithm would be required to reclaim resources.

3.2 Typical Use

3.2.1 Sub-Profiles and Properties

The property repository allows information to be stored about offers. An offer can refer to either:

- a single service, or
- a group of services with the same interface type but with different properties (a parameterised offer).

Information about an offer is held as collections of named properties, grouped together according to some common purpose in sub-profiles. These sub-profiles may be designed independently by different *Sub-Profile Designers* for different purposes. For example:

- property types for "transfer syntax" and "transfer protocols profile" might be required together to support federation
- performance support may require property types "capsule manager interface reference" and "performance quality of service".

It may happen that two such design authorities define different sub-profiles for the same purpose (e.g. facilitating RPC access). The property repository

allows this, since it is only aware of sub-profile names, but it is quite likely that the differences in sub-profile definition represent a federation problem that will need to be resolved. The federation mechanisms could make use of information provided by the repository to solve this problem.

Following the creation of an offer, properties can be associated with it. These properties may come independently from a number of sub-profiles. For example:

- a server's infrastructure may provide basic properties
- a related development system may provide specification properties
- a technical author may provide a manual page (possibly as a *URL* (Uniform Resource Locator) as used in the World Wide Web).

The different *Property Providers* may be unaware of the other properties which have been associated with offers, but the property repository provides them with the functions they need for discovering this information.

The set of named properties provided by a repository is known as its *Properties Profile*. This may vary between repositories and, over time, within the same repository. The properties profile is defined as a set of sub-profiles. Extension of a repository is achieved by the inclusion of additional sub-profiles.[1]

3.2.1.1 *The "Basic" properties sub-profile*

The "Basic" sub-profile incorporates the property types that are required to support a traditional trader interface (i.e. properties that represent interface references and interface signatures).

This sub-profile is a mandatory part of every properties profile. It provides property types for:

- server/export policy controller/monitor interface references, and
- signatures.

3.2.1.2 *The "Anonymous" properties sub-profile*

The "Anonymous" properties sub-profile supports traditional named properties, whose values can be integers, strings or lists of strings. It is not necessary to associate property names in this sub-profile with their value types in advance of their use; the first use will set up such an association implicitly, if none already exists.

3.2.2 Referring to Offers

It is important that different property providers have some common means of referring to the same offer. This means is the *offer ID*, which can be found in the following ways:

- creator interaction
  The ID is returned to the provider of the initial set of properties that establishes the offer (as in [ISO ODP Trader 93]); it can then be communicated to other properties providers for use when amending the set of offers.

1. New properties sub-profile specifications will emerge from the analysis of mechanisms supporting federation across different types of boundary [APM 1139.0.3 94] and the construction of an information model for federation [APM.1229.0.8 94].

- key properties
  The ID can be regarded as a pseudo-property. It can be found, using one or more key properties, by the repository's normal database query facilities. Note that an interface reference is not a candidate key property, insofar as there may be many interface references that refer to the same interface ("aliases" are optionally allowed in naming contexts that conform to the ANSA naming model [APM 1003.1 93]).

3.2.3 Export and Use of an Offer

The following is an example use of the repository in which a client binds to a service whose offer identifies a range of services with the same interface type but different properties.

- *Export phase*

  The monitor (or some associated agent) registers itself with the property repository, supplying a set of properties including the monitor's interface reference and some data which the monitor may wish to use to distinguish this offer from others, and describing the range of services which can be provided.

- *Shopping phase*

  The client queries the database of properties held by the property repository to obtain a set of suitable offers and chooses from that set a single offer that will provide the service required.

- *Instantiation phase*

  The client gives the repository a detailed list of its requirements (that it believes the selected offer can fulfil) and the ID of the offer that it has selected. The repository passes these to the monitor.

  The monitor creates or selects a service that possesses the defined properties and returns its interface reference and properties to the repository.

  The repository returns the interface reference to the client, together with any properties which the client has specifically requested.

- *Binding phase*

  The client uses the *service* interface reference and the returned properties to create a binding to the server. These properties may include information relevant to interception and to quality of service (e.g. as required by [APM 1137.1 94]) and their negotiation.

Similar examples can be given for the simpler cases of a non-parametric monitored offer and of an unmonitored offer. The latter case is the simplest of all, in which the service provider registers itself with the repository, and the instantiation phase consists merely of the client passing the offer ID to the repository, which returns the interface reference and properties.

3.3 Data Types

The specification of data types is given in CORBA IDL [OMG CORBA 93] in section 4.1.

3.3.1 Underlying Information System

A number of data types are needed so that the repository can access the underlying database (see figure 3.2). These include:

- a basic database type (`QueryBaseType`)

  This represents one of the base types that can be stored by the database, and may include information such as length and whether null values are permitted.

- a value of a basic database type (`QueryBaseValue`)

  This represents a value of one of the above types. Length information may be implicit, and there may need to be provision for null values.

- a database query specification (`QuerySpec`)

  This consists of a parametric specification of a query together with a sequence of base values to be used as its actual parameters. In dynamic SQL such queries consist of three parts – the (parameterised) text of a query in SQL, a specification of the types of the input parameters in the query, and a set of values of those types which are substituted for the parameters in the query.

3.3.2 Property Representation

The values of `Properties` are constrained to be able to be constructed from a sequence of values of the base types that can be stored in an underlying information system. (Chapter 5 provides some examples of how this can be done for the property types associated with the basic properties sub-profile.)

The type of a property (in terms of `QueryBaseTypes`) is held in a properties sub-profile manager (see section 3.4.9) and relies on a number of types of information.

A `PropertyTypeSpec` provides information enabling its recipient to understand the mapping from different `PropertyTypes` to the tables maintained by the database management system. It also includes the name associated with the `PropertyType`, and a textual description of its purpose.

3.3.3 Property naming

A property (for which the IDL type is `NamedProperty`) has a name (`PropertyName`) and a value (`PropertyValue`). The name consists of two components: a sub-profile name and a name within that sub-profile.

A `PropertyNameProfile` is a set of `PropertyNames` (e.g. giving a definition of the set of properties required in a query).

A property name is associated with a property type by means of a `NamedPropertySpec`. This consists of:

- the property name
- a textual description of its purpose
- a property type name.

A property type is defined in a `PropertyTypeSpec`. This consists of:

- the property type name
- a textual description of its purpose
- the data type, as a CORBA Type Code string

- a database table name.

Note the contrast with [APM ANSAware 93] in which the same property name can, on different occasions, refer to a number of different types of property.

In a `NamedProperty`, the type of the `PropertyValue` must be consistent with that obtained by the following process:

- use the `Name` in the `NamedProperty` to locate a `NamedPropertySpec` whose `PropertyName` matches `Name`;
- locate a `PropertyTypeSpec` whose `Name` matches the `TypeName` in the `NamedPropertySpec`;
- extract the `TypeCodeString` from the `PropertyTypeSpec`.

The set of properties associated with an offer is accessed using an `OfferID`. It is represented as an integer.

For example:

- offer 23 has a property `BankName`, in sub-profile `Banking`, with value “Barclays”
- there is a `NamedPropertySpec` for property `BankName` in sub-profile `Barclays`, which contains some descriptive text and a `PropertyTypeName` “StringType”.
- there is a `PropertyTypeSpec` with the name `StringType`, which has some descriptive text, a CORBA TypeCode string signifying that the value is a string, and the database table name `STRING_PROPERTIES`.

3.3.4 Monitor Data

Some monitors (the interface to which is defined in 3.4.2) may be able to create or locate parameterised services. A `MonitorConstraint` determines a subset of the possible property value combinations; the offer returned by the monitor will be within that subset.

A relevant example of the use of `MonitorConstraint` information occurs when the monitor is another property repository. If a repository wishes to “advertise” one or more of its services in another repository, the `MonitorConstraint` could include the advertising repository’s `OfferID`, which it would then use to identify the particular offer.

The properties of a monitored offer include a reference to the `Monitor` and a `MonitorConstraint`.

A `MonitorConstraint` is stored as a `PropertyValue` in the “Basic” sub-profile as described in section 3.6.2.

3.3.5 Basic Model

The database of properties associated with interfaces is modelled in terms of `Properties` and `OfferID`s.

This requires that all information stored has an associated `OfferID`. Service interface references, monitor interface references, monitor constraints and interface types are represented as named properties. It is not required that any of these be defined prior to the association of other properties with the same `OfferID`.

A `RepositoryItem` represents an `OfferID` together with (a subset of) its associated `NamedProperties`; a sequence of such items is a `RepositoryInfo`,

which is a subset of the offers held in the repository, with a subset of the properties of each.

3.4 Interface Specifications

Interfaces are specified in CORBA IDL [OMG CORBA 93] in section 4.2.

3.4.1 Property Repository

A property repository is an object with the following interfaces:

- traditional trader (`OldTrader`)
- link manager (`LinkManager`)
- reference manager (`ReferenceManager`)
- type manager (`TypeManager`)
- query manager (`QueryManager`)
- export manager (`ExportManager`)

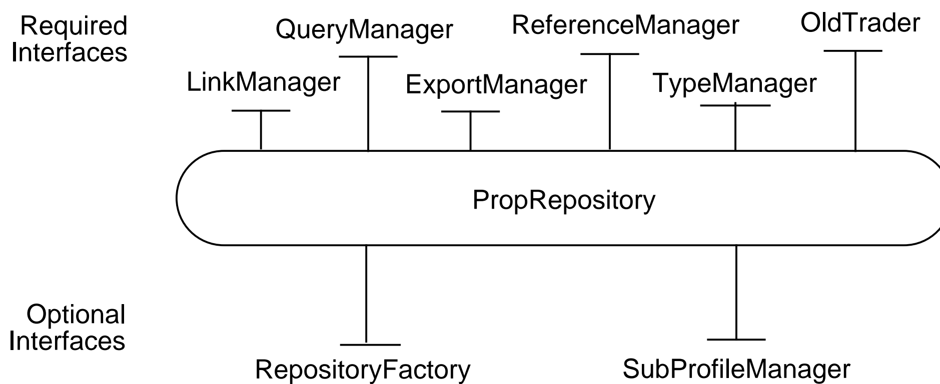
It may support the following additional interfaces:

- repository factory (`RepositoryFactory`)
- sub-profile manager (`SubProfileManager`)

Its type is referred to as `PropRepository`.

The relationship between these interfaces and the corresponding engineering objects may be, but need not be, one-to-one; there are efficiency and management considerations. The interfaces may be distributed over different servers, or all on the same server.

Figure 3.1: Interfaces forming the Property Repository



The interfaces must conform to the specifications in section 4.2, using the data types in section 4.1. A property repository must support at least the “basic” properties sub-profile of chapter 3.6.

3.4.2 Monitor Interface

This interface is not one that is provided by property repositories; it is one which they use when a monitored offer is imported.

It consists of the single operation MonitorGet.

MonitorGet is given a MonitorConstraint. This will typically be constructed by combining the client's selection criteria, a constraint on the OfferId and the constraint originally supplied when the offer was exported. These three components are logically 'and'-ed together.

The Monitor creates or selects an interface according to the composite constraint, and returns an interface reference and those NamedProperties which were requested.

3.4.3 Query Manager Interface

The QueryManager interface has a single operation OpenSession, which creates a QuerySession interface, representing a database connection. This operation may require client authentication. The SQL "connect" primitive provides an example implementation.

The QuerySession interface provides a session of transactions (normally associated with a single client). The invocations within a transaction will have an identical view of the database, regardless of update activity which is taking place as a result of export requests. The operations in the QuerySession interface are DescribeQuery, Query, EndTransaction and CloseSession.

EndTransaction and CloseSession support transactional packaging of query operations; a transaction begins when the interface is first created or after EndTransaction and terminates with EndTransaction or CloseSession.

DescribeQuery takes an SQLString (i.e. a parametric query with no values for the variables referenced within it) and returns two sequences of QueryBaseTypes, indicating the types of the arguments that need to be supplied, and the types of the results of the query. DescribeQuery can be implemented using the "describe" operations in SQL.

Query creates a Selection object representing the information fulfilling the given QuerySpec and passes back a reference to its interface. Query can be implemented by SQL queries (in particular by the SQL "open cursor" primitive).

The information in a Selection object can be that returned by any SQL "select" statement, but the expected use will return a combination of properties associated with offers, possibly including the OfferIDs. The format and order of the tuples depend on the QuerySpec provided to Query, and can be found out by invoking DescribeQuery on the QueryString. The values are returned as a sequence of sequences of QueryBaseValues.

The Selection interface has two operations, FetchSelection and CloseSelection.

FetchSelection returns a number of selected tuples; this number will usually be Units, except for the final successful call. Successive invocations of FetchSelection will return each result tuple exactly once. The SQL primitive "fetch" is an example implementation.

CloseSelection deletes this interface and the remaining information that it provides access to. The SQL primitive "close" is an example implementation.

The CloseSession operation of the QuerySession interface invokes the CloseSelection operation on any Selection interfaces belonging to the session.

As an example, consider the SQL query "SELECT OFFER_ID, PROPERTY_VALUE FROM STRING_PROPERTIES WHERE PROPERTY_NAME = 'BankName';", which returns the BankName property of each offer, together with the Offer IDs.

The DescribeQuery operation will return two sequences of QueryBaseTypes:

- a null sequence because the SQL contains no question-marks indicating that arguments are to be supplied
- a sequence indicating that the types of the results are an integer and a string.

Suppose the Query operation is now invoked, and then the FetchSelection operation of the Selection object is invoked, with Units set to 2. This might return:

```
integer 23, string "Barclays"
integer 42, string "NatWest"
```

A subsequent invocation of FetchSelection, with Units set to 2, might return:

```
integer 69, string "Midland"
integer 95, string "Lloyds"
```

3.4.4 Export Manager Interface

Export creates a new offer and returns its identifier, associating with the offer a set of service properties. These properties will typically include an interface signature, and either a service interface reference or a monitor interface reference with an associated constraint expression.

AddOfferProperties associates further properties with an offer. If properties with the same name as those to be added are already associated with the offer, the new values replace the old.

DeleteOfferProperties removes the properties associated with the given offer that are named in the given PropertyNameProfile.

ListOfferProperties returns a PropertyNameProfile giving the names of all the properties associated with the identified offer.

Withdraw removes the identified offer and all of the named properties associated with it.

3.4.5 Reference Manager Interface

This interface is provided for handling monitored offers, to enable the retrieval of an interface reference which enables binding to the service, together with the associated properties.

LookupReference takes an OfferID (as chosen using Query from a QuerySession interface or Search from the OldTrader interface, for example) and a constraint. If the properties of the offer do not include either a service interface reference, or a monitor interface reference with associated constraint, an error is returned. A check is also made that the identified offer is capable of providing a service satisfying the constraint.

If the properties include an interface reference it is returned directly, otherwise the monitor identified by a monitor interface reference will be invoked, and a composite constraint passed to it as explained in sections 3.3.4

and 3.4.2. The interface reference returned by the monitor will be passed back to the client, together with a set of property values.

3.4.6 Property Repository Factory Interface

Since the traditional trader's notion of "context" has been supplanted by property repositories, this interface offers the function that "create context" operations does in [APM ANSAware 93].

`NewRepository` creates a new property repository conforming to the description given in section 3.4.1.

3.4.7 Link Manager Interface

A `NamedLinkProperty` holds information about a link on which searching policies may act.

A value of type `LinkInfo` contains a name that refers to another `PropRepository` and a set of properties that can be used in a `SearchPolicy`. `SearchPolicies` are referred to by `SearchPolicyNames` and are not explicitly represented in the link manager interface.

When searching or selecting, a `SearchPolicy` may determine whether `Links` are to followed, in what priority and any other details of the search (e.g. depth or breadth of repository tree first).

Objects of the `SearchPolicy` type determine the set and order of linked `PropRepositories` that are to be searched.

When links are considered for following (e.g. in `Search` or `Select` from section 3.4.8) the name of a policy to use is given. This policy is used in conjunction with a link's properties to determine whether the link is followed or not. If it is to be followed, its priority, relative to other applicable links, might be returned as well as the `SearchPolicy` that should be used for subsequent link operations on the linked repository.

Some simple examples of search policies would be:

- 'all' - search all links
- 'local' - search all links with a value for the 'local' property. This might be used to search a set of traders which are located in the same UNIX process, or on the same Ethernet.

`ListLinks` returns the set of links.

`AddLink` creates and names a new link to another property repository. It also associates a set of properties (which can include such things as "local", "friendly", "untrusted" etc.) with the link. If a link with the given name already exists it will be overridden by this new definition.

`RemoveLink` deletes the named link.

`LinkPolicies` interprets the given `SearchPolicyName` in the environment of the current links and returns the sequence of linked property repositories and `SearchPolicyNames` for use in interrogating them. This result is determined both by the `LinkSearchPolicy` provided and any applicable local search policy.

3.4.8 Traditional Trader Retrieval Interface

A `ConstraintExpression` is a boolean expression on properties containing arithmetic, logical and relational operators, and possibly a superlative

function. It is represented as a string, rather than requiring the client to parse it and pass an expression tree.

The database representation of an `InterfaceType` will be determined by the “basic” properties sub-profile (see section 3.6.3); whatever the format, however, it will be capable of being stored as a property.

It is intended that this interface should provide a simpler method of obtaining interface references than that provided by `Query`. It:

- accepts `ConstraintExpressions` expressed in terms of properties, rather than having to use SQL (and therefore having to understand the structure of the database);
- can utilize linked repositories; and,
- does not require the preliminary establishment of a session before it can be used;

but

- `ConstraintExpressions` enable a more limited range of searches than `QuerySpecs`; and,
- offers are not considered if they contain no `InterfaceType` property or no interface reference (e.g. development specification properties provided before an `InterfaceType` property is defined).

One instance of this interface is provided for each `Repository`.

`Select` corresponds to the [ISO ODP Trader 93] operation of the same name and is similar to the [APM ANSAware 93] Lookup operation with Policy Lookup_Random. It returns a service interface reference, and a list of properties, including the interface signature which will conform to the interface type supplied. The properties returned match the `MatchingCriteria` given. If necessary the returned information is extracted from a proxy service as described in `LookupReference` in section 3.4.5.

This information may, depending on a combination of the search policy embedded in the `OldTrader` and the given `SearchPolicy`, be retrieved from linked repositories. If many matches with the `InterfaceType` and `ConstraintExpression` are possible within the searched repositories, information on only one, chosen randomly, is returned.

`Search` corresponds to the [ISO ODP Trader 93] operation of the same name and is similar to the [APM ANSAware 93] Lookup operation with Policy Lookup_All. It performs a similar function to `Select` (including searching linked repositories) except it returns a set of all the matches that could be made and, if it finds any proxy services, it does not invoke the proxy mechanism. The information returned is presented as a sequence of properties as determined by `ServicePropertyNames`. Offers are only returned if their properties include all those in `ServicePropertyNames` and also an interface reference.

3.4.9 Properties Sub-Profile Manager Interface

The sub-profile manager enables its user to interrogate and extend the set of sub-profiles and the set of property types.

An operation on the set of sub-profiles:

- `Profile` lists the names of all the sub-profiles in the repository.

Operations on sub-profiles:

- `SetSubProfile` creates a new sub-profile. The information supplied consists of a name, some descriptive text and specifications of the properties that it incorporates. If a specification for the given `SubProfileName` already exists it is first deleted.
- `GetSubProfile` returns the descriptive text and set of property names associated with a named sub-profile.
- `DeleteSubProfile` deletes the named sub-profile.

Operations on property names within a sub-profile:

- `SetProperty` associates a property name with a property type which must already exist. If another association with the same `PropertyName` exists it is deleted first.
- `GetProperty` retrieves the `NamedPropertySpec` corresponding to the given `PropertyName`.
- `DeleteProperty` deletes the `NamedPropertySpec` corresponding to the given name, if it is not in use as a property of an offer in the repository.

An operation on the set of property types:

- `PropertyTypes` returns a list of all property type names held in the repository.

Operations on property types:

- `SetPropertyType` defines the given property type. If another definition of a type with the same `PropertyTypeName` exists it is deleted first.
- `GetPropertyType` retrieves the `PropertyTypeSpec` corresponding to the given name.
- `DeletePropertyType` deletes the `PropertyTypeSpec` corresponding to the given name if no property specification in the repository refers to it.

3.4.10 Type Manager Interface

Note that the Sub-Profile Manager interface above is concerned with property types, whereas the Type Manager is concerned with interface types.

This interface is used to define a hierarchy of interface type names that can then be used to determine the substitutability of supplied interfaces.

Off-line automatic generation of a type hierarchy is discussed in section 3.6.3. This may eventually mean that the external availability of this interface will be phased out (see section 6.2.1).

A `TypeName` provides a locally unique canonical identification for each different interface signature type. Its representation could be a string `QueryBaseValue` or perhaps a reference to an object in a CORBA interface repository.

`AddType` adds a new type whose interfaces are initially incompatible with all other types.

`AssertConformance` asserts that interfaces with type `Name` are compatible with interfaces of types in the `SuperType` set.

`DeleteType` attempts to remove a type from the `TypeManager`. This will fail if other types are registered as compatible with it or if an offer using this type currently exists.

`ListTypes` returns a list of type names registered in the `TypeManager`.

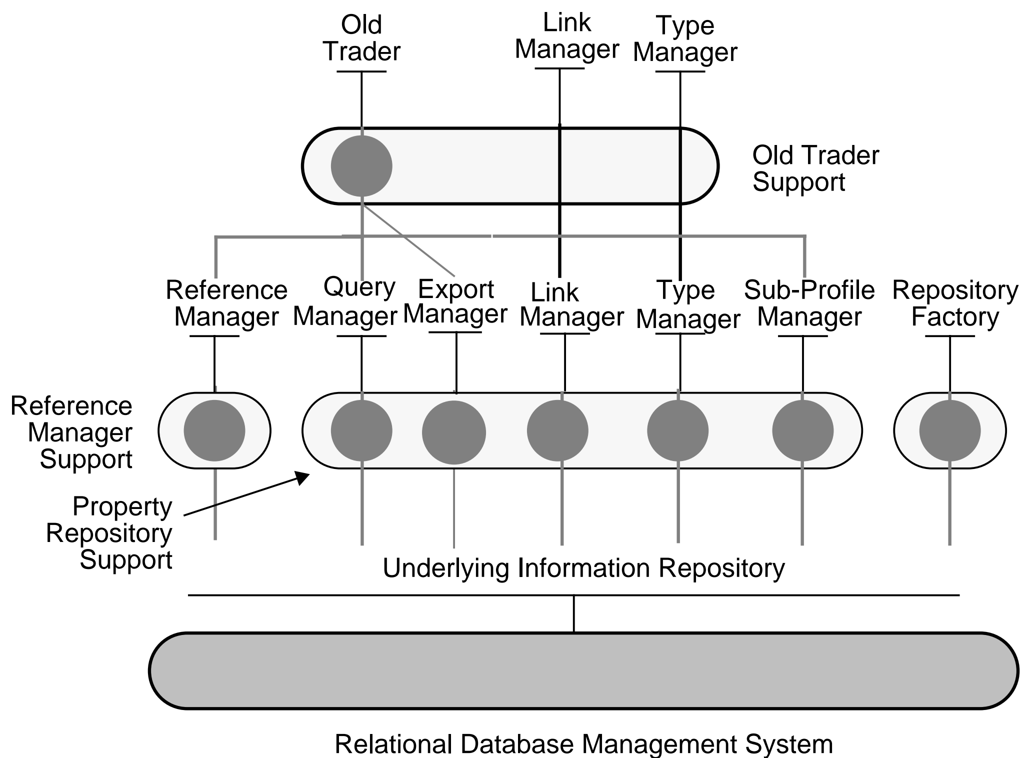
`ListSubTypes` returns a list of types which have been asserted to conform to the given type.

`ListSuperTypes` returns a list of types to which the given type has been asserted to conform.

3.5 Interface Interrelationships

The following diagram provides a decomposition of five main areas of implementation associated with the interfaces specified above.

Figure 3.2: Main areas of interface support mechanism



3.6 The Basic Properties Sub-profile

3.6.1 Purpose

The intention of the definition of a “basic” properties sub-profile is to define the type of properties that need to be supported in order to implement the traditional trader interface (`OldTrader`).

The particular types of information that need to be described are Interface References and Interface Signatures.

A proforma for the information needed to describe a `SubProfile` was given in section 3.4.9.

3.6.2 Interface References

The properties of an offer corresponding to a running, or runnable, service will include one of:

- a service interface reference, or
- a monitor interface reference together with the information (a monitor constraint) that a monitor needs to provide a service interface reference.

All interface references, like other properties, will be held in database tables with the `OfferID` as the primary key. They will be represented as strings, and the repository will therefore not need to know about their internal structure.

3.6.3 Interface Signatures

Initially an interface signature will be represented by a value of the type `TypeName`, as used in the type manager interface, described in section 3.4.10.

Later a signature will become a structured value constructed from the RDBMS base types. See section 8.1.1 for details of this transition.

Interface signatures are required in the repository for two distinct purposes:

- for retrieval
- for conformance testing.

The retrieval of signatures requires that a structured value is stored in the database.

Conformance testing will make use of a pre-computed or asserted type relationship graph. Initially the graph will be based on assertions declared to the type manager (section 3.4.10), but eventually (see section 8.1.1) it will be based on pre-computed relationships using canonical type names.

4 IDL Specifications

4.1 Data Types

4.1.1 Underlying Information System

```
enum QueryBaseKind {...};

union QueryBaseType switch (QueryBaseKind) {
     // something of this form
     ...
};

union QueryBaseValue switch (QueryBaseKind) {
     // something of this form
     ...
};

typedef string SQLstring;

struct QuerySpec {
     SQLstring                     QueryString;
     sequence <QueryBaseValue>     QueryArgs;
};
```

4.1.2 Property Representation

```
typedef string PropertyTypeName;
typedef string Text;

typedef sequence <QueryBaseValue> PropertyValue;

struct PropertyTypeSpec {
     PropertyTypeName    Name;
     Text                Description;
     TypeCodeString      DataType;
     TableNameType       TableName;
};
```

4.1.3 Property naming

```
typedef string SubProfileName;
typedef string PropertySubName;

struct PropertyName {
     SubProfileName      SubProfile;
     PropertySubName     SubName;
};
```

```
struct NamedProperty {
     PropertyName       Name;
     PropertyValue      Value;
};

typedef sequence <PropertyName> PropertyNameProfile;

struct NamedPropertySpec {
     PropertyName       Name;
     Text               Description;
     PropertyTypeName   TypeName;
};

typedef long OfferID;
```

4.1.4 Monitor Data

```
typedef string MonitorConstraint;
```

4.1.5 Basic Model

```
struct RepositoryItem {
     OfferID            Id;
     NamedProperties    Properties;
};

typedef sequence <RepositoryItem> RepositoryInfo;
```

4.2 Interface Specifications

4.2.1 Property Repository

```
struct PropRepository {
     LinkManager        LinkMan;
     QueryManager       QueryMan;
     ExportManager      ExportMan;
     ReferenceManager   RefMan;
     TypeManager        TypeMan;
     OldTrader          OldTrad;
     RepositoryFactory  Factory;
     SubProfileManager  SubProfMan
};
```

4.2.2 Monitor Interface

```
typedef string MonitorConstraint;

interface Monitor {
     void MonitorGet(
          in MonitorConstraint constraint
     ) raises (Failed)
};
```

4.2.3 Query Manager Interface

```
interface QueryManager {
     void OpenSession (
          in string User,
          in string Password,
          out QuerySession Result
     ) raises (Failed);
};

interface QuerySession {
     void DescribeQuery (
          in SQLstring RequirementSpec,
          out sequence <QueryBaseType> Args,
          out sequence <QueryBaseType> Results
     ) raises (Failed);


     void Query (
          in QuerySpec Requirement,
          out Selection Result
     ) raises (Failed);

     void EndTransaction () raises (Failed);

     void CloseSession () raises (Failed);
};

interface Selection {
     void FetchSelection (
          in unsigned long Units,
          out sequence <sequence <QueryBaseValue> > Result
     ) raises (Failed);

     void CloseSelection () raises (Failed);
};
```

4.2.4 Export Manager Interface

```
interface ExportManager {
     void Export (
          in sequence <NamedProperty> Properties,
          out OfferID Result
     ) raises (Failed);

     void AddOfferProperties (
          in OfferID Offer,
          in NamedProperties Properties,
     ) raises (Failed);

     void DeleteOfferProperties (
          in OfferID Offer,
          in PropertyNameProfile Properties
     ) raises (Failed);

     void ListOfferProperties (
          in OfferID Offer,
          out PropertyNameProfile Result
     );
```

```
          void Withdraw (
               in OfferID Offer
          );
     };
```

4.2.5 Reference Manager Interface

```
     interface ReferenceManager {
          void LookupReference(
               in OfferID Offer,
               in string Requirement,
               out NamedProperties ServiceProperties,
               out any Service
          ) raises (Failed);
     };
```

4.2.6 Property Repository Factory Interface

```
     interface RepositoryFactory {
          void NewRepository(
               out PropRepository Result
          ) raises (Failed);
     };
```

4.2.7 Link Manager Interface

```
     typedef string LinkName;
     typedef string LinkPropertyName;
     typedef string LinkPropertyValue;
     typedef string SearchPolicyName;

     struct NamedLinkProperty {
          LinkPropertyName       Name;
          LinkPropertyValue      Value;
     };

     struct LinkInfo {
          LinkName               Name;
          NamedLinkProperties    Properties;
          PropRepository         Repository;
     };

     interface SearchPolicy { // sample definition
          exception DontFollow {};

          void Decide(
               in LinkInfo Link,
               in SearchPolicyName ImportedPolicy,
               out unsigned long SelectPriority,
               out SearchPolicyName ExportedPolicy
          ) raises (DontFollow, Failed);
     };

     interface LinkManager {
          struct LinkStruct {
               PropRepository     Repository;
               SearchPolicyName   Policy;
          };
```

```
        void ListLinks(
             out sequence <LinkInfo> Result
        ) raises (Failed);

        void ShowLink(
             in LinkName Name,
             out LinkInfo Result
        ) raises (Failed);

        void AddLink(
             in LinkName Name,
             in NamedLinkProperties Properties,
             in PropRepository OtherRepository
        ) raises (Failed);

        void RemoveLink(
             in LinkName Name
        ) raises (Failed);

        void LinkPolicies (
             in SearchPolicyName LinkSearchSelectPolicy,
             out LinkPoliciesResultResult
        ) raises (Failed);
   };
```

4.2.8 Traditional Trader Retrieval Interface

```
   typedef string ConstraintExpression;
   typedef string InterfaceType;

   interface OldTrader {
        exception NoMatchingOffers {};

        void Select(
             in InterfaceType ServiceDescription,
             in ConstraintExpression MatchingCriteria,
             in SearchPolicyName LinkSearchPolicy,
             out NamedProperties ServiceProperties,
             out any Service
        ) raises (NoMatchingOffers, Failed);

        void Search(
             in InterfaceType ServiceDescription,
             in ConstraintExpression MatchingCriteria,
             in SearchPolicyName LinkSearchPolicy,
             in PropertyNameProfile ServicePropertyNames,
             out RepositoryInfo Result
        ) raises (NoMatchingOffers, Failed);
   };
```

4.2.9 Properties Sub-Profile Manager Interface

```
   interface SubProfileManager {
        void Profile(
             out sequence <SubProfileName> Result
        ) raises (Failed);
```

```
        void SetSubProfile(
             in ProfileEntry SubProfile
        ) raises (Failed);

        void GetSubProfile(
             in SubProfileName Name,
             out Text DescResult,
             out sequence <PropertyName> PropResult
        ) raises (Failed);

        void DeleteSubProfile(
             in SubProfileName Name
        ) raises (Failed);

        void SetProperty(
             in NamedPropertySpecSpec
        ) raises (Failed);

        void GetProperty(
             in PropertyName Name,
             out NamedPropertySpec Result
        ) raises (Failed);

        void DeleteProperty(
             in PropertyName Name
        ) raises (Failed);

        void SetPropertyType(
             in PropertyTypeSpec Type
        ) raises (Failed);

        void GetPropertyType(
             in PropertyTypeName Name,
             out PropertyTypeSpec Result
        ) raises (Failed);

        void DeletePropertyType(
             in PropertyTypeName Name
        ) raises (Failed);

        void PropertyTypes(
             out sequence <PropertyTypeNames> Result
        ) raises (Failed);
   };
```

4.2.10 Type Manager Interface

```
   typedef string TypeName; // not permanent

   interface TypeManager {
        void AddType(
             in TypeName Name
        ) raises (Failed);

        void AssertConformance(
             in TypeName Name,
             in sequence <TypeName> SuperTypes
        ) raises (Failed);
```

```
        void DeleteType(
             in TypeName Name
        ) raises (Failed);

        void ListTypes(
             out sequence <TypeName> Result
        ) raises (Failed);

        void ListSubTypes(
             in TypeName Name,
             out sequence <TypeName> Subtypes
        ) raises (Failed);

        void ListSuperTypes(
             in TypeName Name,
             out sequence <TypeName> Subtypes
        ) raises (Failed);
   };
```

5 Comparison with Other Traders

5.1 ODP Trader

The Computational Specification of the Trading Function appears in Chapter 8 of [ISO ODP Trader 93].

The repository interfaces have general differences from ODP as follows:

- no client identifier is passed. The issues of authority and trust need to be considered.
- there is no provision in ODP for searching any trader other than that on which the operation is invoked.
- ODP's 'service offer properties' and 'service properties' are combined as 'service properties' for simplicity.

5.1.1 Search

The `Search` operation of the `OldTrader` interface differs from ODP's `Search` operation in the following respects:

- ODP's 'search constraint', which constrains the set of linked traders on which the search is performed, is implemented as a search policy. This simplifies the initial implementation.

5.1.2 Select

The differences between the repository and ODP for `Select` are the same as for `Search`, except:

- there is no 'service property names' parameter in `Select`. There is no good reason for the divergence from ODP; the original reason was that `Search` only returns a single offer, and so there is less need to restrict the amount of information returned than there is in `Select`, which returns multiple offers.

5.1.3 List Offer Details

The repository's version of this operation is in the `ExportManager` interface. Differences are:

- only the property names are returned, not the values; there is no input parameter to specify which properties are required. A later version of the repository should include another operation which takes a list of required property names and returns the values.
- ODP's 'service description' and 'interface identifier' do not appear; the information is returned with the other properties. This difference is justified on the grounds that the repository supports non-runtime use.

5.1.4 Export

The differences are:

- 'service description' and 'service interface id' are optionally included with the properties, as is 'policy controller interface id'. This again is justified for non-runtime use.

5.1.5 Withdraw

No differences except the general ones.

5.1.6 Replace

The repository's version of this is more flexible, being provided as two operations `AddOfferProperties` (which will modify existing values) and `DeleteOfferProperties`. If there are a large number of properties, the ODP operation makes it inconvenient to make small changes.

5.1.7 Add Link

The only significant difference here between ODP and the repository version in the `LinkManager` interface is that the repository allows the caller to choose a name for the new link, whereas in ODP an identifier is returned.

5.1.8 Remove Link

No differences.

5.1.9 Modify Link

This operation is absent from the repository, and should be provided in a later version. The number of link properties is likely to be small, so the repository should follow ODP rather than implementing `AddLinkProperties` and `DeleteLinkProperties` on an analogy with the treatment of the `Replace` operation.

5.1.10 List Link Details

This is called `ShowLink` in the repository; this name should be changed to bring it into line with ODP.

5.2 ANSA Trader

The operations referred to below are those described in [APM 1005.1 93].

5.2.1 Register, RegisterMonitor

The repository's `Export` differs from this operation in that:

- the type and the service reference are included in the properties
- an offer id is returned rather than a reference to an offer object.

The repository does not provide separate interfaces or operations for registration of service offers and registration of monitored offers; they are both performed by using the `Export` operation, but with different properties.

5.2.2 LookUpOne

The repository's `Select` operation corresponds to this. It expects a search policy name in addition to those expected by the ANSA trader.

5.2.3 LookUpAll

The equivalent repository operation is `Search`.

The ANSA trader operation returns service references, whereas the repository operation may return monitor references. The repository's behaviour is more appropriate, because the references are not necessarily being obtained for immediate use.

The repository also allows the caller to specify a set of properties that he is interested in, which is particularly important as the number of service properties multiplies.

5.2.4 Link

This is called `AddLink` in the repository. The only change is the introduction of link properties.

5.2.5 UnLink

`RemoveLink` in the repository is identical.

5.2.6 Contexts

The corresponding repository operation is `ListLinks`. It returns not only the names, but also the properties and the reference at the end of the link.

5.2.7 Select

The effect of this is achieved in the repository by repeated calls of `ShowLink`.

5.2.8 SelectWithPolicy

The effect of this can be achieved in the repository by using repeated calls of `LinkPolicies`, but:

- all the links are returned each time, and you have to find the one of interest
- you have to pass on the appropriate selection policy yourself at each stage.

Perhaps the repository should be enhanced to make this operation easier.

5.2.9 Withdraw

In the ANSA trader, this is a parameterless operation on an `Offer` object; in the repository, the offer is identified by an ID.

5.2.10 Trading context creation

The problem is discussed in [APM 1005.1 93], but no operation is provided. In the repository, the factory enables the creation of new computational traders within the same engineering object.

5.3 ANSAware Trader

5.3.1 Register

The `Export` operation in the repository combines the interface type and interface reference with the properties. There is no equivalent of `NamingContext` (invoke `Export` on a linked repository), nor of `CapsuleRef`.

5.3.2 Lookup

This operation is really two distinct operations, depending on the supplied `Policy`.

`Lookup` with policy `Lookup_Random` corresponds to the repository's `Select`. The repository has no `NamingContext`, but does have a `LinkSearchPolicy`. The result has no context, and the type is included in the properties.

`Lookup` with policy `Lookup_All` corresponds to the repository's `Search`. Again the repository lacks `NamingContext`, but has `LinkSearchPolicy` and also a list of property names of interest. The result has no context, and both the type and the interface reference are included with the properties.

5.3.3 Delete

The corresponding repository operation is `Withdraw`, in which the offer to be deleted is identified by its `OfferID`, rather than by the interface reference. This is a definite improvement, as it allows offers without interface references to be deleted, as well as allowing more selective deletion of those which have one.

5.3.4 AddName

The operation in the ANSAware trader creates both a context and a link to it; to achieve the same effect in the repository one would have to use the factory to create the context, and then use `AddLink` to create the link.

5.3.5 ListNames

The repository equivalent is `ListLinks`, which returns more information.

5.3.6 DelName

This deletes a context and the link to it. In the repository, the link can be deleted by `RemoveLink` but there is currently no way to delete the context.

5.3.7 BindContext

Equivalent to `AddLink`.

5.3.8 UnbindContext

Equivalent to `RemoveLink`.

5.3.9 ProxyExport

This operation and `Register` both correspond to the repository's `Export`. The distinction is made by means of the property list; for a proxy export it would contain a monitor interface reference and a constraint expression, rather than a service interface reference in a normal export.

5.3.10 DeleteProxy

Similar to `Delete`. The two parameters here which identify the offer(s) are replaced by one, the `OfferID`, in the repository.

5.3.11 AddType

The repository operation has the same name, but lacks the `SuperTypes` parameter; `AssertConformance` also has to be called.

5.3.12 MaskType

No equivalent.

5.3.13 UnmaskType

No equivalent.

5.3.14 DelType

Exactly the same.

5.3.15 ListTypes

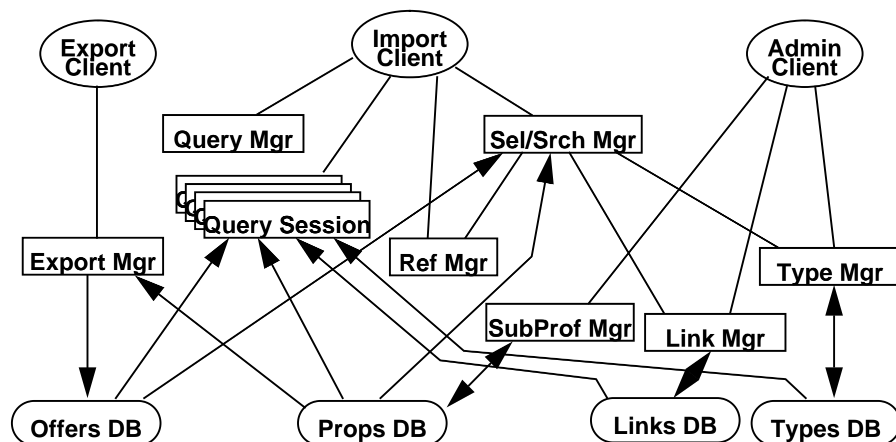
This operation returns a list of types with their supertypes; the repository version just returns the list of types. Callers can then ask for supertypes themselves if they want, rather than being overwhelmed with information which is not required.

6 Design

6.1 Overview

Figure 6.1 gives an overview of the components of the Property Repository.

Figure 6.1: Design Overview



6.2 Design Issues

6.2.1 Behavioural Semantics Specifications

There are two situations in which a trader might need to support specifications of the semantics of services:

1. *development time*

   these specifications are intended to be read by *people*, may be formally or informally expressed, and may be subject to some machine checking (e.g. to check consistency of specification and design) - see section 6.2.3.

2. *runtime*

   these specifications are intended to be read by *machines*, and are formally expressed.

To support development time use, the properties repository only needs to be capable of holding the text of a specification (or a pointer to it, e.g. a URL or a file name) as a property value. Specification checking is a process outside the scope of the repository.

Support for runtime use within the repository is more difficult. A specification language including only alternation, concatenation and recursion constructs is

too complex to guarantee the determination of equivalence in finite time. It might be possible to analyse the specification and extract a set of key properties covering the required range of semantics, and then to provide formal definitions of the properties and their possible values.

Behavioural requirements can be built up out of simple properties, for example:

> *I want an interface with -- printing[postscriptII colour[inkjet]] and (print-queue-auditing or print-job-completion-notification) -- properties.*

The design of properties to represent behavioural semantics is not addressed here, but properties provide the engineering capabilities to deal with a number of possibilities.

6.2.2 Client/Server Symmetry

Traditional traders have been asymmetric, in that the client expresses its requirements on the server, but the server has no opportunity to express its requirements on potential clients.

To make trading more symmetric, the property repository would have to allow a server to state its requirements as a query on client properties.

The prototype could be enhanced in future to allow a server to specify a query string (in SQL or in the ANSAware constraint language, for example), and to allow a client to specify properties which could be matched against the server's query string. Client properties would have to be specified in SQL queries as well as on the traditional trader interface.

6.2.3 Development Process

To support the development process, we need to deal with statements of requirements, and specifications of server behaviour at different levels of abstraction. These will include both very high level abstractions of its behaviour (such as a list of elicited requirements) and very low level abstractions of its behaviour (such as machine-X o/s-Y linkable modules).

In general these specifications do not have a fixed size and their representation within the trader database needs consideration. The possibilities are:

- The ALLBASE SQL data type 'LONG VARBINARY', otherwise known as Binary Large OBjects or BLOBs
- filenames in the database pointing at specifications held outside
- World Wide Web Uniform Resources Locators (URLs) pointing at specifications which can be on any machine on the internet.

The repository will support 'LONG VARBINARY'; the other possibilities are automatically supported, because filenames and URLs are just strings.

Note that the ALLBASE implementation of 'LONG VARBINARY' uses files.

6.2.4 Interface Signatures in the Model

Interface signatures are required for various purposes at various times:

- generation of stubs at compile time
- generating correct dynamic invocations at run time
- checking the safety of an interaction at bind time.

Note that a signature is not necessarily sufficient on its own to allow a suitable interface to be found; types with the same structure do not necessarily perform the same function, and properties should be used to avoid this problem.

The distinctive purpose of signature specifications (signature templates in [ISO ODP Trader 93]) is Type Safety. The repository can be used to provide information to ensure that an interface is obtained which has the right characteristics in terms of performance, dependability, security etc., and type safety is just another one of these, though it is special in that it is mandatory in any offer which is to be used for interaction.

The above discussion shows that type signatures should not be treated specially - they are part of a fundamental sub-profile, and are mandatory on any offers which are to be used for interaction, but in other respects they are just another property.

6.2.5 Extensibility

The properties repository holds specifications of what is available, expressed in terms of different types of property, and statements of what is required, expressed as constraints on the values of those properties. The properties are grouped into sub-profiles, and the specification of a property type will contain:

- a human-readable description
- a specification of the property value type (as a CORBA type code string)
- a database table name, defining how it is stored.

Other information which ought to be included is:

- restrictions on the use of properties to particular interface types
- a specification of combinations of properties, one of which must be present in an offer which is intended for invocation.

There is a table in the database which holds details of the sub-profiles which are currently installed.

Although there are issues still to be resolved, we are assuming that:

- an SQL database is sufficient for holding an adequate range of property types
- additional property types can be added dynamically
- most reasonable queries can be expressed in SQL.

6.2.6 Matching Algorithms - the options

There are two decisions to be made about the matching algorithm:

- what language is it expressed in ?
- at what point in the system is the matching done ?

These decisions are not independent; for example, an algorithm expressed in SQL can only be implemented cheaply if the matching is performed in the repository.

6.2.6.1 Representation of the Matching Algorithm

Use a database query language (for our purposes, SQL):

Advantages:

- cheap to implement, as the SQL is just passed to the database

Disadvantages:

- some queries may not easily be expressed in SQL, e.g. complex queries on substantial amounts of text
- some text properties may be implemented as filenames or URLs, and queries on the file content cannot be handled in SQL
- knowledge of the database structure is required
- typical queries become fairly complex, involving many joins.

Use a tailored algorithmic specification:

Advantages:

- any matching algorithm can potentially be used
- the internal structure of a property need not be visible at the SQL level (unless SQL access is also allowed)

Disadvantages:

- a language for writing matching algorithms needs to be defined
- this language needs to be parsed and processed

Use a wide base set of matching algorithms and a means to compose them:

Advantages:

- any matching algorithms can be chosen as basis of set
- the internal structure of properties need not be visible at the SQL level.

Disadvantages:

- need to find some means to represent the composition of named matching algorithms
- not arbitrarily extensible; available forms of matching algorithm are limited by the size of the base set.

6.2.6.2 *Location of the Matching Algorithm*

In the `QueryManager`:

Advantages:

- only one central representation of the matching algorithm is required

Disadvantages:

- when new property types are added, information may need to be included to allow new matching algorithms.

In `OldTrader`:

Advantages:

- allows the `Repository` to be free of anything but standard query (e.g. SQL) functionality

Disadvantages:

- matching algorithms are not available via the `QueryManager` interface, which reduces its usefulness.

In the trader’s client:

Advantages:

- clients have a great deal of flexibility, and can do any kind of matching they require.

Disadvantages:

- performance - because the matching algorithm is executed further from the database, all relevant offers are sent to the client for checking
- because the client has to do all the matching itself, it cannot take advantage of any centrally supported matching algorithms for given property types.

6.2.7 Matching Algorithms - the decision

The following is supported:

- matching algorithm in SQL, located in the `QueryManager`
- matching algorithm in the ANSAware constraint language, located in the `OldTrader`
- interfaces which allow intelligent repository clients to retrieve a number of offers, and choose one of them according to whatever criteria they like.

The above is both cheap to implement and also flexible.

6.2.8 Design of Interface References

There are two main principles involved here which concern the relationship between the property repository and the design of interface references:

- the storage requirements of the property repository must not constrain the design of interface references
- it must be possible for interface references to be passed between objects, and used for invocation without requiring any explicit involvement of the property repository.

The repository makes the assumption that interface references can be converted to strings and constructed from strings. This must necessarily be that case, because any distributed computing infrastructure must be able to marshal interface references, and the marshalled form can then be converted to a string of hexadecimal digits (though of course there might be a string form which is more convenient for humans to read). We also make an assumption about the maximum length of such a string.

An interface reference is an invocation name (in terms of [APM 1003.1 93]) which can be passed around before binding, during which it may have traversed a number of naming contexts and undergone a number of name translations. Within the local naming context it may have aliases (even if aliases are not routinely generated, they may be difficult to prevent as a consequence of export and subsequent import from federated naming contexts).

6.2.8.1 What is required in an interface reference ?

On the most basic level, an interface reference needs to contain a single engineering address (network endpoint).

Further considerations result in other information that should be contained in interface references, however:

- replicated servers (for dependability) require the possibility of more than one engineering address
- transparent provision of federation, security or Quality of Service (QoS) may require further information (e.g. transfer syntaxes for federation).

Some of this information is mandatory, i.e. required to support every instance of interaction. Other information is optional, and is only needed to support different optional transparencies (e.g. use of transactions).

The general model of an interface reference, therefore, will include a number of values of different property types.

6.2.8.2 *How are interface references obtained ?*

Although an interface reference has been defined above in terms of property types, and although instances *can* be provided by a property repository, the repository need not be the only sources, of interface references, or even an important source, in accordance with the second principle above.

6.2.8.3 *How do you prevent interface references from becoming too large ?*

Section 6.2.8.1 suggests that a considerable amount of information may be required in an interface reference; the second principle above suggests that all the information *must* be contained in the interface reference. The result would then be large interface references, resulting in performance problems.

To deal with this, it will be necessary to resort to indirection. Some of the information may not appear in the interface reference directly; instead it is accessible via an embedded interface reference.

The design of an interface reference will therefore be a compromise between size and performance:

- if there are too many literal copies of info items it will be too large and performance will suffer
- if the most frequently used items of information are represented by interface references, performance will suffer

Another consideration is that of independence. It is desirable that independent authorities (e.g. for remuneration and security) are not required to know about each other; this independence might be compromised if too much information in an interface reference is included literally.

6.2.8.4 *Summary*

- the term “interface reference” is over-used and probably should not be used
- no particular interface reference design will be “right” for all circumstances
- an interface reference should not be used as a key to find other properties - the ANSA naming model implies that it has no canonical form
- an interface reference should contain, literally or by reference, sufficient information to enable it to be used without explicit involvement of the property repository.

6.2.9 Federation

The current repository design assumes that properties sub-profiles with the same name, but in different repositories, are the same (e.g. they have the same purpose, define the same set of properties and represent the properties identically).

The design does not yet address the issues of co-ordination between sub-profile managers to ensure uniform management and seamless integration.

In ODP terms [ISO ODP Trader 93] we are currently addressing *trading syndicates* (one administrator) but not *trading federation* (which requires an arbitrator).

6.2.10 Dynamic Services and Orbix

Some of the operations involved in the Property Repository are really dynamically typed (the current implementation, however, conceals this by using sequences of `QueryBaseValue`). Examples are those in the `QueryManager` and `Export` interfaces. These ought eventually to be implemented in a proper dynamic way in Orbix.

These operations are of the basic form:

```
op: goodargs, badarg1 .. badargX -> anyresult1 .. anyresultY
```

where

- the types associated with the `goodargs` are known at compile time
- the types of `badargs` and `anyresult` are unknown at compile time, but these types depend on the types of `goodargs` in some well-understood way.

The server somehow needs to determine the type of `badargs`, and the client needs to determine the type of `anyresult`. There are two ways of doing this:

1. through a transfer syntax which includes an indication of types (this is currently supported by Orbix), or
2. by derivation, based on knowledge of `goodargs`.

Whichever of these is chosen, the server needs to unmarshal its arguments dynamically, which is not currently possible with Orbix. What is required is a Dynamic Skeleton (or Server) Interface (DSI) to complement the existing Dynamic Invocation Interface (DII) on the client side. The DSI is also required for generic gateways as a result of the ORB interoperability work.

There are indications that proposed approaches to federation will be inefficient in an environment in which the use of dynamically created (use once) interfaces is commonplace.

6.3 Representation in the Database

6.3.1 Property Type Specification

Property types are specified in terms of the basic RDBMS types from which they are composed, and the names of the tables in which the values are held.

A property can be composed of a fixed or variable number of values from RDBMS base types. A CORBA Type Code string [OMG CORBA 93] in the database (with certain restrictions) allows the structure to be determined.

6.3.2 Property Representation Strategies

All properties will be stored in RDBMS tables with the `OfferID` as the primary key.

There are various options for representing properties within database tables:

- tables in which a row contains an `OfferID` and the values of a number of well-known properties, so that there is no need to include a property name field.
- more than one property can be stored in the same database table, if the property values are of the same type, and a 'property name' field can be included.

Which of these is the best option may depend on a number of factors:

- whether the properties are mandatory or optional
- whether particular properties always occur in combination.

Whether optional properties have a default or not is relatively unimportant, because nulls can be used to signify that a property is absent and that there is no default.

See figures 6.2 and 6.3, which show properties in database tables with no property name field.

Figure 6.2: Storage option for mandatory and defaulted properties

OfferID	$C1_{t1}$	$C1_{t2}$	$C1_{t3}$	$C2_{t1}$	$C3_{t1}$	$C3_{t2}$

Single table including mandatory and optional properties

With optional properties, knowing the frequency of occurrence will assist in making the choice of representation.

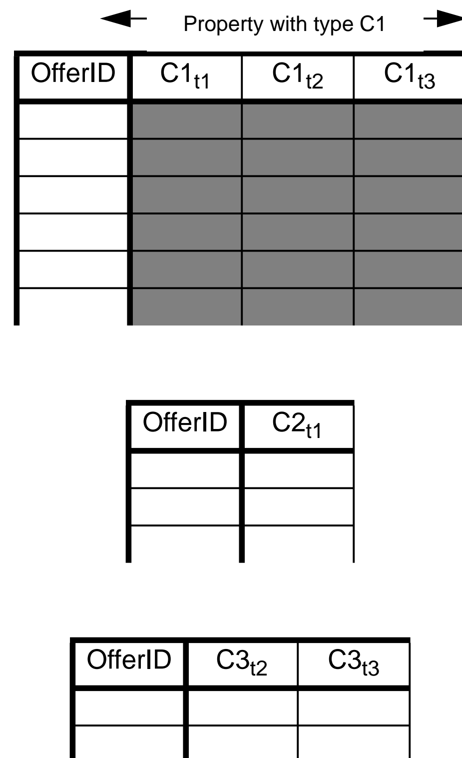
If properties always occur in combination, it makes sense to store them in the same table.

Care is required in the naming of database tables and columns, particularly if two properties have the same name but are in different sub-profiles.

If a property type includes the notion of set, it can be represented as if the set construct were absent but with a tuple for each distinct value in the set. A list or sequence (in which order is significant) can be represented similarly, with the addition of an integer index value.

In the initial implementation of the property repository, all tables have a 'property name' field, which avoids the naming problem at the expense of extra storage in the database. This approach also simplifies the formulating of queries.

Figure 6.3: Storage option for optional properties



Different tables for optional properties

6.4 Options for implementing repositories

There are a number of ways in which a new repository (context) can be implemented within ALLBASE/SQL:

- *Database Environment (DBE)*

  Essentially maps to a server and a directory within that server in which data is stored. Before a program can carry out any SQL operations, it must “connect” to a DBE. A program can be connected to multiple DBEs at the same time, and can designate which one is the “current” one at any time. Queries are executed against the current DBE.

  Using ALLBASE/NET, a DBE can be remote with no change to the program, so long as it uses an alias in the “connect”.

- *Database*

  Essentially a naming, and to some extent an authorization, construct. Any table or view has a concatenated name, including a prefix which is by default the username of the creator.

  The set of all tables and views with the same prefix is a database. By default only the creator of a table or view has access to it. Permission must be specifically granted to others. When a program refers to a table or view and does not provide a prefix, the default is the username under which the program is running.

- *Different sets of tables within a database*

  There would be a naming convention whereby the repository (context) name would be included in the table name.

- *In the same tables, with a 'repository' field to distinguish*

All these options involve some sort of overhead (use of filestore, SQL variability or SQL complexity):

- *DBE*

  This involves significant filestore overhead, as a DBE requires a DBE file (12K or so), a file to contain the tables (some megabytes) and a log file (a megabyte or two). However, there is a compensating flexibility in the use of filestore as the other options would put more and more data into an existing DBE, eventually overflowing.

  This is the only option where the SQL for accessing different repositories is the same.

- *Database*

  There would be more tables in the DBE, as each table needs to be duplicated.

  The SQL requires all table references to be prefixed by a distinguishing name and a fullstop.

- *Tables*

  The database would need more tables.

  The SQL requires different table names to be used (using a naming convention as described above), depending what repository is being searched.

- *Repository Field*

  The database would not need any new tables, but each table would need an extra column.

  The SQL requires the addition of 'where' clauses in many places.

There is not much real difference between the second and third options; the 'database' option has the advantage however that the repository name is separated from the table name by a fullstop rather than simply being concatenated with it.

The fourth option has the disadvantage of SQL complexity compared with the second and third, and no adequate compensating advantage.

The real choice is between the first two options.

The decision has been taken to implement repositories as DBEs:

- the advantage of keeping the SQL constant is unique to this option.
- the overhead seems acceptable; new repositories will be created from time to time, but not at a great rate (about as frequently as new contexts are created in traditional traders).

The question of mapping from various interfaces and repository databases to processes and databases in implementation is already covered elsewhere.

6.5 Use of DBMS and SQL by the various interfaces

6.5.1 Repository Factory

6.5.1.1 NewRepository

This creates a new Database Environment (DBE) as discussed in section 6.4.

6.5.2 Query Manager

6.5.2.1 OpenSession

This opens a new connection to a property repository DBE. using the SQL "connect" operation.

6.5.3 QuerySession

6.5.3.1 DescribeQuery

`DescribeQuery` maps onto the SQL "describe" operations.

6.5.3.2 Query

This is implemented as an SQL "open cursor" operation, followed by the allocation of a `Selection` interface to enable access to the results of the query.

Note that the user may employ the sub-profile manager interface before invoking this operation, to determine the tables required in the `QuerySpec`.

6.5.3.3 EndTransaction

This maps onto "commit work" in SQL (though "rollback work" would work just as well as the `QuerySession` interface provides no update access).

6.5.3.4 CloseSession

This maps onto "release" in SQL, preceded by "commit work" if a transaction is in progress.

6.5.4 Selection

6.5.4.1 FetchSelection

This corresponds to an SQL "fetch cursor" operation.

6.5.4.2 CloseSelection

This corresponds to an SQL "close cursor" operation.

6.5.5 Export Manager

6.5.5.1 Export

The sub-profile manager is consulted to find information about the types of the properties provided. This is used to construct SQL statements to place the property values in the appropriate database tables. A new `OfferID` is allocated, associated with each given property and returned.

6.5.5.2 *AddOfferProperties*

SQL statements are constructed, with the aid of the sub-profile manager, to add the property values to the database, possibly replacing old values. The new values are associated with the given `OfferID`.

6.5.5.3 *DeleteOfferProperties*

The property values are deleted from the database, using SQL statements constructed with the aid of the sub-profile manager.

6.5.5.4 *ListOfferProperties*

The database is queried in order to discover which properties the given offer has. The names of these properties are returned.

6.5.5.5 *Withdraw*

This is achieved by finding out from the sub-profile manager what tables might need rows to be deleted, and then deleting them with an SQL “delete” with a “where” clause.

6.5.6 Reference Manager

6.5.6.1 *LookupReference*

The `PropertyNameProfile` is used to look up a set of `NamedPropertySpecs` (one for each name). Each of these refers to a `PropertyTypeSpec` that identifies the tables which must be queried to retrieve the required properties from the database.

This retrieval should not be performed until the service interface reference has been constructed (in order to give monitors an opportunity to insert or update properties in the database).

6.5.7 Link Manager

6.5.7.1 *ListLinks, AddLink and RemoveLink*

These are implemented by using a link relation in the current DBE. The information contained includes tuples representing the information defined by `LinkInfo`. There is also a ‘link properties’ relation.

6.5.7.2 *LinkPolicies*

There are various options here:

- a fixed number of policies can be explicitly encoded in the implementation of this operation.
- a relation in the database can be used to define the policies; this would contain triples {policy name, match condition, new policy} where the match condition defines how links are chosen for following and the new policy is the one to use when using the link. The matching condition could be represented, for example, as an SQL “select” statement which returns items from the ‘link info’ relation and makes use of the ‘link properties’ relation.
- more complex representations are also possible.

The option that has been chosen for the first implementation is the first one, which has the advantage of simplicity. The fixed policies are “none”, “local” and “all”.

The search policy name given is used to determine which policy to apply and this in turn will determine, in combination with a local search policy, the ordered set of repositories to use and the onwards search policy to apply in each case. The list of repositories is generated by the application of the search policies to the properties of each link.

6.5.8 Type Manager

The operations defined on this interface operate on two database tables:

- one containing the set of type names which have been declared
- another, with two columns, representing the ‘conforms to’ relation.

If the type names are long, it might be worthwhile to add an integer ‘key’ value to the type names table, and use the key instead of the name in the ‘conforms to’ table.

6.5.8.1 AddType

The `TypeName` provided is added in to the ‘type names’ table.

6.5.8.2 AssertConformance

The ‘conforms to’ table is updated to include the asserted conformance relationship, and any that can be deduced from it and the existing relationships.

Note that the insertion of a new type may require the registration of a number of conformance relationships (in both directions) between it and existing types.

6.5.8.3 DeleteType

If no offers have the given `TypeName` as the value of the relevant property (in the basic sub-profile), and if no types conform to it, this operation removes the type from the ‘type names’ table and removes all entries involving it from the ‘conforms to’ table.

6.5.8.4 ListTypes

This simply returns the entire contents of the ‘type names’ table.

6.5.8.5 ListSubTypes

This queries the ‘conforms to’ table to find the subtypes of the given type.

6.5.8.6 ListSuperTypes

This queries the ‘conforms to’ table to find the supertypes of the given type.

6.5.9 Old Trader Retrieval

6.5.9.1 Search

The `ConstraintExpression` provided is written in the ANSAware trader constraint language [APM ANSAware 93], which has a fairly rich syntax involving named properties. To perform the constraint matching, the expression must be analysed and the properties type specification for the named properties may need to be referenced.

Two approaches to performing the matching are considered:

- translate the `ConstraintExpression` into a `QuerySpec` (i.e. an SQL query).
- retrieve a set of `OfferIDs` based on the interface type (of which there are likely to be a relatively small number), then retrieve the property values mentioned in the `ConstraintExpression` for those IDs, then search this information locally to decide which ones qualify.

The second option has been chosen; the first could generate some extremely complicated and inefficient SQL for complex expressions. The chosen option might be subject to some inefficiency too, because a large number of unwanted offers may be retrieved, but at least the implementation is straightforward.

It would also be possible to enhance the second option to find a set of `OfferIDs` which have a conforming interface type and which have all the properties mentioned in the `ConstraintExpression`. This would not be difficult to implement, and could filter out many of the unwanted offers.

A local search policy is used, in conjunction with the supplied `LinkSearchPolicy`, to construct a set of remote `OldTrader` interfaces with which to interact to generate a set of candidate offers, by invoking the `Search` operation with the `SearchPolicyNames` returned by `LinkPolicies`.

Finally, it may be necessary to re-apply the `ConstraintExpression`. For example, it might contain a superlative expression stating that we want the minimum value of the `cost` property; the offers available at this stage have the minimum value of `cost` within the repository from which they came, but it is still necessary to eliminate all those that do not have the minimum value across all repositories.

6.5.9.2 Select

Locally the given `ConstraintExpression` is processed as described above.

Remote repositories are searched, and a set of candidate offers returned, in the same way as for the `Search` operation above. Note that this process still uses the `Search` operation on the remote traders, so that all matching offers are available for selection.

A random selection is made from the available offers.

The same method is used as in `LookupReference` in `ReferenceManager` to obtain an interface reference. If no interface reference results, another selection is made from the set returned and a further attempt is made.

7 Database Tables used by the Repository

7.1 Tables which must be present

7.1.1 Sub-Profile information

The sub-profile information is in a table SUBPROFILE_INFO:

```
SP_NAME                 VARCHAR(32)
SP_DESCRIPTION          VARCHAR(3200)
```

The two fields just contain the name of the sub-profile, and a textual description.

7.1.2 Property Type Information

This is in PROPERTY_TYPE_INFO:

```
NAME                    VARCHAR(32)
TYPE_CODE               VARCHAR(1600)
DESCRIPTION             VARCHAR(1600)
TABLE_NAME              VARCHAR(20)
```

This table contains the property type name, a description of the property type, a CORBA type code string showing the data type, and the name of the database table in which properties of that type are stored.

7.1.3 Property Information

The table PROPERTY_INFO defines the type of a property within a sub-profile:

```
PROPERTY_NAME           VARCHAR(32)
SUBPROFILE_NAME         VARCHAR(32)
TYPE_NAME               VARCHAR(32)
PROPERTY_DESCRIPTION    VARCHAR(3200)
```

The SUBPROFILE_NAME and TYPE_NAME fields are foreign keys, i.e. SUBPROFILE_NAME must match SP_NAME in SUBPROFILE_INFO, and TYPE_NAME must match NAME in PROPERTY_TYPE_INFO.

7.1.4 Offer Information

The user names of exporters are recorded in the OFFER_INFO table:

```
OFFER_ID                INTEGER
EXPORTER                VARCHAR(32)
```

A complete list of the property names which have been supplied for each offer is available in the table OFFER_PROPERTIES:

```
OFFER_ID                INTEGER
SUBPROFILE_NAME         VARCHAR(32)
PROPERTY_NAME           VARCHAR(32)
```

The table `NEXT_NUMBERS` is currently only used for allocating `OfferIDs`, though it could have other uses:

```
NAME                    VARCHAR(32)
VALUE                   INTEGER
```

The `NAME` field will always be set to ‘OfferID’ in the only row in this table.

7.1.5 Type Information

A list of known interface types is in the table `TYPE_NAMES`:

```
NAME                    VARCHAR(32)
```

The conformance relations are represented in the table `TYPE_RELATIONS`:

```
SUB_TYPE                VARCHAR(32)
SUPER_TYPE              VARCHAR(32)
```

where a row will exist in the table iff `SUB_TYPE` conforms to `SUPER_TYPE`.

7.1.6 Link Information

Links between repositories are represented in the table `LINK_INFO`:

```
LINK_NAME               VARCHAR(32)
ITEM_NAME               VARCHAR(32)
ITEM_REF                VARCHAR(3200)
```

A link will be represented by multiple rows in this table, with identical values of `LINK_NAME`. The value of `ITEM_NAME` will indicate which interface the reference in `ITEM_REF` is. The allowable values are: “LinkManager”, “QueryManager”, “ExportManager”, “ReferenceManager”, “TypeManager”, “OldTrader”, “RepositoryFactory” and “SubProfileManager”.

The link properties are held in the table `LINK_PROPERTIES`:

```
LINK_NAME               VARCHAR(32)
PROPERTY_NAME           VARCHAR(32)
PROPERTY_VALUE          VARCHAR(80)
```

7.2 Tables which are normally present

The tables listed in this section will normally be present, but need not be. Their names should not be hard coded, but should be obtained by a query on the table `PRPOPERTY_TYPE_INFO`.

All these tables have the same structure, differing only in the type of the `PROPERTY_VALUE` field:

```
OFFER_ID                INTEGER
SUBPROFILE_NAME         VARCHAR(32)
PROPERTY_NAME           VARCHAR(32)
ITEM_NUMBER             INTEGER
PROPERTY_VALUE          various
```

The list of table names, with the types of their PROPERTY_VALUE fields, is:

```
BINARY_PROPERTIES       VARBINARY(3200)
DATETIME_PROPERTIES     DATETIME
DATE_PROPERTIES         DATE
IFREF_PROPERTIES        VARCHAR(3200)
INTERVAL_PROPERTIES     INTERVAL
NUMERIC_PROPERTIES      FLOAT
STRING_PROPERTIES       VARCHAR(80)
TEXTREF_PROPERTIES      VARCHAR(128)
TEXT_PROPERTIES         VARCHAR(3200)
TIME_PROPERTIES         TIME
TYPE_PROPERTIES         VARCHAR(32)
```

7.3 Additional tables containing property values

Any additional tables which contain property values must be set up in a similar way to those listed above, i.e. the first four fields must be OFFER_ID, SUBPROFILE_NAME, PROPERTY_NAME and ITEM_NUMBER, and the following field(s) must have type(s) compatible with the type code in PROPERTY_TYPE_INFO. There will be more than one such field if the type code indicates a structure.

If there is only one extra field, some of the current code assumes that its name is PROPERTY_VALUE; if there is more than one, it is wise to avoid the use of that name.

8 Planned Evolution of Implementation

Not all of the features described are planned for immediate implementation. Priority has been assigned to different features for implementation in the short, medium and long term. As a result, some features will be implemented cheaply in the short or medium term, and re-implemented in a fuller way in the longer term.

This chapter documents the implementation priorities.

8.1 Properties Sub-profile Support

8.1.1 Support for signatures

Comparison between interface types (signatures) is required in the implementation of the Traditional Trader retrieval interface (3.4.8) and this functionality must also be available to the user of the Query Manager interface (3.4.3).

The ANSAware trader in [APM ANSAware 93] uses a name to represent an interface type and uses the type manager interface (3.4.10) to define the “satisfies” relation between different interface types explicitly. Compared with automatic checking of signatures, this method has advantages:

- it does not assume that it makes sense to substitute one interface type for another just because the infrastructure would support it (i.e. semantic knowledge about services is used)
- it is much easier to implement.

On the other hand, it suffers from a serious disadvantage:

- its correct working is dependent on the user of the type manager; if he asserts an incorrect substituability relationship, interaction faults may occur.

In [APM 1139.0.3 94], information to support the two elements (interaction and semantics) of the check is placed in separate categories.

Signatures will then be expected to be used to support only the former type of check. Inclusion of separate information for these two checks is a long term aim, although the semantic substitutability element of the checks will e excluded in the medium term.

Support for signatures will be implemented as follows:

- *short term*:
  - — substitutability checking is done by name, based on assertions
  - — no signature information available
- *medium term*:
  - — substitutability checking still done by name, based on assertions

  - signature information available
- *long term*:
  - substitutability checking done by comparison of signatures
  - type hierarchy generated automatically, as described in section 3.6.3
  - type manager becomes an internal interface - it will only be used by the automatic type hierarchy generation process.

8.2 Repository Support

8.2.1 Support for the property repository factory

Initially, new property repositories can only be created manually, by interacting directly with the underlying database. In the medium term, an SQL interface will be provided to support this function (equivalent to the creation of a new context in the ANSAware trader).

8.3 Traditional Trader Support

8.3.1 Constraint Expression Support

Initially the range of constraint expressions that can be supported will be restricted to those supported by the ANSAware trader's constraint language.

In the long term support may be extended to enable queries on large binary values and to enable more complex queries on structured values.

8.3.2 Search Policy Support

Initially search policies will be supported that use link properties to implement at least the following:

- search no linked repositories;
- search local linked repositories; and,
- search all linked repositories.

No additional manager defined or user defined policies will be supported.

In the medium term, policies allowing the depth of the search to be controlled will be provided.

In the long term, a policy representation will be defined enabling managers to create a range of special purpose policies.

8.4 Summary of Timetable

8.4.1 Short term

Signatures: type names and assertions used for substitutability checking, structural information not available.

Property Repository Factory: manual creation of repositories.

Constraints: single value properties, and 'is this string in this list', only

Search policies: at least ‘search none’, ‘search local’ and ‘search all’.

8.4.2 Medium term

Signatures: type names and assertions used for substitutability checking, full structural information available.

Property Repository Factory: interface available for dynamic creation of repositories.

Constraints: as short term.

Search policies: support ‘depth of search’ policies.

8.4.3 Long term

Signatures: automatic generation of type hierarchy from structural information, and use of this hierarchy for substituability checking. Type manager interface becomes ‘internal use only’.

Property Repository Factory: as medium term.

Constraints: support for expressions involving large binary values and structured values.

Search policies: dynamic creation of arbitrary search policies.

8.5 Future Evolution

Currently the property repository supports only properties associated with services. Extensions should be provided to allow information to be held associated with the offers themselves and with clients.

9 Experience of initial implementation

The initial implementation of the Property Repository using Orbix and Allbase, on the whole, proved to be easier than expected.

Particular positive points are:

- the use of dynamic SQL[1] often made implementation easier, even when static SQL could have been used. A particular example of this is when the old trader is looking for properties; the code to extract string and numeric properties out of tables would have contained much unnecessary duplication if a static approach was used.
- putting a table name as a field in another table was a useful approach in achieving dynamic extensibility.
- the `QueryBaseValue` and `QueryBaseType` types, which are effectively a 'database any', made life easier in many places.

On the negative side:

- type relationships are not easy to represent in a relational database in a way that makes SQL querying straightforward. It is necessary to build up a table which contains an entry for each type relationship deduced, not merely for each relationship explicitly asserted. The size of this table could become quite large as the number of types in the system increases.

1. the SQL statements are not available to the SQL pre-processor, and therefore the types of the results of queries are not known at pre-processing time either.

References

[APM 1003.1 93]

Van der Linden, R J, *The ANSA Naming Model*. APM Ltd., Cambridge, UK, February 1993

[APM 1005.1 93]

Deschrevel J-P, Herbert A J, *The ANSA Model for Trading and Federation*. APM Ltd., Cambridge, UK, February 1993

[APM 1137.1 94]

Wai F, Otway D J, Howarth N J, Herbert A J, *A Performance Framework*, APM Ltd., Cambridge, UK, March 1994

[APM 1139.0.3 94]

Hoffner Y, Girling C G, *Boundaries and Domains*, APM Ltd., Cambridge, UK, April 1994.

[APM 1140.0.6 94]

Hoffner Y, *A Model of Co-operative Trading*, APM Ltd., Cambridge, UK, April 1994.

[APM.1229.0.8 94]

Cameron E J, van der Linden R J, *Information Model for Federation*, APM Ltd., Cambridge, UK, August 1994.

[APM ANSAware 93]

ANSAware Version 4.1 Manual Set, APM Ltd., Cambridge, UK, February 1993.

[ISO ODP Trader 93]

ISO/IEC and ITU-T, *Information technology – Open Distributed Processing – ODP Trading Function*, ISO/IEC JTC1/SC21/WG7 N880, November 1993

[OMG CORBA 93]

Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., SunSoft Inc., *The Common Object Request Broker: Architecture and Specification*, **Revision 1.2**, December 1993

IDL Index

A

AddLink 29
AddOfferProperties 27

C

CloseSelection.................................... 27
CloseSession 27
ConstraintCriteria 29

D

DeleteOfferProperties 27
DeleteProperty 30
DeletePropertyType 30
DeleteSubProfile................................ 30
DescribeQuery 27

E

EndTransaction 27
Export.. 27
ExportManager 27

F

FetchSelection 27

G

GetProperty 30
GetPropertyType 30
GetSubProfile 30

I

InterfaceOfferID 26
InterfaceType 29

L

LinkInfo 28
LinkManager 28
LinkName 28
LinkPolicies 29
LinkPropertyName 28
LinkPropertyValue 28
LinkStruct 28
ListLinks 29
ListOfferProperties 27
LookupReference 28

M

Monitor 26
MonitorConstraint 26

N

NamedLinkProperty 28
NamedProperty 26
NamedPropertySpec 26
NewRepository 28
NoMatchingOffers 29

O

OldTrader 29

P

Profile 29
Property 25
PropertyName 25
PropertyNameProfile 26
PropertySubName 25
PropertyTypeName 25

PropertyTypes 30
PropertyTypeSpec 25
PropRepository 26

Q

Query 27
QueryBaseKind 25
QueryBaseType 25
QueryBaseValue 25
QuerySpec 25

R

ReferenceManager 28
RemoveLink 29
Repository 27
RepositoryFactory 28
RepositoryInfo 26
RepositoryItem 26
RepositorySession 27

S

Search 29
SearchPolicy 28
SearchPolicyName 28
Select 29
SetProperty 30
SetPropertyType 30
SetSubProfile 30
ShowLink 29
SQLstring 25
SubProfileManager 29
SubProfileName 25

T

Text 25
TypeName 30

W

Withdraw.. 28