ESPRIT Project No. 25 338

Work package F

Service Interaction

Service Interaction Design

ID:	DF3Design	Date:	18/05/98
Author(s):	SAB, NPT	Status:	
Reviewer(s):	APM	Distribution:	

Change History

Document Code	Change Description	Author	Date
DF3.1	Deliverable	Steve Battle	24/02/98
DF3.2	Revision	Steve Battle	1/04/98
DF3.3	Revisions in line with User Access	Steve Battle	21/04/98
DF3.4	Revisions for review	SAB, NPT	18/05/98

1 Introduction

The aim of this work-package is to develop a framework that will give agents access to Internet based services. The aim is not to develop specific services but to create the necessary tools for building the pilot applications.

The technical requirements for the service interaction work-package distinguish between basic service interaction, and so-called meta-level service interaction. While the former covers the mechanics of making services available within a distributed system, a meta-description of a service describes what that service is, and how it may be used; it's behavioural semantics.

The following table shows the timetable for the development of service interaction components, which fall under the broad categories of *service shell* and *service directory*. The service shell refers to the interface to the profile object and associated tools; the service directory refers to trading services.

Service shell

deliverable	***description***
DF5.1	Service profile object for basic service interaction with support for agent missions.
DF5.2	Service profile tools based on the service signature.
DF5.3	Service profile tools based on the service contract.

Service directory

deliverable	***description***
DF6.1	Basic trading.
DF6.2	Dynamic trading.
DF6.3	Federated trading.

2 Requirements and Analysis

2.1 User requirements

The user requirements of Service Interaction may be described as a set of use-cases. The area of user activity we have modelled here is analogous to web-browsing. The area of agent development is not appropriate to model in this form. The user requirements are set against a background of service providers “advertising” their services in some generally accessible place.

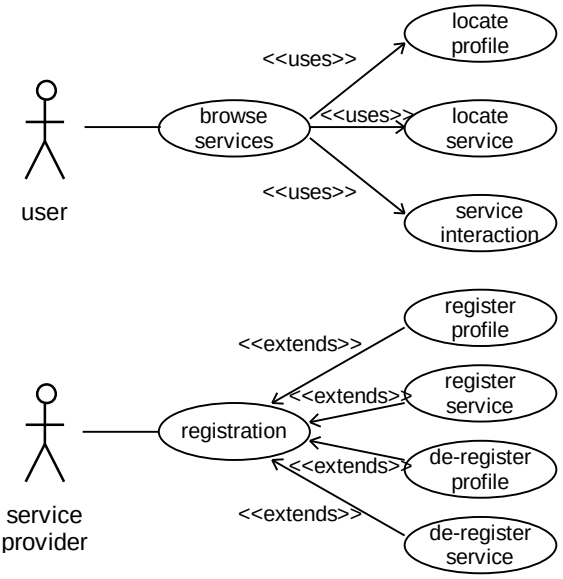


Figure 1 use case diagram for user and service provider

In many ways the action of the agent is invisible to the user, which is why agents do not appear in the use-cases above. An agent can be regarded as a means of getting a job done, in this case the task of browsing and using services.

The extends relationship is used between registration and the separate cases of registering and de-registering profiles and services because none of these is essential to the main use case. A service may be registered without a profile where it is anticipated that only hard-coded clients will ever access that service. A service profile may be registered without a service implementation if that profile represents an abstract interface, or vertical service domain.

2.2 Domain Object Model

The domain object model allows the objects in the problem domain to be clearly stated in context with other domain objects.

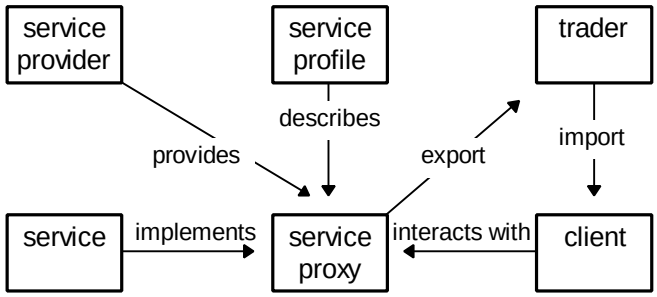


Figure 2 - Domain Object Model

2.2.1 Service

A service is an object that clients can interact with, either to gain information, or to effect some action such as an on-line purchase. A service may be implemented as a Java RMI service, or as a CORBA service or even as a CGI back-end, in which case the service proxy is required as a wrapper.

2.2.2 Service provider

The service provider is the agency responsible for the creation of the service proxy and the service profile that describes it. They may or may not be responsible for the service(s) underlying the proxy. To make a new service (proxy) or service profile available, these objects must be exported to a trader.

2.2.3 Service proxy

The service proxy object allows service providers to straddle the boundary between FollowMe places and the real world beyond. The interfaces in this world may include legacy CGI interfaces or CORBA services, which the service proxy is used as a wrapper. While the services themselves may be fixed at a particular address, the service proxy has the advantage of mobility, allowing the possibility of load-balancing on the service side (see the service deployment work-package). The service proxy may define a value-added service incorporating information from commonly available services (see Information Providers in Pilot Application 1, WP I).

2.2.4 Service profile

The service profile object is a meta-description of the service interface (corresponding to the service `shell' defined in the technical annex). It includes descriptions of the service signatures; the operations defined within one or more interfaces. Accompanying this is the behavioural model supported by the service. The service profile is also the means by which service providers can deliver agent missions to their clients.

2.2.5 Trader

The Trader is the object which allows service proxies to make their presence known to clients that wish to use them and as such is the first point of contact for any client wishing to locate a service. See chapter 3 for a full description of Trading.

2.2.6 Client

Within FollowMe, the client of a service is normally an agent which is an autonomous program issued by a user (see the Autonomous Agents work-package).

2.3 Basic service interaction

This analysis of requirements takes the use-cases identified in document 'DF2: Service Interaction Requirements', and distributes their functionality across a number of objects, including those found in the domain object model. The object notation is from [1].

2.3.1 Service registration

A service is made available to agents by registering it with a trader located within an agent place (a subclass of MOW place). The trader must be supplied with a reference to the service interface. In addition, the service may be associated with a number of named properties by which suitable services can be located. These properties generally include, the service type (the name of the interface to which the service conforms), and the service name (names should not be relied upon to uniquely determine a service, as spoofing cannot be ruled out). A service that is not currently available should be unregistered from the trader.

The relationships between a service, its interface, and the trader are illustrated in the object diagram below. When the service provider starts up a service, it is the responsibility of that service to register itself with the local trader.

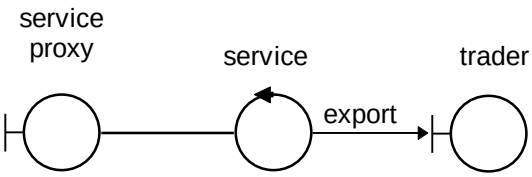


Figure 3

2.3.2 Service profile registration

The service profile is a separate object that can be used to describe both the service signature (the typed interface), and the service contract (the behavioural semantics). The service profile is also the means by which service providers can make their own agents available to users; the service profile may contain any number of agent profiles which operate against that service. Both the service and its nested agent profiles include textual descriptions of their function which is intended to allow users to find and select the appropriate service/agent combination.

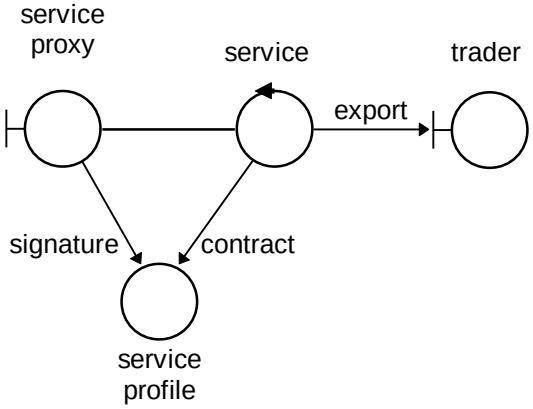


Figure 4

The existence of a service profile does not guarantee the existence of a service implementation. A service profile may remain registered with the trader when a service that implements it is unregistered. Any number of implementations may share the same service profile. A service profile need only be unregistered when it is no longer valid.

In addition to the service profile, the diagram below introduces a repository object. This is required if the profile object needs to be stored in some publicly available place independent of a given service provider. A reference to the service profile is exported to the trader with properties summarising the types of interface it defines.

2.3.3 Locate a service profile

The service profile can assist the client in selecting the appropriate service. For clients with a user interface, the textual descriptions indicate the function of the service (and of any agents defined within the service profile). It will be common for the service profile to be requested prior to the service itself.

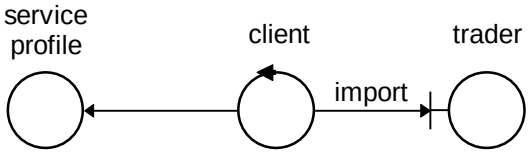


Figure 5

2.3.4 Locate a service

Where an agent does not contain a direct reference to a service it must have some way of locating the appropriate service. These location facilities are available at the agent place and are described more fully in document DD3: Autonomous Agents Design.

The result of naming or trading is a reference to a particular service interface and the corresponding service profile. Interfaces are strongly typed, it is up to the agent to have with it the code necessary for interacting with interfaces of this type.

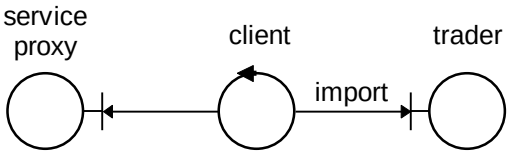
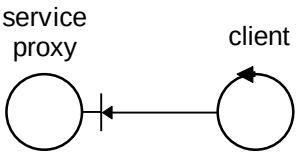


Figure 6

2.3.5 Service interaction

Given a valid service interface, the client may access the service. Long term interactions with a service may be required where the results of a query are not immediately available. In these cases the client passes a self-reference to the service which may call-back later on.



2.4 Sequence diagrams

In this section we amplify on the object diagrams of the previous section to describe in more detail how these objects interact with each other. Many of these interaction diagrams must assume a context established by elements from the Autonomous Agents work-package, particularly for logging-in and agent/user interaction. All such assumptions are indicated. The set of messages presented in the interaction diagrams above, are organised by object in appendix 1.

2.4.1 Registration

One possible sequence of interactions is shown in the diagram below. A service is shown registering both itself and its profile which is deposited in the repository. Note that the repository is itself a service that must be requested from the trader. It also shows the service de-registering itself (but not its profile) at some later time. The diagram does not exclude the possibility of registering just the service or service profile by itself.

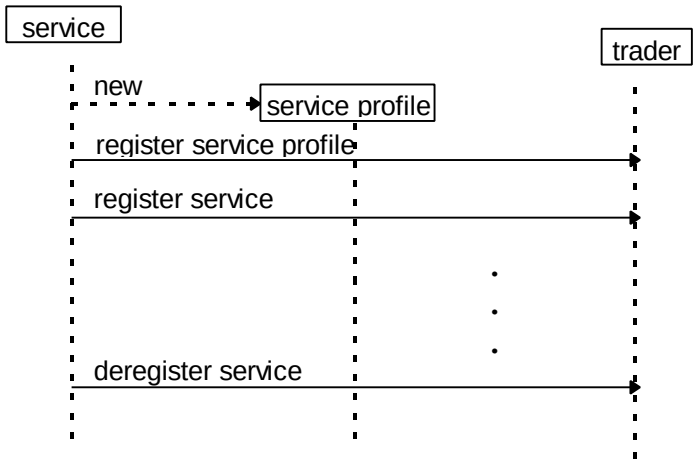


Figure 7 Service / service profile registration

2.4.2 Browse services

The simplest kind of agent/service interaction is a client-driven model. Messages from multiple clients are arbitrarily interleaved. In most cases the client will be a task agent.

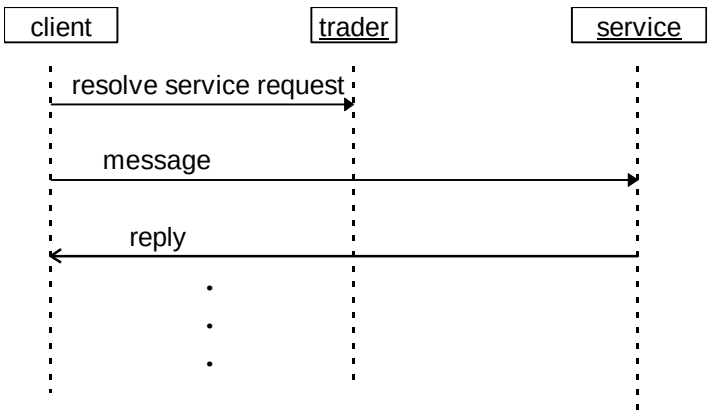


Figure 8 locate and interact with service

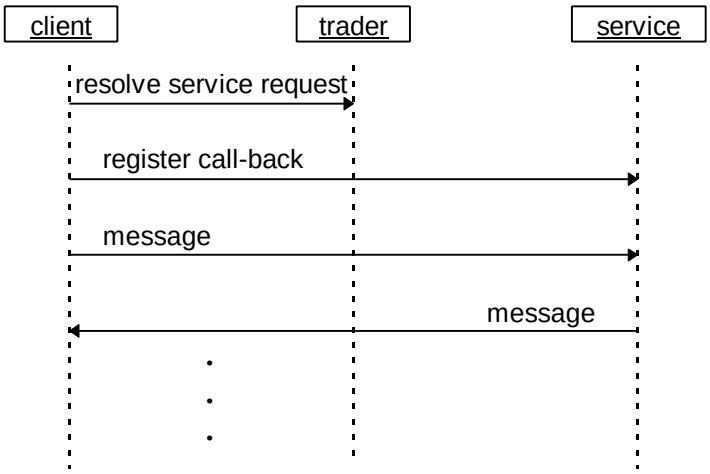


Figure 9 service interaction with call-backs

A third pattern for service interaction involves the dynamic creation of a service *handler*. The primary service assumes the role of a *factory*. Each client has sole access to its own service handler.

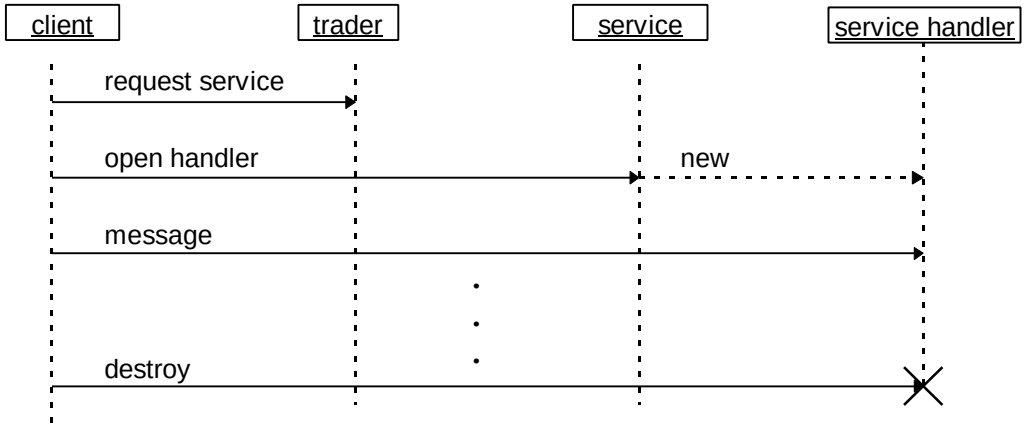


Figure 10 service interaction using a service handler

This pattern is adopted in interactions with user-access. The user access service creates a new connection through which the agent communicates with the user. In this case the client may also register a call-back with the service handler again allowing two-way communication.

2.5 Service proxies

All of the services described are themselves objects within the FollowMe framework. Because most services will actually operate outside this framework, the service interface is actually a *proxy* for that service. In this way, the service provider is able to create wrappers for a variety of services including those defined as Java IDL, CORBA IDL, mobile objects, and perhaps even CGI. Service proxies will represent the service in the FollowMe agent framework and since proxies will be based on the Mobile Object class provided by the MOW, they will be capable of movement. This allows service proxies to migrate between Agent Places and allow sophisticated proxy deployment schemes to be implemented, such as that being provided by the Service Deployment WP.

3 Service Trading

A Trader as defined in [2] has already been identified as a pivotal component of distributed systems that need to provide a dynamic service based infrastructure. It is a facilitator in distributed system in that it allows services to advertise their offer of service which can subsequently be selected by clients and the service be used. There are two notable Trader references available, [2] and [3] both of which influence our design presented later, however a full implementation of either of these standards is beyond the scope of this WP

The basic operation of a Trader can be described as follows and with reference to Figure 11 - Basic Trading:

A service wishing to advertise its presence exports an interface reference which includes details of the services type and location (1). A client wishing to use a service of a particular type contacts the Trader and requests a service offer of the required type (2). The Trader consults the offers it has and depending on the number available, may return no offer (if are there are none), a single offer or list of offers (3). The client then picks an offer and interacts with the service (4).

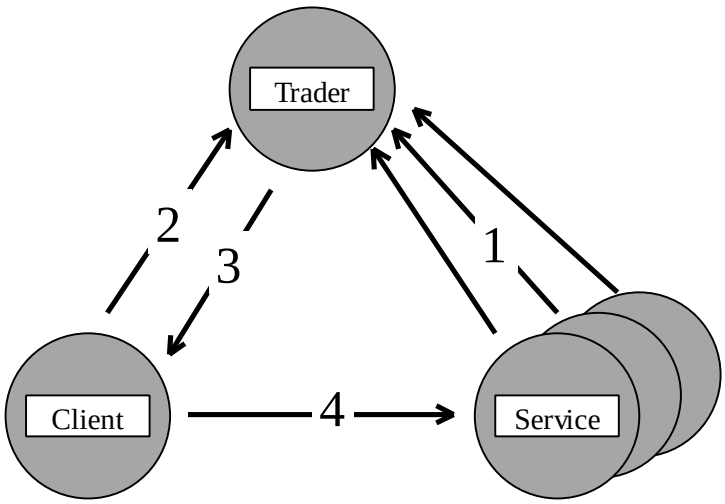


Figure 11 - Basic Trading

A commonly used example describing the use of a Trader is with print services. There may be several printers available to a client and each of these printers export their offer of service to the Trader together with attributes describing the capabilities of the service. For example, a printer will have a page-per-minute rating and may be capable of colour printing or not. A client wishing to print a colour document will request the Trader to return a reference to a printer (the service type) that can print in colour. This reference is known as a service offer. The Trader searches its offer list to see if it knows of any colour printers and returns the offer to the client which can then use the printer.

3.1 Service Types

Service types are general descriptions of a kind of service. They normally consist of the following:

- a type name (in the printer example, this would be “printer”)
- a definition of the services interface (normally defined in an Interface Definition Language)
- zero or more attributes of the service (see 3.13)

3.2 Service Offers

Service offers describe a specific service that is available based on the service type. A service offer consists of:

- a reference to the service (where it is located)
- zero of more attributes of the service being provided

3.3 Properties

The exporting of a service offer with page-per-minute ratings allows the service to be described more accurately than the service type name declaration allows. These service attributes are known as properties and are specified in the form of name-value pairs. For example, a print service may the specify the name-value pair of (“PPM”, 10) where “PPM” is the name of the property and 10 is the value.

The permissible properties are defined by the service type and may have one of the following characteristics:

1 **Static and Optional**
The exporting of the property by the service is optional however if it is exported by the service, the value of the property does not change for the duration of the offer.

2 **Static and Mandatory**
The service must export the property and the value of the property does not change for the duration of the offer.

3 **Dynamic and Optional**
The exporting of the property by the service is optional however if it is exported by the service, the value of the property may change over the duration of the offer.

4 **Dynamic and Mandatory**
The service must export the property and the value of the property may change over the duration of the offer.

In order for clients to issue requests with requirements on properties, a language is defined. This language allows clients to issue property constraints of the form “PPM > 5”.

3.4 Dynamic properties

When a given service property may changes over time, it may be better to represent it as a dynamic property. In this case, no value is associated with the property name, but the trader is given permission by the service to query it about that property when a request for that service is being considered. Dynamic properties may be used for communicating load-balancing data to the trader so that the least loaded service for example, may be selected as the most appropriate service to use.

3.5 Interworking Traders

If the trader cannot resolve the requested service, the request may be propagated to other traders in order to resolve the request. Traders may be organised hierarchically, such that unresolved requests can be passed upwards. A trader may have any number of children from which it imports their database of object references and properties. Each trader can be thought of as covering a specific domain, and the parent trader covers the domain of its children. A trader is able to resolve queries for all services within its domain.

4 Trader Design

In Service Interaction we will provide a Trader that allows not only services to found, but agents to be located also. The Trader will be a core component of an Agent Place and one Trader will be present in every Agent Place. When a service proxy or agent is created or arrives in an Agent Place, it will inform the Trader which will record the interface in it's own local store and in an Information Space. When the Trader receives a request, it will firstly check to see if has a local reference to an interface that is suitable. If it cannot find one, then it will consult a repository which holds details of all interfaces available and if a suitable interface is found, it will be returned to the requester. To provide the repository, we intend to use the Information Space WP and implement a Trader Information Space (IIS) to which all Traders will have access. When a proxy arrives at an Agent Place, it will export its interface to the Trader together will any properties. The Trader will record the interface locally and also in the IIS. When a proxy leaves an Agent Place because of migration or destruction, it should withdraw its interface from the Trader.

4.1 Types and Service Profiles

In order for interface requests to be based on the type of the proxy, a type space must be defined by the application developers. However, since we wish to incorporate the notion of a service profile, we will extend the idea of type to produce a service profile space. The space may be viewed as a file-system, for example, a stock application may use several information feeds represented by service proxies. The proxies may represent feeds from different stock exchanges including the Nikkei, FTSE and Dow-Jones. The profile space for this example would be:

```
/Service/StockFeeds/Nikkei
/Service/StockFeeds/FTSE
/Service/StockFeeds/DowJones
```

Additionally, agents may be represented in the profile using the name of the agent to make it unique. For example:

```
/Agent/PersonalAssistant/Scully
/Agent/PersonalAssistant/Mulder
/Agent/TaskAgent/StockAgent_1
/Agent/TaskAgent/StockAgent_2
```

In order to allow new service profiles to be added at run-time, the Trader will provide methods to allow the registering and retrieval of service profiles will be used. The default Trader will maintain the service profile space. When the Trader is started, it will look in its IS for service profile objects which define the current service profile space and use structure defined therein. If no service profiles are found, the Trader will start with an empty profile space which will need to be added to reflect proxy service types present in the application.

4.2 Implementation overview

We will provide an interface for a Trader (see Appendix C) and a default implementation, `TraderImpl`. The implementation will have the following features:

- Allow proxies and agents to export their interfaces and properties
- Allow proxies and agents to withdraw their interface
- Allow interface requests with property constraints to be resolved
- Allow service profiles to be added and withdrawn

Property constraints may be used to give tighter control over the characteristics of the service implementation required. For example, a property constraint string for the printer example could be of the form "`PPM > 5 and colour`" which if evaluated to boolean true, will mean that the corresponding offer is a match. The operators ==, !=, >, <, >= and <= may be used in constraint expressions. Expressions may be combined using `and`, `or` and `not`.

When resolving a request, the Trader has default behaviour when searching its offer space, that is to return the first matching offer it finds. There are many different search policies that may be employed by the Trader for which we will provide a structure. Since there is no great need to provide customised searching, we will only specify the following search policies which may be extended if the need arises:

`return_random_offer`: if more than one offer matches, return a random matching offer
`return_first_offer`: default behaviour; returns first matching offer
`return_all_offers`: returns an enumeration of all matching offers

4.3 Classes

4.3.1 Class `TraderImpl`

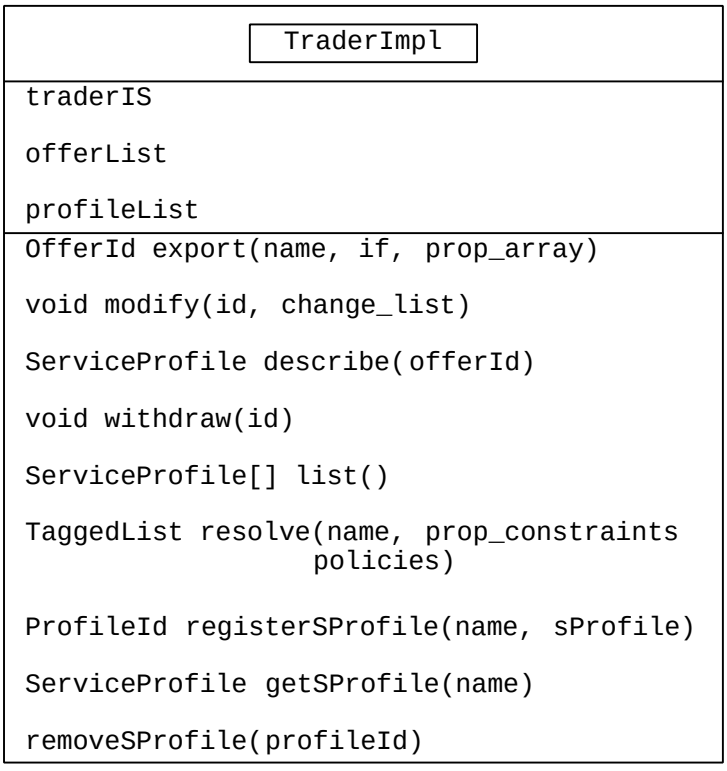


Figure 12

The class diagram for the default Trader interface implementation, `TraderImpl` class is shown in Figure 12. It provides the functionality made available to clients by way of the interface.

`traderIS`: a handle to the Trader Information Space

`offerList`: the list of offers held by the Trader

`profileList`: list of Service Profiles held by the Trader

`TaggedList resolve(name, prop_constraints, policies)`
`TaggedList`: an enumeration of `Tagged` objects
`name`: the name of the interface type which will be a `String`
`prop_constraints`: a `String` that specifies the property constraints.
`policies`: an enumeration of `Policy` objects that overide the default policies of the Trader

Exceptions
`BadNameException`: the `name` passed is badly formed
`NameNotFoundException`: there are offers of the `name` given
`BadPropertyException`: a property name is badly formed
`DuplicatePropertyException`: the property has been specified more than once
`MandatoryMissingException`: a mandatory property was not exported
`BadPolicyException`: a policy name is not recognised
`BadConstraintException`: the `prop_constraints` string is badly formed

`OfferId export(name, if, propValList)`
`OfferId`: a handle that can be used to modify or remove the offer at a latter date
`name`: the name of the service type
`if`: the interface of the service which will be of type `Tagged` (see DB2, Mobile Object Workbench)
`propValList`: an instance of the class `PropertyValueList` which contains zero or more `PropertyValue`(s) specifying the characteristics of the service.

Exceptions
`BadNameException`: the `name` passed is badly formed
`BadPropertyException`: a property name is badly formed
`DuplicatePropertyException`: the property has been specified more than once
`MandatoryMissingException`: a mandatory property was not exported

`OfferInfo describe(offerId)`
`offerId`: an `OfferId` used to identify a target offer
`OfferInfo`: a structure used to describe an offer

Exceptions
`BadOfferIdException`: the `offerId` passed is not valid

`void modify(id, change_list)`
`id`: an `OfferId` that is used to identify the target offer
`change_list`: an instance of the class `PropertyValueList` which contains `PropertyValue`(s) specifying the new values of dynamic properties

Exceptions
`BadOfferIdException`: the `offerId` passed is not valid
`BadChangeException`: the change list badly formed
`ModifyStaticException`: an attempt to change a static property

`void withdraw(id)`
`id`: the `OfferId` returned from a previous export operation

Exceptions
`BadOfferIdException`: the `offerId` passed is not valid

`ServiceProfile[] list()`
Returns a list of all the `ServiceProfile`(s) the Trader holds for its service proxy offers.

`ProfileId registerSProfile(sProfile)`
`sProfile`: a Service Profile object.
`ProfileId`: an identifier that can be used to remove the service profile

`void removeSProfile(profileId)`
`profileId`: service profile handle to be removed

Exceptions
`BadProfileIdException`: the `profileId` passed is not valid

4.3.2 Class `Property`

The `Property` class will define the permissible name-value pairs where the name is a `String` and the `type` is one the base Java types; `boolean`, `char`, `byte`, `int`, `short`, `float`, `double` or `String`. Lastly, the property mode is one of the four characteristics mentioned earlier.

The class Property is illustrated in Figure 13.

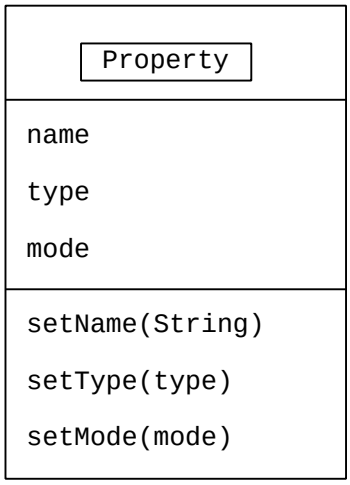


Figure 13

`type` may be set to one of the following:

```
TYPE_BOOL
TYPE_CHAR
TYPE_BYTE
TYPE_INT
TYPE_SHORT
TYPE_FLOAT
TYPE_DOUBLE
TYPE_STRING
```

`mode` may be one of the following:

```
MODE_STATIC_OPTIONAL
MODE_STATIC_MANDATORY
MODE_DYNAMIC_OPTIONAL
MODE_DYNAMIC_MANDATORY
```

4.3.3 Class `PropertyValue`

The `PropertyValue` class is illustrated in Figure 14.

4.3.4 Class `PropertyValueList`

The `PropertyValueList` class is a container for `PropertyValue`(s) and is shown in Figure 15.

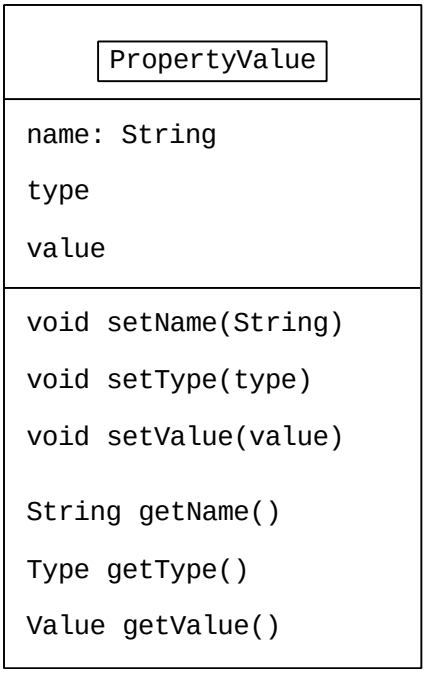


Figure 14

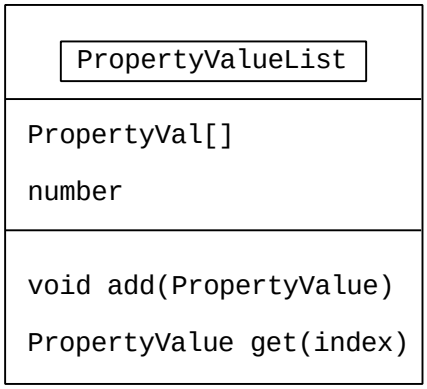


Figure 15

4.3.5 Class `OfferInfo`

The class in Figure 16 describes an offer held by the Trader.

4.3.6 Class TaggedList

The class TaggedList is illustrated in Figure 16

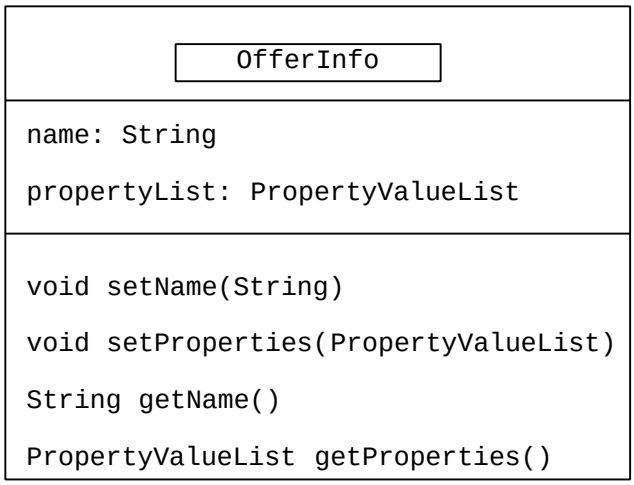


Figure 16

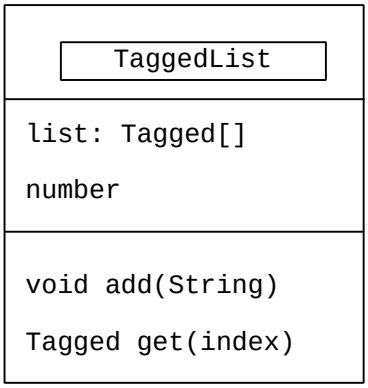


Figure 17

4.3.7 Class `Policy`

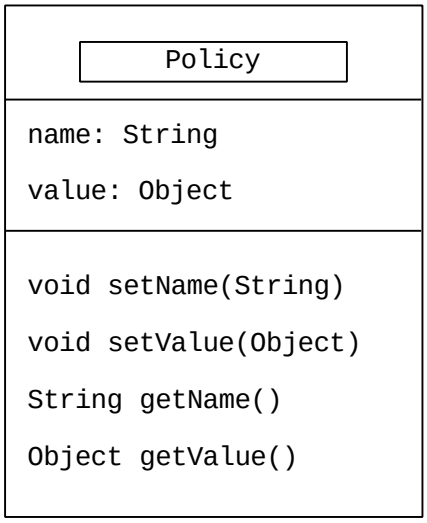


Figure 18

As mentioned earlier, we only define a small number of policies which the trader will support however we include in the `Policy` class a value object which may be used with a policy however in our first implementation we will not use it.

4.4 Behaviour Required from a Service Proxy

In order for services to have a presence in application built using the agent framework, they must implement a Service Proxy for the service. This will extend `Agent` which allows it to reside in an Agent Place and have mobility potential. The application developers will have to provide the a number of behaviours to allow agents to interact with the service and also implement the means by which the proxy provides the service:

- Construct an appropriate Java interface definition of the proxy
- Register the proxy with the Trader
- Optionally export a Service Profile to the Trader
- Remove its offer from the Trader when it is destroyed

The following sections will describe the general operations that may be performed. The operations will be described with reference to the printer example and will include Java code fragments where appropriate. The exception handling has been omitted for clarity.

4.4.1 Constructing a Java interface

The application developer must start by defining the Java interface of the service proxy. Carrying on with the printer example, this will be of the form:

```
interface Printer {

   public abstract void print(PsStream stream);
}
```

This interface must be implemented by the developer and an implementation is outlined here:

```
public class PrinterImpl extends followme.Agent implements Printer {

   public void print(PsStream stream) {
      // Code to print the document
}
```

The following operations must also be implemented.

4.4.2 Defining a Service Profile

A service profile must be defined for every service proxy to enable the Trader to check that details supplied in an export are valid.

```
Property[] props = new Property[2]; // create an array of 2 Property objects
props[0].setName("Name");
props[0].setType(Property.TYPE_STRING);
props[0].setMode(Property.MODE_STATIC_MANDATORY);
props[1].setName("PPM");
props[1].setType(Property.TYPE_INT);
props[1].setMode(Property.MODE_STATIC_OPTIONAL);
```

The above code fragment shows two Property objects being created. Then the first property object is initialised to be a mandatory string being the printers name. The second property is defined to be an optional `int` type.

The 'property' object constructed above is an ideal candidate for representation by the content description tools defined in the Personal Profiles work-package.

4.4.3 Finding the Trader

In order to register, modify or remove an offer, the Trader interface must first be acquired. A Trader is present in every Agent Place which has an operation `Trader getTrader()` (see DD3). This returns an interface to the Trader which may then be used. The following code is used to retrieve the Trader interface:

```
Trader trader = AgentPlace.getTrader();
```

4.4.4 Registering the Service Proxy

To advertise itself, the Service Proxy must export its offer of service to the Trader together with any properties. In the printer example, two properties were specified; Name and PPM. This illustrated by the following code fragments:

```
PropertyValueList propValList = new PropertyValueList();
```

```
PropertyValue prop;
prop = new PropertyValue(“Name”, “laser1”);
propValList.add(prop);
prop = new PropertyValue(“PPM”, 10);
propValList.add(prop);
```

This code creates and initialises a sequence of property values that can be exported by the service proxy.

```
OfferId offerId = trader.export(“Printer”, interface, propValList);
```

This registers the interface with the Trader under the offer name of Printer with the associated properties.

4.4.5 Modifying Properties

Only properties that are “dynamic” can be modified and then only by the proxy that lodged them. When a service is selected and it has dynamic properties, the Trader will call the `update()` method on the proxy interface. The proxy implementation must implement the `update()` method with code that modifies the appropriate properties. An example update method is shown below:

```
public void update() {

   // First get new values for dynamic properties
   int newValue = fromSomewhere();

   PropertyValueList propValList = new PropertyValueList();
   PropertyValue prop;
   prop = new PropertyValue(“PPM”, newValue);
   propValList.add(prop);
   trader.modify(offerId, propValList);
}
```

4.4.6 Describing an Offer

Once an offer has been exported and an OfferId obtained, the proxy can get a description of the offer as illustrated below:

```
OfferInfo offerInfo = trader.describe(offerId);
```

4.4.7 Removing an Offer

When an offer can no longer be met, the offer should be removed. This is illustrated below:

```
trader.remove(offerId);
```

When a proxy crashes and leaves its offer with the Trader, the offer has become stale. If many stale offers accumulate, there may be a need to periodically clean-up the stale offers. This will be achieved by the Trader periodically calling a cleaning routine that periodically “pings” the proxy. The `ping()` method is implemented by the `Agent` class and immediately returns if the proxy is still active or times-out if the proxy has crashed.

4.4.8 Listing ServiceProfiles

The Trader implements a `list()` method which returns an enumeration of `ServiceProfiles` corresponding to service offers held. This allows agents to discover new services as illustrated below:

```
ServiceProfileList list = trader.list();
```

4.5 Agent interaction with Trader

Agents interact with the Trader through the agent scripting language. Agents, like service proxies, must be well behaved. They must register with the local Agent Place Trader when they arrive at an Agent Place, and remove their presence from the Trader when they leave an Agent Place. Since the implementation of this is described in detail elsewhere, the reader is directed to the Autonomous Agents DD3 document.

5 Service Profiles

The technical annexe distinguishes between basic service interaction and so-called meta-level service interaction. While the former covers the mechanics of making services available within a distributed system, a meta-description of a service describes how that service may be used; its behavioural semantics. Service profiles will be designed to work in close conjunction with the task agent language of the Autonomous Agents work-package, and in particular with the requirement for supporting goal-directed activity.

This work constitutes a major research goal of this work package, and consequently its design status at this point is relatively open. This section aims to give an overview of this work, and how it fits in with the autonomous agent framework.

5.1 The service signature

The service signature defines the types of operations and their parameters defined within one or more interfaces. This information may be captured succinctly in the CORBA Interface Definition Language (IDL) defined in [4]. To capture this information within the service profile we aim to use the content description language tools defined in the personal profiles work-package. This mapping provides an object model, of the content of an IDL document. Getting this object model right is critical for programs that are to read and process the service profile. The use of a standard content description language, rather than CORBA IDL itself, say, allows us to extend the core set of primitives to include descriptions of the behavioural semantics underlying the interface; the service contract.

5.2 The service contract

The service contract is motivated by the need to check that agents work correctly with respect to given services, and to work out what kinds of agent-building tools would be supported by this technology.

An explicitly represented behavioural semantics provides greater scope for checking that both the service and the client conform to the same interface. More significantly within the FollowMe project, it provides the raw material for the creation of new clients against a service interface. One of the requirements of the Autonomous Agents work-package is to look at the feasibility of providing *goal directed* agent behaviour. These goals can be understood as a high level specification about what the interaction between service and agent is intended to achieve. In other words, agent goals and service behaviour are expressed in the same language. To show that an agent's behaviour is directed towards a certain goal, we need to show that there is a sequence of steps leading from some initial state to the final goal state. Many of these steps represent basic operations against services, so it is important that we have a rich language in which to express their effects, in terms of the changes they make to the entities relevant to that service. Defining this language is the job of the service interaction work-package.

While the behavioural semantics of a given service is effected by the specific implementation of that service, most object models avoid the issue of trying to specify this semantics in a language independent, declarative manner The

declarative specification of an object's behaviour is often limited to the declaration of a signature; the operation names and their argument and result types.

The design of a service contract must take into account the tightly paired interaction between client and service. If a client is to be run against a service we will need to ensure that the client and service are complementary in the operations they are each willing to *send* and *receive* at any moment. Each send or receive operation must be qualified by the conditions necessary for its successful invocation, and the condition it establishes on termination. These conditions will be expressed in a simple predicate form.

5.2.1 Example

The service contract defines how a service can be used. It does this by defining the *state transitions* that may occur at each interface. The required behaviour can be described as a state transition graph, where each node represents a state the service may be in at any time, and the arcs drawn between them represent operations on, or by, the service. The state transition diagram in Figure 19 represents a service with three discrete states. Following the first operation, Op1, we may perform any number of Op2's, followed finally by a single Op3. Such a behaviour might be typical of a program opening a file, writing successive lines to it, and then closing it.

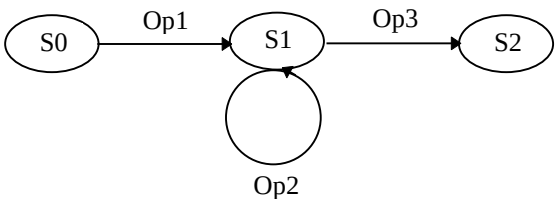


Figure 19 - State Transition Diagram for a simple service

In general, the actual number of states a service may assume is not finite, so we cannot use a simple finite state diagram like the above. Instead each 'state' can be regarded as a logical condition covering a (possibly infinite) set of individual states. The service contract is described in terms of the transitions in the diagram; each clause of the contract is comprised of the triple (start state, operation, end state). This definition draws on Floyd-Hoare logic.

- The start state, or precondition: The conditions under which the operation may occur.
- The end state, or post-condition: The conditions established by the operation within the context of the precondition.
- The operation itself: expressed in the task agent language.

The diagram above would be captured by the three clauses, (S0,Op1,S1), (S1,Op2,S1), (S1,Op3,S2).

As a more concrete example of this, let us devise a set of interfaces based on a hypothetical user access service. This example is not intended as a specification of the actual user access interface. The idea is that we have a pair of related interfaces:- a UserAccess interface can be used to **open** a service handler to which we can **write** output strings. When the client has finished with the handler it can be **closed**. The service signature might look something like the following.

```
<idl>
  <module>
    This service supports user/agent interaction
    <interface name="UserAccess">
      <operation name="open" type="Handler"/>
    </interface>

    <interface name="Handler">
      <operation name="write">
        <parameter type="string">
      </operation>
      <operation name="close"/>
```

```
    </interface>
  </module>
</idl>
```

Introducing variables, *UA* and *H*, representing user access and the handler, we can represent the service contract in a state diagram. The variable, S, represents an arbitrary string.

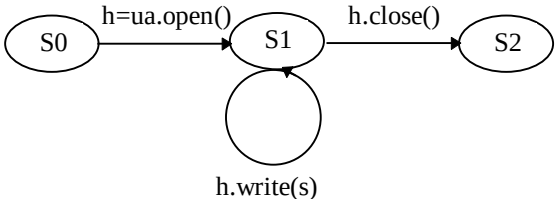


Figure 20 - Service Contract State Diagram

Now we need to consider the conditions represented by each state. In state S0 we are assuming that *UA* is an instance of UserAccess. We could represent this with the unary predicate, UserAccess(UA). The post-condition established by this operation is that *h* is an instance of Handler, "Handler(H)". When we finally close the handler, we must assert the negation of this last assertion, "not Handler(H)". These contract clauses can now be defined using the *contract* tag, <receive>, which defines an operation which if executed by the service in a state where the precondition holds, will establish the post-condition when it terminates.

The contact clauses are declared within the appropriate interface objects. The fact of writing a string to the handler is recorded in the assertion, **written(S)**.

```
<idl>
  <module desc=" This service supports user/agent interaction">
    <interface name="UserAccess">
      <operation name="open" type="Handler">
        <receive>
          <pre>UserAccess(UA)</pre>
          <post>Handler(this)</post>
        </receive>
      </operation>
    </interface>

    <interface name="Handler">
      <operation name="write">
        <parameter type="string" name="S">
        <receive>
          <pre>String(S)</pre>
          <post>written(S)</post>
        </receive>
      </operation>
      <operation name="close">
        <receive>
          <pre>Handler(H)</pre>
          <post>not Handler(H)</post>
        </receive>
      </operation>
    </interface>
  </module>
</idl>
```

The contract described above considers only messages initiated by the client. How can we incorporate events communicated back to the client?

5.2.2 Example 2 - A call-back contract

Events are commonly implemented using the call-back mechanism. The service provides an operation to add a *listener* of a given type. If the client implements the listener interface it can register itself as a listener. When an event occurs, the service notifies all the relevant listeners. The first step is to define a listener interface, which is included as a third interface in the existing module.

```
<interface name=”Listener”>
  <operation name=”event”>
    <parameter type=”string”>
  </operation>
</interface>
```

To implement this interface, the client must support all of the operations defined in the listener interface; in this case the ‘event’ operation. The event can be invoked on a listener providing it has been registered. We add this contract to the interface above. Because this is an operation invoked on the client rather than the service, the contract is denoted by a <send> tag.

```
<interface name=”Listener”>
  <operation name=”event”>
    <parameter type=”string”>
    <send>
      <pre>registeredListener(L) and String(E)</pre>
    </send>
  </operation>
</interface>
```

Finally we need to add an operation to the Handler interface which registers a new listener, and in the contract, asserts registeredListener(L), where L is the client.

```
<module>
  . . .
  <interface name=”Handler”>
    . . .
    <operation name=”AddListener”>
      <parameter type=”Listener” name=”L”>
      <receive>
        <pre>Listener(L)</pre>
        <post>registeredListener(L)</post>
      </receive>
    </operation>
  </interface>
</module>
```

The main function of contract data is to assist users and developers in the construction of agents. Within the scope of the FollowMe project we anticipate the development of tools to verify the design of an agent against a contract. Combining the service contract with other generic script fragments we can start to see how an agent might be assembled to accomplish a particular goal. The task agent script shown below is adorned with assertions that match the preconditions of the operations that follow them, and the preconditions of those that precede them. For the sake of simplicity, exceptions have been ignored.

```
<script>
  s = “Hello world” ; <assert>string(S)</assert>
  ua = trader.request(. . .) ; <assert>UserAccess(ua)</assert>
  h = ua.open() ; <assert>Handler(h)</assert>
  h.write(s) ; <assert>written(s)</assert>
  h.close() ; <assert>not Handler(h)</assert>
</script>
```

This kind of verification would allow the user to check that given, *string(s),* for s= “hello world”, the above script satisfies the goal, *written(s),* drawn from the service domain. Agents can therefore be analysed and selected on the basis of *what* they do rather than *how* they do it.

6 References

[1] Jacobsen, Christerson, Jonsson, Overgaard, *Object-Oriented Software Engineering*.

[2] ODP Trading Function, ISO/IEC JTC1/SC21

[3] OMG Trader Specification ,<http://www.omg.org/>

[4] OMG *CORBA/IIOP 2.2 Specification* (IDL Syntax and Semantics), <http://www.omg.org/cichpter.htm>

Appendix A Service profile DTD

```
<!ELEMENT MODULE        (TYPEDEF | CONST | EXCEPTION | INTERFACE | MODULE)*>

<!ATTLIST MODULE        DESC    CDATA #IMPLIED>

<!ELEMENT INTERFACE     (INHERITS | TYPEDEF | CONST | EXCEPTION | ATTRIBUTE | OPERATION |
                        CONTRACT | INVARIANT)*>

<!ATTLIST INTERFACE             DESC    CDATA #IMPLIED>

<!ELEMENT TYPEDEF       (SEQUENCE | STRUCT | UNION | ENUM)*>

<!ATTLIST TYPEDEF       TYPE    CDATA #IMPLIED>
<!ATTLIST TYPEDEF       SIZE    CDATA #IMPLIED>
<!ATTLIST TYPEDEF       NAME    CDATA #IMPLIED>

<!ELEMENT SEQUENCE      (SEQUENCE)*>

<!ATTLIST SEQUENCE      TYPE    CDATA #IMPLIED>
<!ATTLIST SEQUENCE      SIZE    CDATA #IMPLIED>

<!ELEMENT STRUCT        (MEMBER)*>

<!ATTLIST MEMBER        NAME    CDATA #IMPLIED>

<!ELEMENT UNION         (CASE | DEFAULT)*>

<!ATTLIST UNION         NAME    CDATA #IMPLIED>
<!ATTLIST UNION         SWITCH CDATA #IMPLIED>

<!ELEMENT CASE          EMPTY>

<!ATTLIST CASE          VALUE CDATA #IMPLIED>
<!ATTLIST CASE          TYPE    CDATA #IMPLIED>
<!ATTLIST CASE          SIZE    CDATA #IMPLIED>
<!ATTLIST CASE          NAME    CDATA #IMPLIED>

<!ELEMENT DEFAULT       EMPTY>

<!ATTLIST DEFAULT       TYPE    CDATA #IMPLIED>
<!ATTLIST DEFAULT       SIZE    CDATA #IMPLIED>
<!ATTLIST DEFAULT       NAME    CDATA #IMPLIED>
```

```
<!ELEMENT ENUM              (ENUMERATOR)*>

<!ATTLIST ENUM              NAME   CDATA #IMPLIED>

<!ELEMENT ENUMERATOR        EMPTY>

<!ATTLIST ENUMERATOR        VALUE CDATA #IMPLIED>

<!ELEMENT CONST             EMPTY>

<!ATTLIST CONST             TYPE    CDATA #IMPLIED>
<!ATTLIST CONST             NAME   CDATA #IMPLIED>
<!ATTLIST CONST             VALUE CDATA #IMPLIED>

<!ELEMENT EXCEPTION         (MEMBER)*>

<!ATTLIST EXCEPTION                 NAME   CDATA #IMPLIED>

<!ELEMENT INHERITS          EMPTY>

<!ATTLIST INHERITS          NAME   CDATA #IMPLIED>

<!ELEMENT ATTRIBUTE         EMPTY>

<!ATTLIST ATTRIBUTE         READONLY (TRUE | FALSE) “FALSE”>
<!ATTLIST ATTRIBUTE         TYPE    CDATA #IMPLIED>
<!ATTLIST ATTRIBUTE         NAME   CDATA #IMPLIED>

<!ELEMENT OPERATION         (PARAMETER | RAISES | CONTEXT)*>

<!ATTLIST OPERATION         ONEWAY (TRUE | FALSE) “FALSE”>
<!ATTLIST OPERATION         TYPE    CDATA #IMPLIED>
<!ATTLIST OPERATION         NAME   CDATA #IMPLIED>

<!ELEMENT PARAMETER         EMPTY>

<!ATTLIST PARAMETER         MODE   (IN | OUT | INOUT) “INOUT”>
<!ATTLIST PARAMETER         TYPE    CDATA #IMPLIED>
<!ATTLIST PARAMETER         NAME   CDATA #IMPLIED>

<!ELEMENT RAISES            EMPTY>

<!ATTLIST RAISES            NAME   CDATA #IMPLIED>

<!ELEMENT CONTEXT           EMPTY>

<!ATTLIST CONTEXT           VALUE CDATA #IMPLIED>


<!ELEMENT CONTRACT          (#PCDATA)>

<!ATTLIST CONTRACT PRE      CDATA #IMPLIED>
<!ATTLIST CONTRACT POST     CDATA #IMPLIED>

<!ELEMENT INVARIANT         (#PCDATA)>
```

A service profile may also contain missions as defined in the Autonomous Agents work-package.

Appendix B Service Signature & Contract Tags

The following tag definitions describe the proposed XML/IDL mapping.

The <*module*> tag

Function: defines a naming scope
Attributes: name desc
Contains: <typedef> <const> <exception> <interface> <module>

The <*interface*> tag

Function: signature of the operations defined in an interface
Attributes: name desc
Contains: <inherits> <typedef> <const> <exception> <attribute> <operation> <invariant>

The <*typedef*> tag

Function: user-defined type
Attributes: type size name
Contains: <sequence> <struct> <union> <enum>

The type is one of the IDL primitive types; boolean, char, octet, string, int and float, or a scoped name.

The <*sequence*> tag

Function: defines bounded and unbounded sequences
Attributes: type size
Contains: <sequence>

The <*struct*> tag

Function: defines a structure
Attributes: name
Contains: <member>

The <*member*> tag

Function: defines a structure or exception member

Attributes: type size name
Contains: <sequence> <struct> <union> <enum>

The <*union*> tag

Function: defines a union structure
Attributes: name switch
Contains: <case> <default>

The <*case*> tag

Function: a specific case within a union structure
Attributes: value type size name
Contains: nothing

The <*default*> tag

Function: the default case within a union structure
Attributes: type size name
Contains: nothing

The <*enum*> tag

Function: defines an enumeration
Attributes: name
Contains: <enumerator>

The <*enumerator*> tag

Function: defines an enumerator within an enumeration
Attributes: value
Contains: nothing

The <*const*> tag

Function: declare a constant
Attributes: type name value
Contains: nothing

The <*exception*> tag

Function: user-defined exception
Attributes: name
Contains: <member>

The <*inherits*> tag

Function: interface inheritance
Attributes: name
Contains: nothing

The <*attribute*> tag

Function: attribute of an interface
Attributes: readonly type name
Contains: nothing

The <*operation*> tag

Function: operation on an interface
Attributes: oneway type name
Contains: <parameter> <raises> <context> <send><receive>

The <*parameter*> tag

Function: specifies an operation parameter
Attributes: mode type name
Contains: nothing

The parameter mode is one of in, out, or inout.

The <*raises*> tag

Function: raised exception
Attributes: name
Contains: nothing

The <*context*> tag

Function: idl context
Attributes: value
Contains: nothing

The <*invariant*> tag

Functions: defines an invariant relationship
Attributes: interface module
Contains: constraint expression

The <*receive*> tag

Functions: Defines a valid behaviour on the service
Attributes: none
Contains: pre and post conditions

The <*send*> tag

Functions: Defines a valid behaviour on the client
Attributes: none
Contains: pre and post conditions

That concludes the set of tags necessary to define the service signature. The signature alone can be used to generate client and server stubs. These are skeletal pieces of code that a programmer can use as a starting point in their implementation of a client or a service. Standard Java mappings will be accommodated, and this work-package will additionally look at the possibility of creating client stubs in the task agent language.

Appendix C Interface Specifications

Interface `Trader`

The Trader interface defines the functionality provided by a Trader implementation and allows a client to request an interface based on it's type and a list of properties that define the services operational characteristics.

Methods

```
TaggedList resolve (String name, String prop_constraints, PolicyList policies)
     throws BadNameEx, NameNotFoundEx, BadPropertyEx, DuplicatePropertyEx,
     MandatoryMissingEx, BadPolicyEx, BadConstraintEx;


OfferId export (String name, Tagged interface, PropertyValList props)
     throws BadNameEx, BadPropertyEx, DuplicatePropertyEx, MandatoryMissingEx;


OfferInfo modify(OfferId offerId, PropertyValueList props)
     throws BadOfferIdEx, BadPropertyEx, ModifyStaticEx;


OfferInfo describe(OfferId offerId)
     throws BadOfferIdEx;


void withdraw (OfferId offerId)
     throws BadOfferIdEx;


ServiceProfiles[] list ();


ProfileId registerSprofile(sProfile);


void removeSProfile(profileId)
     throws BadProfileIdEx;
```