14

Enabling Interworking of Traders

Andreas Vogel[a] andreas@dstc.edu.au
Mirion Bearman[b] myb@ise.canberra.edu.au
Ashley Beitz[a] ashley@dstc.edu.au

[a] CRC for Distributed Systems Technology
DSTC, Level 7, Gehrmann Laboratories
University of Queensland, 4072, Australia

[b] CRC for Distributed Systems Technology
Faculty of Information Sciences and Engineering, University of Canberra
Bruce 2616, Australia

Abstract

To enable the interworking of traders, two type based problems have to be solved: the transformation of type identifiers when an interworking operation crosses a type boundary and a typed definition of policy rules to enable the passing of rules between interworking traders. Our solution to the first problem is based on a refined type definition for type identifier, a protocol to transform type identifiers between different type description languages and an identification mechanism for type managers. To resolve the second problem we have formalised the definition of policy rules. We show its application to specify importer, trader and global search policies. The proposed solutions are being prototyped in the framework of a global project for interworking amongst heterogeneous traders.

Keyword Codes: C.2.4; H.4.3
Keywords: Distributed Systems; Information Systems Application

1. Introduction

1.1. Trader overview

Distributed systems span heterogeneous software platforms, hardware platforms, and network environments. In order to utilise services in such systems, service users have to be aware of potential services and service providers. Furthermore, the locations and versions of services change quite frequently in large distributed systems, which makes late binding between service users and service providers a useful feature. To support late binding, mechanisms to locate and access services dynamically have to be provided. ODP's trading function provides a mechanism for dynamically finding services, and is realised by a trader.

A trader is a third party object that enables clients to find information about suitable services and servers. Services provided by service providers can be advertised in, or exported to,

K. Raymond et al. (eds.), *Open Distributed Processing*
© Springer Science+Business Media Dordrecht 1995

a trader. Such advertisements, known as service offers, are stored in the trader's database. The advertising object (the service provider or object acting on behalf of the service provider) is called an exporter. Potential service users (importers) can import from the trader, which means obtaining information on available services and their accessibility. The trader tries to match the importer's request against the service offers in its database. After a successful match the importer can interact with the service provider.

More detailed information about the trader can be found in the ODP Trading Function Standard, [14] which has reached the '*committee draft*' status and in a tutorial on the trader [2]. A number of prototypes of the trader have been implemented for a number of middleware platforms, e.g. DCE [4,19,22], ONC [17], ANSAware [1], and CORBA [16]. The trader has also been specified with various formal description techniques [11, 9].

1.2. Interworking traders

A trader usually covers the service offers of a particular domain. Domain boundaries can be administrative (e.g. organisations or divisions), topological (e.g. a local area network), technological (e.g. a DCE cell), etc. These boundaries need to be crossed to locate and use services in a large, potentially world-wide, distributed environment. Interworking (formerly called federation) of traders enables the discovery of services outside of a trader's domain, with domain boundaries being transparent to a service user. Interworking trader protocols have been proposed since 1991, see e.g. [3, 18,21,16].

The current Trading Function Standard[14] documents the basic agreements on interworking. Operations for interworking are provided at the trading interface. The operations are: *import*, *export*, *withdraw*, and *modify*. Links contain interface identifiers to linked traders for interworking. Information on the capability of the linked traders and information on what is on offer at these traders are expressed as link properties.

Additionally, it has been agreed that there are trader policies and importer policies that control interworking. Trader policies that affect interworking include global search policy, resource consumption policy, and domain boundary crossing policy. Importer policies define an importer's intended scope on an import operation. These policies are expressed as an importer policy parameter of the import operation. The resultant policy used to effect an import operation across traders is the unification of an importer's import policy with the trader policies of individual traders linked for interworking.

1.3. Motivation, problem description and outline

To successfully interwork two interacting traders must have the same understanding of the parameters in the interworking operations. The type definitions of these parameters are defined in the trading interface specification. However, there are the following two typing problems:

Type identifier

In ODP systems, a type repository function manages a repository of type specifications and type relationships [12, 15]. A type specification defines the syntax and the semantics of the type and is defined using a type description language. In addition, a type identifier is associated with each type specification and acts as a pointer into the type repository for the retrieval of the type specification. The type repository function can be provided as part of a trader, e.g. in the ANSAware trader [1] and a X.500 based

trader [23] or it can be provided as a separate object, e.g. in the DSTC's infrastructure [4, 7]. In this paper, we call the object that provides the ODP type repository function, a type manager.

Type identifier problems arise from interworking traders because identifiers can originate from different type managers. Each type manager may

- use internally different naming systems for type identifiers,
- support different type description languages.

In Section 2 we propose a type definition for type identifiers and a protocol between type managers and traders to solve the problems listed above.

Policy rule

In the computational viewpoint, polices are expressed as rules. Each rule represents a single policy and is expressed as a proposition over both link properties and trader properties. Specifically, policy rules are specified as *'a proposition that a named property either exists or has a specified relationship to a stated value'* [14]. However, in order to pass such policy rules as parameters and to unify rules from different traders, such rules and rule types have to be formally specified. In Section 3 we introduce a formalisation of rules by the definition of a minimal rule language and demonstrate how this rule language can be used to specify importer policy rules and trader policy rules.

In order to validate our interworking ideas, a project on interworking of heterogeneous trader implementations has been initiated. Section 4 gives an overview on design decisions and results reached so far for the project. Finally, Section 5 concludes the article and outlines some future work.

2. Interworking over type domain boundaries

In this section, we first introduce a refined type definition for type identifiers to facilitate the crossing of type boundaries. We then suggest a protocol between traders and type managers to obtain a common understanding of types between interworking traders. We illustrate this protocol using service type identifiers. The term 'type domain' is used to describe the scope of a particular type manager.

2.1. Type identifier

To accommodate the identification of type definitions for a specific type repository we suggest the following definition for a type identifier. A type identifier consists of two components:

- an interface identifier which points to a type manager
- an opaque identifier which points to an entry within the type manager.

The opaque identifier determines the type definition. It is raw data to be interpreted by the type manager addressed by the interface identifier. Interface identifiers are discussed separately in the following section.

2.2. Interface Identifier

The Trading Function Standard does not further specify what a (computational) interface identifier is. We propose the following working definition:

Interface identifiers are divided into direct identifiers and indirect identifiers. A direct interface identifier determines an interface in a particular middleware environment. Currently we support (in the interworking trader project) the DCE and the CORBA environment. A DCE identifier can be represented either by a string binding or a Cell Directory Service CDS entry name. A CORBA identifier is represented by an object reference.

Indirect identifiers have two components. One component contains the actual identifier as raw data, the other component determines the type of this raw data. The latter component is a type identifier as described in the previous section.

To avoid endless levels of indirection during interworking, the interface identifier of the type manager is always a direct interface identifier.

More detailed research on service interface identifiers is reported by the authors in [5].

2.3. Example

The following example shows a type identifier. It uses DCE unique universal identifiers as internal identifiers. The interface identifier of the type manager is given as a direct identifier within a DCE environment using a CDS entry name.

• interface identifier of the type manager:

```
kindInterface = direct
middleware = DCE
dceKindBinding =  bindByCds
dceCdsBinding =  "/.:/a2/andreas/tm_server"
```

• opaque identifier within the type manager:

```
internalIdentifier = "2e49c9ea-3105-11ce-b3ca-08002bbceeee"
```

2.4. Type identifier transformation protocol

In this section we introduce a protocol to transform type identifiers between different type domains. We illustrate the protocol by the import operation and the service type identifier.

Figure 1 illustrates a scenario with two type domains represented by two type managers, *Type Manager 1* and *Type Manager 2*. The two traders, *Trader 1* and *Trader 2*, are in the respective type domains and *Trader 1* has a link to *Trader 2*. An exporter has exported a service with a certain type to *Trader 2*. The service type is expressed by an identifier `ServiceTypeIdX`, which points to a type description stored in *Type Manager 2*. An importer requests a service offer of the type identified by `ServiceTypeIdY`, a pointer into the type repository of *Type Manager 1*. The service types to which `ServiceTypeIdX` and `ServiceTypeIdY` point are assumed to be equivalent.

Given the above scenario, interworking (illustrated for the import operation) can be achieved for the following cases:

(i) *ignorant*

```
import( ..., ServiceTypeIdY, ...)
```

Trader 1 is not concerned about crossing type domains. It passes the requested type identifier over to *Trader 2*, and *Trader 2* has to deal with this type identifier.

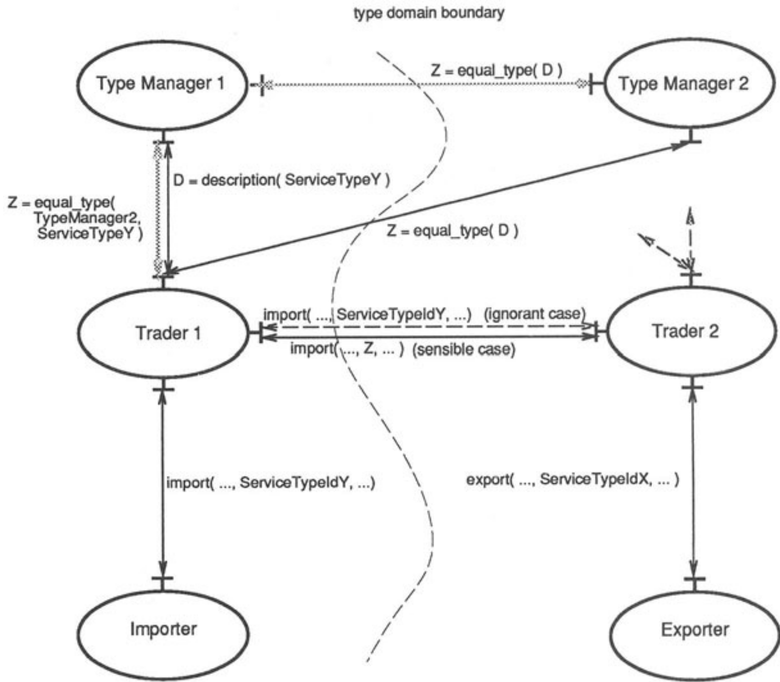

Figure 1. Interworking crossing a type domain boundary

(ii) *sensible*

`import( ..., Z, ...)`

Trader 1 produces a type identifier valid in the type domain 2 before interworking with *Trader 2*.

In both cases, the given type identifier `ServiceTypeIdY` has to be transformed into one which is understood by the *Type Manager 2*. The transformation process has two steps. The first step, performed by *Type Manager 1*, is the de-referencing of the type identifier `ServiceTypeIdY` into type description `D`. In the second step, *Type Manager 2* finds a type identifier in *Type Repository* 2 which matches the type description `D`.

Computationally, there are two ways to do the transformation:

explicit
: the trader obtains the type description from *Type Manager 1* and asks *Type Manager 2* of a matching type identifier. In Figure 1 this is indicated by the black lines between *Trader 1*, *Type Manager 1*, and *Type Manager 2*.

type manager interworking
: the trader asks its type manager to obtain an identifier for an equivalent type. The type manager employs a type manager interworking protocol to get the requested type identifier, as illustrated by the gray lines in Figure 1.

To apply any of the above variants of type domain crossing mechanisms, the issue of a type description language shared by the two type managers still needs to be addressed.

2.5. Identification of a type manager

A trader has to know the type manger of the other domain to understand service type identifiers as motivated in the above scenario. The issue here is how a trader acquires knowledge about the interface identifier of another domain's type manager. In the given scenario, either *Trader 1* (*ignorant* case) or *Trader 2* (*sensible* case) has to know the interface identifier of *Type Manager 2* or *Type Manager 1*, respectively. In the first case, this knowledge could be expressed as a link property.

We define properties as a list of triplets containing the property name, the property type and the property value. The interface identifier of the linked trader's type manager can be expressed as a link property. An example of a value specification has been given in Section 2.3.

```
property name = "type manager interface identifier"
property type = InterfaceIdentifierType
property value = ...
```

2.6. Type description languages

To enable type identifier transformations, two interworking traders/type managers must agree on a common type description format. The most obvious method is to use a standardised type description language which is supported by all type managers. However, a number of such type languages are already established in their respective domains, for examples, OMG's CORBA-IDL [20], OSF's DCE-IDL [24], OSI's ASN.1 [13], and Microsoft's MIDL [10]. It therefore seems infeasible, and contrary to the ODP approach, to aim for a single standardised type description language. Our approach accommodates the co-existence of a variety of type description languages. That is, we acknowledge the existence of the above type languages and use them to form a set of exchangeable type languages.

There remains the issue of how to agree on a common type description format between two traders (and their type managers). In the case of an assumed type manager interworking protocol, the issue is delegated to this protocol. For the *sensible* case the set of type description languages supported by a trader (and its type manager) could be expressed as another link property, e.g.

```
property name = "supported type description languages"
property type =  TypeDescriptionLanguageSetIdentifier
property value = { DCE-IDL, CORBA-IDL } .
```

Another solution is to directly enquire the type manager for the set of supported type description languages.

The bootstrap problem, i.e. the definition of unique names for type description languages, can be solved by registration of the names with a standardisation organisation.

2.7. Discussion

The proposed type identifier transformation protocol allows for a number of options. We favour the *sensible* case over the *ignorant* one. The decision is due to the asymmetry of links, i.e. a trader stores properties of the trader that it is linked to but not vice versa. The use of link properties simplifies the type identifier transformation protocol.

From the architectural point of view of a distributed system, the type manager interworking has the advantage of hiding the type identifier transformation from the interworking traders. However, there is neither a standard for type management nor a standard for type manager interworking protocols. For prototyping in the trader interworking project, the explicit transformation option is used. However, within the DSTC's architectural model approach [6], the option of type manager interworking will be supported.

3. Policy Rules

In this section, we first introduce our minimal rule language for the specification of policy rules. This is followed by examples of applications of the language to important policies required for interworking.

3.1. Minimal rule language

Policy rules are specified in the computational language as *'a proposition that a named property either exists or has a specified relationship to a stated value'* [14]. We have formalised the proposition by the definition of a minimal rule language. The language contains two fundamental boolean expressions:

```
exist( property )
```

which is true if the property `property` exists; and

```
relationship( relationship, property, value )
```

which is true if the value of property `property` and the value `value` satisfy the relationship `relationship`. We have already introduced the type of a property in Section 2.5. The type of `value` has to be of the same type as that specified for the property. Additionally, the logical operators NOT, AND, and OR can be used to construct more complex expressions. This minimal rule language has been included in the type definition of the trading specification used in interworking trader project[1].

Associated with each relationship is a relationship type which is managed by the type manager. The typing of relationships removes the need for the standardisation of any relationship and provides freedom in the definition of rules. Within our type definitions a relationship is expressed by a type identifier as defined and illustrated in Sections 2.1-2.3. An example of a relationship specification (as stored in the type manager) is shown below:

name: ==
types: integer, integer
semantics: is true if x and y have the same value, otherwise false

---

[1] The DCE IDL version can be accessed at the following URL: http://www.dstc.edu.au/public/iwt/proxy.dce.idl .

3.2. Importer policy rules

Importer policy rules specify constraints on the scope of the import operation such as

- on the set of traders to be searched,
- on the use of resources, and
- on the domains to be crossed.

The following examples illustrate the use of our minimal rule language to express an importer policy.

An example of constraints on the set of traders to be visited is where the importer only wishes to visit traders which trade a certain category of service types (which is also known as abstract service types). The corresponding link property is here referred to as `abstractServiceType` and has the service identifier type. If the importer is only interested in meteorology services identified by, say, `weatherServiceId` and has the service identifier type, then this can be expressed as:

```
relationship( subtype, abstractServiceType, weatherServiceId )
```

An example of use of resources is where an importer wants to make sure that no fees are required for searching a trader. This can be expressed as:

```
(exist( fee ) AND relationship (==, fee, 0)) OR NOT exist(fee)
```

An example of domains to be crossed is where an importer wants to ensure that only traders located in Australia are used. If a trader property `traderLocation` is assumed, then the constraint can be expressed as:

```
exist( traderLocation ) AND relationship( EQ, traderLocation,
"Australia" )
```

3.3. Trader policy rules

Trader policy rules that impact on interworking include:

- resource consumption policy rules and
- domain crossing policy rules.

An example of a resource consumption policy is where the total time for an import operation must be less than 10 seconds which can be expressed as:

```
relationship( <, searchTime, 10)
```

An example of a domain crossing policy is to limit the crossing of organisational domain to a single domain.
The corresponding trader property is assumed to be `organisation`. The constraint to interwork only with traders within the DSTC can be expressed as:

```
relationship( EQ, organisation, "DSTC")
```

3.4. Global search policy rules

Global search policies specify how to traverse the graph of linked traders. Global search policies can be included in importer policies and in trader policies. To illustrate the specification of global search policy rules, we assume that the predefined properties `searchDirectionPriority`, `searchLocationPriority` and `searchMethod` have the following enumeration types, respectively:

search direction priority = { depthFirst, widthFirst }
search location priority = { localFirst, remoteFirst }
search method = { sequential, broadcast }

An example of a valid global search policy specification would be:

```
relationship(IN, searchDirectionPriority, depthFirst) AND
relationship(IN, searchLocationPriority, localFirst) AND
relationship(IN, searchMethod, sequential)
```

When traversing the graph of linked traders, loops can occur. There are two methods to avoid loops. One method is for a trader to keep track of the operations that have visited it. This requires that operations must be uniquely identifiable. The other method is for an operation to keep track of the traders that it has visited. This information can be stored in the importer policy parameter of the import operation as it initiates a search on another trader. This parameter can be expressed using the minimal rule language as follows:

```
NOT relationship( IN, link.traderId, visitedTraders )
```

where `link.traderId` is the trader identifier to which a certain link is pointing and `visitedTraders` is the set of identifiers of traders which have already been visited by the operation.

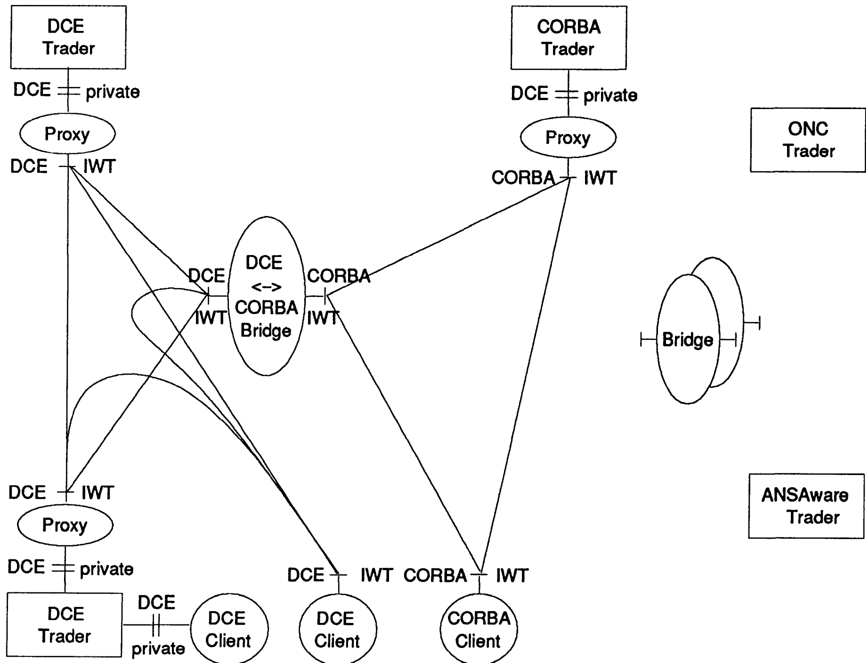

Figure 2. Test-bed of the interworking trader project

4. Prototyping

To validate the proposals made in the previous sections the interworking trader project[2] has been initiated to achieve and demonstrate interworking of traders existing in heterogeneous environments.

The trader prototypes provided by the project partners differ in both the specification of the interface (due to the frequency of change in versions in trader working documents) and the heterogeneity of middleware domains that the traders are implemented in. Interface definitions present a dilemma since private interfaces serve well in particular systems but interworking requires common interfaces. To overcome this, an ad hoc proxy approach is used. A proxy supports two interfaces:

- a refined 'standard' one (which includes all the proposals made in this paper) and
- a 'private' one (which supports the local trader implementation).

For interworking over different middleware domains, we suggest a similar ad hoc solution - bridges which provide interfaces to two middleware domains. Such bridges seem to be easily implementable, but the problem of a general ODP infrastructure [8] remains unsolved. Proxies as well as bridges transform parameters between their respective interfaces.

Figure 2 illustrates the architecture of the project's test-bed based on proxies and bridges where the project defined interfaces are labelled IWT (InterWorking Traders). The IWT interface contains all the proposals on type refinements made in the previous sections. There are equivalent specifications of the IWT interface in both, DCE IDL and CORBA IDL. The transformation between them has been facilitated by a tool set currently under development at the DSTC. Within the DSTC, the DCE proxy has been implemented The implementation of a DCE-CORBA bridge is under development at DSTC.

5. Conclusions and Future Work

We have identified two type based problems which hinder interworking of traders:

- the transformation of type identifiers when an interworking operation crosses a type boundary and
- a formal definition of policy rules to enable the passing of policy rules between interworking traders.

Our proposed solution to the first problem is based on a refined type definition for type identifiers, a protocol to transform type identifiers between different type managers, and an identification mechanism for type managers. To resolve the second problem we have formalised the definition of policy rules and have shown its application to specify importer, trader and global search policies.

To validate the proposals we have initiated the interworking trader project. Within this project, a refined trading interface specification, which includes all the type refinements proposed in this article, has been defined. Furthermore, interworking through this refined interface has also been achieved.

The implementation of further infrastructure objects, i.e. proxies and bridges, will continue and will enable more traders to participate in the interworking project.

---

[2] Find details about the project at: http://www.dstc.edu.au/public/iwt/welcome.html .

Apart from the interworking trader project, we are working on the enhancements of our type manager, in particular in the transformation of type descriptions from one language into another. Tools for DCE IDL↔CORBA IDL transformations are nearly completed.

Finally, we propose to the ISO standardisation body for the Trading Function that the requirements we have found for interworking be considered for inclusion in the Trading Function.

6. Acknowledgements

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the department of the Prime Minister and Cabinet of Australia.

The authors wish to thank our colleagues at DSTC, in particular, Jaga Indulska, Kerry Raymond, and Andy Bond, for the many fruitful discussions. Also thanks to all who helped to get the interworking trader project off the ground, in particular to Kay Müller (Univ. of Hamburg), Mike Beasley (APM) and Richard Soley (OMG).

References

1. "The ANSA Reference Manual.," Architecture Projects Management Limited, Cambridge (1989).
2. M. Bearman, "ODP-Trader" in *Open Distributed Processing, II,* ed. J. de Meer, B. Mahr, and S. Storp, pp. 19-33, North-Holland (1994).
3. M. Bearman and K. Raymond, "Federating Traders: an ODP Adventure." in *Open Distributed Processing,,* ed. J. de Meer, V. Heymer, and R. Roth, pp. 125-143, North-Holland (1992).
4. A. Beitz and M. Bearman, "An ODP Trading Service for DCE" in *Proccedings of the First International Workshop on services in Distributed and Networked Environments,* pp. 34-41, IEEE Computer Society Press, Los Alamitos, CA (1994).
5. A. Beitz, M. Bearman, and A. Vogel, "Service Location in an Open Distributed Environment," Technical Report #21, CRC for Distributed Systems Technology, Brisbane (1995).
6. A. Berry and K. Raymond, "The A1! Architectural Model" in *Proceedings of 2nd International Conference on Open Distributed Processing,* ed. K. Raymond and J. de Meer (1995, in press).
7. C.J. Biggs, W. Brookes, and J. Indulska, "Enhancing Interoperability of DCE Applications: A Type Manager Approach" in *Proccedings of the First International Workshop on services in Distributed and Networked Environments,* pp. 42-49, IEEE Computer Society Press, Los Alamitos, CA (1994).
8. A. Bond and D. Arnold, "An Approach to Implementing an ODP Infrastructure" in *Proceedings of 1994 ACS Asia-Pacific Conference,* pp. 9-16, Gold Coast, Queensland, Australia (September 15-18, 1994).
9. J.S. Dong and R. Duke, "An Object-Oriented Approach to the Formal Specification of ODP Trader" in *Open Distributed Processing, II,* ed. J. de Meer, B. Mahr, and S. Storp, pp. 312-323, North-Holland (1994).
10. G. Eddon, *Network Remote Procedure Calls: Windows NT, Windows, DOS,* P-H (1993).

11. J. Fischer, A. Prinz, and A. Vogel, "Different FDTs Confronted with Different ODP-Viewpoints of the Trader" in *FME'93: Industrial Strength, Formal Methods,* ed. J.C.P. Woodcock and P.G. Larsen, pp. 332-350, LNCS 670 Springer Verlag (1993).
12. J. Indulska, K. Raymond, and M. Bearman, "A Type Management System for an ODP Trader" in *Open Distributed Processing, II,* ed. J. de Meer, B. Mahr, and S. Storp, pp. 169-180, North-Holland (1994).
13. ISO, *Information Processing - Open systems Interconnections - Specification of Abstract Syntax Notations One (ASN.1).,* IS 8824, International Standard.
14. ITU/ISO, "Reference Model of Open Distributed Processing - Trading Function," Committee Draft ISO/IEC/JTC1/SC21 N807 (October 1994).
15. ITU/ISO, "Reference Model of Open Distributed Processing - Part 3: Prescriptive Model," Draft International Standard 10746-3, Draft ITU-T Recommendation X.903 (1994).
16. L. A. de Paula Lima Jr. and E. R. Mauro Madeira, "A Model for a Federated Trader" in *Proceedings of 2nd International Conference on Open Distributed Processing,* ed. K. Raymond and J. de Meer (1995, in press).
17. L. Kutvonen and P. Kutvonen, "Broadening the User Environment with Implicit Trading," in *Open Distributed Processing, II,* ed. J. de Meer, B. Mahr, and S.Storp, pp. 157-168, North-Holland (1994).
18. L. Kutvonen, "Federation transparency in ODP Trading function," Technical Report C-1994-32, Department of Computer Science, University of Helsinki (May 1994).
19. M. Merz, K. Müller, and W. Lamersdorf, "Service Trading and Mediation in Distributed Computing Systems" in *Proceeding of the 14th International Conference on Distributed Computing Systems,* pp. 450-457, IEEE Computer Society Press, Los Alamitos, CA (1994).
20. Object Management Group and X/Open, *The Common Object request Broker: Architecture and Specification* (1991).
21. C. Popien and B. Meyer, "Federating ODP Traders: An X.500 Approach" in *Proceedings of the International Conference on Communications,* pp. 313-317, Geneva.
22. A. W. Pratten, J. W. Hong, J. M. Bennett, M. A. Bauer, and H. Lutfiyya, "Design and Implementation of a Trader-Based Resource Management System" in *Proceedings of CASCON'94,* Toronto (1994).
23. A. Waugh and M. Bearman, "Designing an ODP Trader Implementation using X.500" in *Proceedings of 2nd International Conference on Open Distributed Processing,* ed. K. Raymond and J. de Meer (1995, in press).
24. X/Open Company Ltd., *X/Open Preliminary Specification X/Open DCE: Remote Procedure Call* (1993).