ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

OSI DIRECTORY ACCESS SERVICE AND PROTOCOL

TR/32

December 1985

Free copies of this document are available from ECMA,
European Computer Manufacturers Association
114 Rue du Rhône – 1204 Geneva (Switzerland)

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

OSI DIRECTORY ACCESS SERVICE AND PROTOCOL

TR/32

December 1985

BRIEF HISTORY

This ECMA Technical Report is one of a set of standards and technical reports for Open Systems Interconnection. Open Systems Interconnection standards are intended to facilitate homogeneous interconnection between heterogeneous information processing systems. This Technical Report is within the framework for the coordination of standards for Open Systems Interconnection which is defined by ISO 7498.

This ECMA Technical Report is based on the practical experience of ECMA member companies world-wide, and on the results of their active participation in the current work of ISO, CCITT and national standard bodies in Europe and the USA. It represents a pragmatic and widely based consensus.

A particular emphasis of this Technical Report is to specify the homogeneous externally visible and verifiable characteristics needed for interconnection compatibility, while avoiding unnecessary constraints upon and changes to the heterogeneous internal design and implementation of the information processing systems to be interconnected.

In the interest of a rapid and effective standardization, this Technical Report is oriented towards urgent and well understood needs. It is intended to be capable of modular extension to cover future developments in technology and needs.

Adopted by the General Assembly of ECMA as ECMA TR/32 on December 12, 1985.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

1. GENERAL

1.1 Scope

This ECMA Technical Report:

- Describes an overall model for an OSI Directory System;
- Defines a naming convention to be used when accessing an OSI Directory System by means of a Directory Access Protocol;
- Defines an information structure to be used to declare and retrieve information stored by an OSI Directory System by means of a Directory Access Protocol;
- Specifies a Directory Access Service and a Directory Access Protocol conforming to the overall model, the naming convention and the information structure.

This Directory Access Service and Protocol is a functional subset of what will ultimately be required in OSI Directory Standards. Major areas outside the scope of this ECMA Technical Report are:

- Directory System Service and Protocol, by which the providers of the Directory System interact to maintain consistency among themselves:
- Service Elements for administration of the Directory System.

This Technical Report does not define or describe either implementation design characteristics or intra-system interfaces.

1.2 Field of Application

The Directory Access Service and Directory Access Protocol defined in this ECMA Technical Report may be used to access a Directory System containing information about an OSI Environment or other environments, with the following provisos:

- There is no requirement for instantaneous global commitment of updates: transient conditions, where both old and new versions of the same information are available, must be tolerated by users.
- It is a characteristic of the Directory System that, except as a consequence of differing access rights or as yet unpropagated updates, the results of directory queries will not be dependent on the identity or location of the enquirer. (This characteristic may render the Directory System unsuitable for some telecommunications applications, for example, some types of routing).

1.3 References

ECMA TR/31	Remote Operations: Notation, Concepts and Connection-Oriented Mappings
ISO DIS 8824	Abstract Syntax Notation.1 (ASN.1)
ISO 7498	Data Processing Open Systems Interconnection Basic Reference Model
CCITT Rec. X.409	Message Handling Systems: Presentation Transfer Syntax and Notation

CCITT Rec. X.410 Message Handling Systems: Remote Operations and Reliable Transfer Server (Chapter 2)

1.4 General Overview

A Directory System (DS) is a facility which supports the storage and interrogation of information about named things or people. It is simpler than a general-purpose database in matters such as the structure of the information stored and the procedures for updating it.

Several needs have been identified for a Directory System to be used in the context of Open Systems Interconnection (OSI): among these are needs for mapping application names to addresses, mapping the names of people to mail addresses, and expanding named distribution lists.

Such a Directory System for OSI may be centralized in special cases, but also can be distributed and replicated. Thus there is a need for standards for: a Directory Access Protocol, by means of which the Directory System can be accessed by its users; a Directory System Protocol governing the interactions between the providers of the Directory System when the Directory System is distributed and/or replicated; and Service standards related to these two protocols.

This ECMA Technical Report defines a Directory Access Protocol and the associated Service for OSI, concentrating on urgent and well-understood needs. It makes full use, both of current experience in this technical field, and of the current state of the art in OSI Directory standardization in ISO and CCITT.

Section 2 describes the overall architecture of an OSI Directory System, and thus provides the architectural context for the Directory Access Service and Protocol. This architectural context covers the following matters:

- Overall configuration of the Directory System;
- Positioning of the Services and Protocols within the OSI Application Layer;
- The form of Names used to access the Directory System;
- A basic capability for Authentication of the initiator of an access request.

Section 3 defines the Service Elements of the Directory Access Service, and the Directory Access Protocol. The definition uses the formal notation for specifying Remote Operations which is defined in CCITT Rec. X.410, Chapter 2, and use of this notation has the combined effect of specifying the Service Elements, Service Primitives and Protocol Data Units. Clause 3.10 also specifies how the Protocol Data Units are mapped into the underlying Remote Operation Service defined in ECMA TR/31.

Section 4 defines Conformance to this Technical Report. Conformance is to externally visible behavior as manifested in the Protocol. Option combinations have been kept to a minimum.

1.5 Relationship to Other Standards

This ECMA Technical Report is positioned as a SASE standard in the Application Layer of OSI. It invokes other Application Layer Services which are defined in ECMA TR/31. It uses the notation for Service/Protocol Abstract Syntax Definition defined in CCITT Rec. X.409 and X.410, Chapter 2--equivalently ISO DIS 8824 (ASN.1) and ECMA TR/31.

The Directory Access Service defined in this ECMA Technical Report may be invoked by other OSI Application Layer standards and by private standards outside the scope of OSI.

2. ARCHITECTURE

2.1 General Requirements and Characteristics

2.1.1 Directory System Environment

The OSI Directory System (DS) must exist and provide services within the following environment:

I) Many OSI networks will be large.

II) An OSI network will constantly undergo change.

a) Individual Application Processes (AP), groups of these processes, other OSI resources, and entire end-systems will enter and leave the OSI network.

b) The address, availability, physical location, etc., of Application Processes or other OSI resources may change at any time.

III) Although changes in OSI networks occur frequently, the useful lifetime of any particular Application Process or OSI resource is not short. An AP will typically interact with other APs much more frequently than it will change its address, availability, physical location, etc.

2.1.2 Directory Service Characteristics

The need for a directory service arises from the contrast between the constant change of the OSI network as a whole, and the need to isolate (as far as possible) the user of the network from those changes. Thus, a user of Directory Services is able to view the OSI network as a more stable entity than a network user who is not a user of the directory. For example, if the physical location of a resource in the network changes, then the user of that resource will not be affected by that change--provided that a "name" rather than the physical location was used to reference the resource.

Another need for Directory Services arises from the desire to provide a more "user-friendly" view of the OSI network. For example, the use of aliases, the provision of a network "yellow pages", etc., help to relieve the burden of finding and using network information.

The Directory Service allows a user to obtain a variety of information about the network. The Directory Service provides for the maintenance, distribution, and security of that information.

2.1.3 Facilities of the Service

A Directory System maintains a database consisting of a set of names and, for each name, a set of properties to be associated with the name. The total collection of information managed by the Directory System is known as the Directory Information Base (DIB). The logical structure of the DIB is defined in clause 2.4. The DS assists users, strictly upon request, with dynamic binding of data needed for their operation, and offers the following services:

- Name to property binding service. This service binds names to properties. Name to address binding is an example of the use of this service. This service is analogous to a "white pages" directory.

- Name to set of names binding service. This service binds names to a set of names. An example of the use of this service is distribution lists for electronic mail. This service can be considered as a "mailing list".

- Property to set of names binding service. This service lists the names that have a given property. One example of this is a "yellow pages" directory. Such a directory can be used, for example, to find the set of all names of men who are plumbers, or, in an open system, to find the set of all the names of printers that have Elite typefaces and have wide carriages.

The Directory Service has the ability to distinguish between categories of data. For example, "567" can be in one case a presentation address of a resource, and in another case a password to access a resource. This example tells that one cannot rely on syntax alone in order to recognize the type of information that is being given. The DS described herein distinguishes the semantic class of each property.

The Directory Service provides for use of aliases, which are alternative names for the same object.

2.1.4 Model of the Directory System

In the OSI environment, it seems unlikely that a centralized implementation of the DS will be able to satisfy the functional and organizational requirements of OSI systems. The DS defined herein lends itself to decentralized, i.e. distributed, implementation without precluding a centralized one. Figure 1 shows a functional model of a distributed DS: the DS consists of a collection of Directory Service Agents (DSA) which, when necessary, communicate with each other in order to provide the services to the DS users, and of Directory User Agents (DUA) which reside in the user's End System and handle the protocol for communication between a user and a DSA. A centralized DS is a special case of this model, namely when there is only one Directory Service Agent.

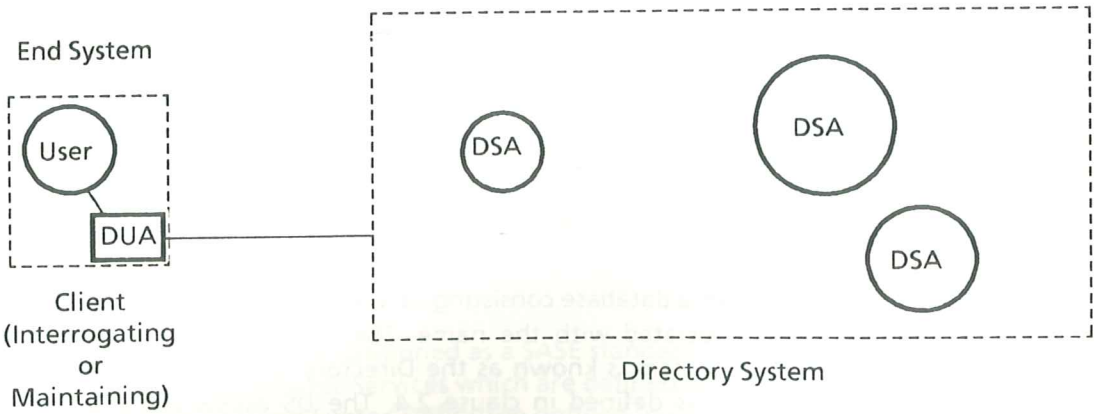

Figure 1. Functional Model

Different classes of service arise from different levels of user privilege, where user privilege means authority to perform certain operations on the directory. The protocol used by an interrogating user is a subset of the protocol used by a maintaining user.

It is an essential feature of the Directory Service that a user has to send the "update" message to only one DSA; the directory will be updated without further work by the user. The multiple DSAs in the distributed DS application communicate with each other in order to maintain the integrity and consistency of directory information. However, after an update has been initiated, there may be a temporary period of inconsistency, which will be resolved when the update has been propagated through all the DSAs that maintain that information. The DS users must, therefore, know that the answers from the DS may occasionally be incorrect, and the users must be able to deal with this possibility.

2.1.5 Performance Characteristics

A standard should consider the effectiveness of implementations in the environments for which it is intended. A DS should be optimized with regards to its performance and reliability, as it is essential to the operation of an OSI network. It must fulfill information needs with minimal degradation of network performance.

This OSI Directory Technical Report should consider the following performance criteria:

- Efficiency of information retrieval (query): Directory information may change frequently, but is typically used far more frequently than it changes.

- Optimization for local environments: In most networks, the majority of interactions occur within local environments, and only a minority occur across their boundaries.

- Information availability: In order to minimize the chance of network failure, replication of part or all of the Directory information base may be desirable; update operations should not lock out large quantities of data; and so on.

2.2 Layered Model

The Directory System is positioned in the OSI Application Layer, and is structured in accordance with a Client-Server model described in the following clauses.

2.2.1 Client-Server Model

In order to discuss the elements of standardisation required for Client-Server type Application protocols, it is necessary to formulate and adopt a viable model for a Client-Server relationship which is aligned to the OSI Reference model, see Figure 2.

2.2.1.1 Server System

A Server System is a functional entity that performs a set of basic and specific application services (the Server Services), by means of Service Agents within the Server System.

The set of services provided by a Service Agent are made available to its Client through an Access Protocol.

One or more Service Agents connected through a network may interact to perform the requested service. In that case they co-operate by means of an inter-service Agent Protocol. When the Server System is provided by a single Service Agent, this is commonly referred to as the "Server".

2.2.1.2 Client System

A Client System is a system in which one or more Clients of a Server System reside.

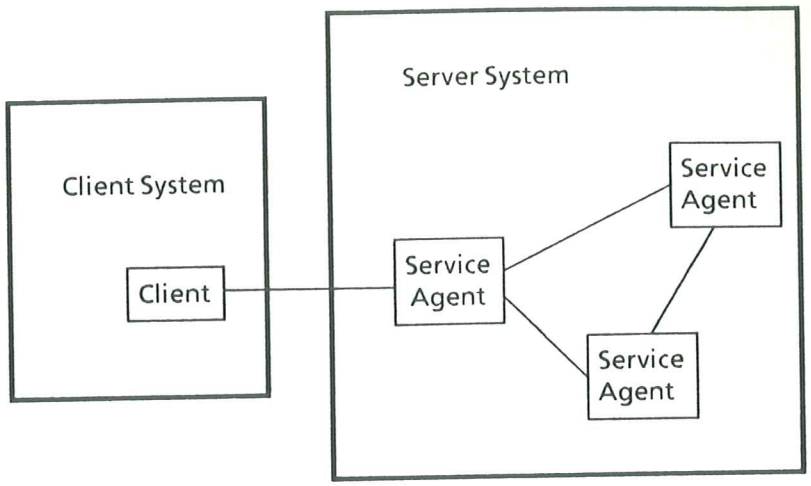

Figure 2. Functional Model of a Client-Server Relationship

A Client is a functional entity that requests services provided by a Service Agent for that particular service.

Clients gain access to a Service Agent by means of an Access Protocol.

A Client may be :

- a User (or User Agent in the general sense);
- a Service Agent of another Service.

2.2.1.3 Access Protocol

An Access Protocol is the standard way for Clients to gain access to a Service Agent. It is the means that allows location of Clients remotely from the Service Agents.

The Acces Protocol consists of two components:

- a service-specific protocol defined for the particular Service Agent;
- a Remote Operation Service (ROS) protocol to allow the invocation of these services.

2.2.1.4 Agent Protocol

An Agent Protocol is the standard way for a Service Agent to gain access to another Service Agent of the same specific Service. It is the means that allows a Service to be distributed between a number of Service Agents.

If the Agent Protocol uses ROS, it consist of two components:

- a service-specific protocol defined for the particular Service Agent.
- a Remote Operation Service (ROS) protocol to allow the invocation of these services.

2.2.2 Layered Representation

The Client-Server layered model in Figure 3 has two sublayers within the Application Layer of the OSI model, the lower of which is the ROS sublayer, allowing the interchange of information between ROS-users. The upper sublayer is the Service sublayer.

The Service sublayer provides access to a service for a single Client. It achieves this by use of resources in more than one OSI end system, and a service-specific protocol (ie. service-specific use of the common ROS interaction paradigm) is used between the cooperating parts of the service "X". Because those parts are complementary entities, rather than the peer entities of the OSI model, it is necessary to distinguish between the entity associated with the User and the entities associated with the "X" Server.

The terminology is as below for service "X":

"X" Client Entity	A component providing part of the "X" Service which is co-located with the User.
"X" Service Agent Entity	A component providing part of the "X" Service which is located remotely from the User.
"X" Access Protocol	The protocol between the "X" Client Entity and the "X" Service Agent Entity (or Server).
Remote Operation Service Entity	The entity which provides the sublayer service for Remote Operations.
Remote Operation Service Protocol	The protocol between a pair of Remote Operation Service entities.

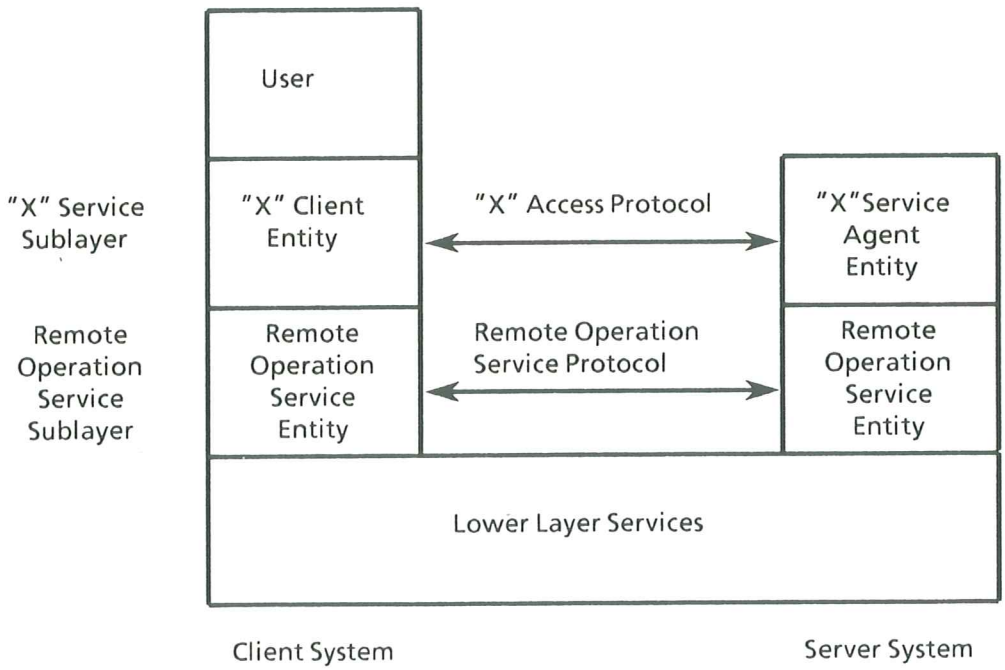

Figure 3. Client-Server Layered Model

For the Agent to Agent communication, the same type of model applies. The only difference is that one of the Service Agents is performing the Client role and the other the

Server role. The protocol used between two Service Agents is called the "X" Agent Protocol.

2.2.3 Use of the Layered Model for Directory

The Directory Access Protocol defined in this ECMA Technical Report is an "X" Access Protocol in terms of the Client-Server model. It operates between a Directory User Agent (DUA) which is a "X" Service Access Entity, and a Directory Service Agent (DSA) which is a "X" Agent Entity.

This Directory Access Protocol is mapped onto the Remote Operation Service defined in ECMA TR/31, Section 5.

This Directory Access Protocol supports a Service, the Directory Access Service, which is also defined in this ECMA Technical Report. The client of this service is called the Directory Service User.

2.3 Directory System Contents

The Directory System manages and answers queries about a set of database entries. An entry consists of a name and a set of properties. The Directory System maintains the set of entries so that it can return a property given a name. Thus, it provides the following mapping:

$$name \rightarrow \{ (type_1, value_1), ..., (type_k, value_k) \}$$

The entire collection of entries is referred to as the Directory Information Base (DIB). The DIB may reside on one DSA or, more commonly, be distributed over many.

Each property consists of a property type, and a property value. There can be at most one property of a given type in an entry. A property is used primarily to hold a network location, or a list of other object names. These two uses define two major types of property value, item and group. Given an object name and a property type, the Directory Service will return the value of that property--either a block of data (for an item property), or a list of names (for a group property).

2.3.1 Names and Their Use

A name is a linguistic construct that singles out a particular object from among a collection of objects. Names are unambiguous, but not necessarily unique. A name must certainly be unambiguous, that is, denote just one object. However, a name need not be unique, that is, be the only name that unambiguously denotes the object.

This ECMA Technical Report uses a naming convention suitable for use in Open Systems Interconnection.

The naming convention is intended to govern the naming of many of the kinds of objects found in the OSI environment, such as application processes, real entities, and, in general, any kind of OSI resource that is convenient to refer to by a name. These objects are engaged in any or all of a wide range of applications, such as OSI management, message handling systems, and on-line data bases.

2.3.2 The Name Tree

The set of names recognized by a Directory System is modelled in this ECMA Technical Report as an oriented tree called the Name Tree. For an example, see Figure 4.

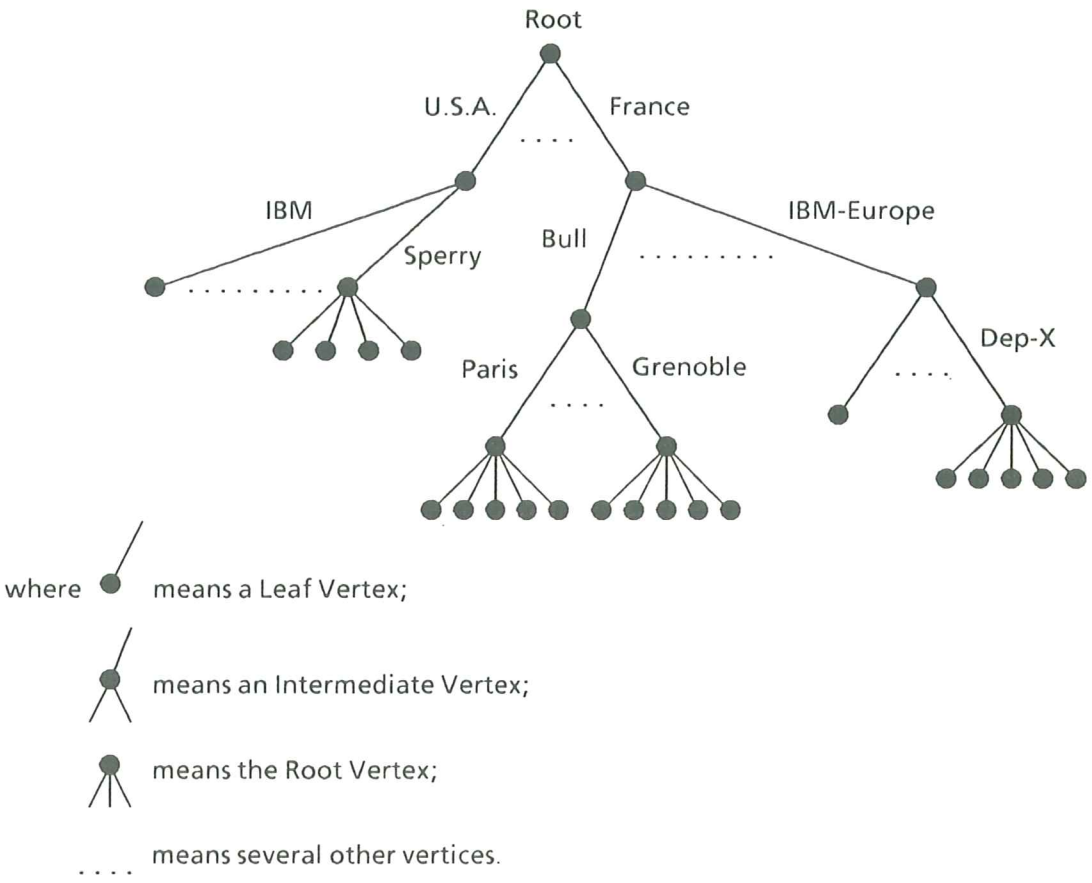

Figure 4. Name Tree Example

If X and Y are vertices of the Name Tree and there is an arc from X to Y, then X is said to be the Parent of Y and Y is said to be a Child of X.

One vertex of the Name Tree has no Parent, and this vertex is called the Root Vertex of the Name Tree. Every other vertex has one, and only one, Parent.

Vertices other than the Root Vertex are either Intermediate Vertices or Leaf Vertices. A Leaf Vertex is not permitted ever to have Children, whereas an Intermediate Vertex can have an unconstrained number of Children (and may also, sometimes, have none). Only Leaf Vertices have entries associated with them. Intermediate Vertices are intermediate points in the hierarchical Name Tree and have no entry (and hence no properties) associated with them.

Every arc of the Name Tree is labelled in such a way that every arc emanating from a single vertex has a different label. (However, arcs emanating from different vertices may have the same label). The labels on the arcs are called Name Parts, and are used to form Names for the Directory Service, as described in the next clause.

2.3.3 The Structure of Names

Every Name used in the Directory Service is constructed as a sequence of Name Parts, such that each Name Part is the label of an arc of the Name Tree. The Name Parts in the sequence, when taken in order, specify a path from the Root Vertex to some other vertex of the Name Tree, and it is this other vertex which is denoted by the Name.

Each Name Part is constructed of a Name Part Type and a Name Part Value. The Name Part Value gives some information about the object being named (however, see the note at the end of this clause), and the Name Part Type tells what class of information this is. For example, a person could be named in some such way as the following (this example is illustrative, not exact):

Country = UK
Company = Smith and Smith
Site = Scunthorpe
Personal Name = J. Smith

In this example, "Country," "Company," "Site," and "Personal Name" are Name Part Types. However, in the Directory Access Protocol, Name Part Types are encoded as integers, for efficiency.

Note 1
Although both Name Part Values and Property Values contain information, there is a great difference. A Property may be specified to contain any information that it is required to hold in the Directory System. Name Parts are specially selected for their use in constructing names, and this selection is strongly influenced by considerations of Naming Authorities and the administration of the Directory System.

2.3.4 Aliases and Distinguished Names

The Directory Service provides facilities which allow one object to be known by more than one Name.

Suppose that it is desired to know a certain object by three different names, say {A, B, C}, {P, Q}, and {W, X, Y, Z}. In this case, one of the Names is designated as the Distinguished Name of the object, and the other Names are called Aliases. The Distinguished Name is the name by which the leaf entity was first created.

The Distinguished Name and the Aliases of an object, being different Names, determine different paths in the Name Tree, and thus designate different vertices of the tree. The fact that these vertices represent the same object is reflected as follows. The vertex denoted by the Distinguished Name is called the Distinguished Entity, and those denoted by the Aliases are called Reference Entities. The Directory System holds with each Reference Entity a Reference Pointer to the Distinguished Entity. Reference Pointers are so organized that only one level of referencing is possible.

If the Distinguished Entity is a leaf vertex, the associated Reference Entities are also leaf vertices. In this case, the Properties are held at the Distinguished Entity and only the Reference Pointer is held at the Reference Entity. The Reference Pointer is dereferenced whenever it is necessary to access the Properties of the object.

If the Distinguished Entity is an Non-Leaf Vertex, the associated Reference Entity is also an Non-Leaf Vertex. When the Directory System encounters a Reference Entity while resolving a Name, it dereferences the Reference Pointer immediately, and continues the Name

resolution proceeding from the Distinguished Entity. Thus, the children of such a Reference Entity are not distinct from the children of the associated Distinguished Entity.

2.3.5 Patterns and Name Comparisons

Patterns are names where only a partial specification of the name is given. The Directory Service takes a pattern and attempts to find a name, or set of names, in the database that matches it. Patterns have the same structure as names. A partial specification is effected by putting a special character, called a wildcard character, in one of the Name Part Values. The occurrence of one or more wildcard characters is what distinguishes a pattern from a name. (A name can be used as a pattern; in this case, the matching process is the trivial one, i.e., a character-by-character comparison.) In the Directory Access Protocol, the wildcard character is the asterisk ("*"). Usually, wildcard characters may only occur in the last Name Part of a name.

A pattern specifies a name in the Directory Service according to the following algorithm: object names are compared for equality with the pattern on a character-by-character basis from left to right; if a character in the pattern is a wildcard character, it is considered to match zero or more other characters of an object name. For example, "A*B" would match "AB," "AAB," "AbB," "ACdEfGB," and in fact any string that begins with an "A" and ends with a "B".

Comparisons of a specified name or pattern with an object name ignore the case of the alphabetics. Thus, a lower and upper case "A" are considered equivalent. In the example above, "A*B" would also match "ab," "aAB," "Abb," etc.

2.3.6 Directory System Naming Authorities and Naming Domains

The structure of the Naming Tree reflects, among other things, the hierarchical distribution of authority to give names to objects in order to register these names in the Directory System.

We reflect this idea by defining the concepts of a Naming Domain (ND) and a Naming Authority (NA), as follows.

The Naming Domain associated with a non-leaf vertex is the complete subtree whose root is that vertex. (Since the subtree is complete, Naming Domains can contain smaller Naming Domains, but otherwise Naming Domains cannot overlap.)

The Naming Authority associated with a non-leaf vertex is that authority whose responsibility is to ensure that all the vertices within the associated Naming Domain have distinct names. Associated with this responsibility is the right to permit or refuse that any object be given a Name within the Naming Domain controlled by that Naming Authority.

If Vertex X is the Parent of Vertex Y, then the Naming Domain of X completely contains that of Y, and, moreover, the Naming Authority for X has complete authority over that for Y. In practice, in some cases, the Naming Authority for X will simply appoint a Naming Authority for Y and allow the latter Naming Authority to function independently. However, in other cses, the Naming Authority for X may choose to act directly as the Naming Authority for Y also. This decision belongs solely to the Naming Authority for X. The Directory System has no awareness of this decision.

The concepts of Naming Authority and Naming Domain are also used in the description of the distribution of the Directory Information Base and in the administration of this distribution. This is described in Clause 2.4 below.

2.3.7 Definition of Name Part Types

Name Part Types belong to three broad classes:

I) A small number of Name Part Types are of completely general applicability, regardless of the type of object represented or the application using the Directory. An example is "Country Name," which is a generally applicable way to partition the top level of the Name Graph of any international Directory System.

II) Some Name Part Types have a specific purpose, which is subject to international standardization. An example is "Application Entity Title," which is specific to the (internationally standardized) naming of Application Entities in the OSI Environment, and cannot validly be used for any other purpose. Such Name Part Types will be defined in standards relating to the special areas for which they are needed.

III) An unpredictable number of Name Part Types may be defined by private enterprises for use in private Directory Systems constructed using this Directory Service. Such Name Part Types may be defined by the private enterprises, but only in such a way as does not impact or constrain international standards.

The functions of the Directory Service make no distinction between the three classes of Name Part Types.

In the Directory Access Protocol, a Name Part Type is encoded as an integer for efficiency. This also makes it simpler for an application to present the user with the type description in the local language by translation from the integer value.

2.3.8 Other Naming Concepts

2.3.8.1 Abbreviations

In certain contexts, it is possible that a Name may be used without the first few Name Parts, which are deduced from the context. (This is similar, for example, to omitting the country code when dialing a telephone call to a receiver in the same country as the caller.) A Name whose initial Name Parts are omitted in this way is called an Abbreviation.

This ECMA Technical Report contains no features to support Abbreviations (though it does not prevent implementations from supporting them at their user interfaces). It is for further study what such features may be included, and whether these features are within the scope of the Directory Service or of higher level interfaces. In any case, we anticipate that this concept will only be used within a DUA or in the dialogue between a DUA and a DSA. It will not be used in the protocol between DSAs, nor will it affect the structure of the Name Tree itself.

2.3.9 Properties of Leaf Entities

A Leaf Entity is a name and its set of associated properties. Different Leaf Entities can have different sets of properties. Each property has a type and a value. As mentioned, there are only two kinds of Property Value, either an item or a group property. The value of an item property is not inspected by the Directory System and may consist of

any data the user wishes. On the other hand, the value of a group property is understood by the Directory System to be a sequence of names called members. The Directory System knows how to add, delete, enumerate, and search for members in a group property.

2.3.10 Classes of Property Types

Although the Directory Service defines some Property Types for its own use and for universal use, an application using the Directory Service will normally define its own Property Types.

The natural language names of Property Types used in Service specifications and in informal descriptions, are not subject to any constraint, not even a requirement for uniqueness (except within the contexts in which they are used). However, the integer values which encode the Property Types in the Directory Access Protocol are required to identify Properties uniquely, taking into account that other applications may also be accessing the same DSA, and even the same Leaf entity.

In order to ensure the required unambiguity, Property Type values in the Directory Access Protocol are integer values.

2.3.11 Allocation of Values For Name Part Types and Property Types

It is anticipated that the international standards organizations (ISO, CCITT) will eventually establish authorities for the registration of Name Part Types and Property Types.

In the absence of these authorities, early implementation must allocate their own values, and in the case of co-operating systems, bilateral agreements must be reached in order to ensure consistent use of Name Part Type and Property Type values.

2.4 The Directory Information Base

The Directory System acts as the manager of the Directory Information Base (DIB), which serves as a global directory for an entire collection of interconnected, mutually-addressable OSI Application Processes (which in this document, we shall term an OSI network). This highly specialized database is not to be confused with general purpose databases designed to support various kinds of management information systems. There is also no implied form or level of file or data management support; that chosen can depend entirely on the local facilities and scale of directory required.

This Technical Report covers the Directory Access Service for enquiry and update. There are, however, a number of other administrative functions which are mentioned in various sections of this document, but are not part of this Technical Report.

These include:

- The administration of access controls.
- The control of replication and distribution of sections of the DIB.
- The verification of sections of the DIB.
- The registration of name and property types.

2.4.1 The Directory Service Agents

The Directory System consists of the collection of Directory Service Agents, which cooperate in order to provide the user visible functionality of the Directory System. Each Directory Service Agent (DSA) is an OSI Application Process having responsibility for some part of the entire DIB, and possessing the capability of directly answering user requests relative to that portion of the DIB, as well as guiding the users towards satisfaction of requests over other portions of the DIB. The part of the DIB maintained by each DSA is called its local DIB.

2.4.2 Directory Information Base Groupings

The Directory System comprises a number of interconnected management domains, each of which comprises one or more Directory System Agents (DSA). The concept of a management domain is related to that of a Naming Domain, as described in the next clause.

2.4.3 Distribution and Replication of the Directory Information Base

The Directory Service is designed to allow distribution and replication of its information base. Each of these two features has important uses in precisely those applications for which this protocol is intended.

If an OSI network contains only one DSA, its local information base is the entire DIB. The DSA is able to answer directly any user request; there cannot be--by definition--any inconsistencies between local information bases. We term this special case centralized. This case will occur frequently in small organizations or in companies developing OSI networks.

Distribution of the information base is defined as the property that there are multiple DSAs, at least one of which has a local informal base which is a proper subset of the DIB.

Replication is defined as the property of some portion of the DIB of being held by multiple DSAs. The DIB may be partially or totally replicated.

The unit of distribution and replication is the Naming Domain. A management domain holds all or none of the information associated with a particular ND.

Replication and/or distribution may occur independently of each other. Thus, in any particular instance of the Directory Information Base, exactly one of the following configurations will be found:

- centralized;
- distributed without replication;
- replicated without distribution;
- distributed and replicated.

The last is the most general case, of which all others are special cases. There are different reasons for moving from an initial centralized implementation to the others, and different implications of doing so.

Replication provides three important benefits:

- reliability of the DIB despite inadvertent damage to a copy;

- availability of the DIB despite server or communication failures;

- efficiency of access to the DIB despite geographically-dispersed access patterns.

The last benefit includes the reduction of traffic loads in internetwork links, which are often the low bandwidth traffic bottlenecks, e.g., in the case of multiple LANs interconnected by routers.

Distribution provides the following benefits:

- ability to handle the DIB of large OSI networks with modest means;

- ability to fully replicate large DIB with modest means;

- ability to configure the database for efficient handling of geographically-dispersed access patterns;

- feasibility of joining two or more OSI networks into a single one with minimal administrative overhead.

The benefits of distribution and replication are sufficiently important, that this Standard supports distribution and replication. These features have, however, the following implications.

2.4.4 Implications of Distribution

Database distribution implies the necessity of "navigating" the database in order to find the data.

In particular environments, the Directory System is a critical OSI resource; in these cases, the Directory System will off-load much of the navigation work on to the DUA, by virtue of the following mechanism.

When a DSA cannot satisfy an operation locally, because the pertinent naming domain does not reside in the local DIB, it tells the DUA that it has contacted the wrong DSA and gives the DUA a hint about where to look next. The hint is a name of a leaf entity that contains properties that locate other known DSAs. The DUA uses this hint to query the Directory System about the locations of the DSAs that do possibly serve the desired Naming Domain. The DUA can then repeat the operation at the proper DSA. This information could be cached so that these extra queries can be avoided on future operations.

2.4.5 Hints and the Process of Database Navigation

This ECMA Technical Report specifies the following aspects of the hint mechanism:

- the form of hints received by DUAs; and

- the operations (i.e., the algorithm) that the DUA must perform using the hint.

This algorithm, by which hints guide the DUA to contact different DSAs, must be guaranteed to converge in a certain number of steps (except in cases where the process encounters transient situations such as inconsistencies of the DIB or other problems). Furthermore, it is important that DUAs know the maximum number of steps in which the

algorithm will converge, so that they are able to detect malfunctioning of the Directory System.

2.4.6 Implications of Replication

Replication within the DIB has two ramifications for modifying a leaf entity. First, when a change is made to a leaf entity, there may be an appreciable delay before all copies of that leaf entity are made consistent. Users must be able to properly contend with the DIB being transiently incorrect. Second, if two copies of a leaf entity are modified "simultaneously" by two users, the Directory System will see to it that, after the updating transient has passed, all copies of the leaf entity will contain the data corresponding to the most recent change. This means that all prior updates will be rejected. The user(s) will be unable to predict which update will prevail.

Propagation of updates in a replicated DIB implies the need for a Directory System Protocol for DSA to DSA communication. It is under consideration to develop a DSA-to-DSA protocol as a companion ECMA Technical Report. Pending completion of that Technical Report , however, it is possible to design distributed Directory Systems conforming to this present ECMA Technical Report, so that administrators maintain the (replicated) data necessary for DIB navigation (see clause 2.4.4) by ad hoc means.

2.4.7 Characteristics of the Directory System

The Directory System is not a general-purpose, shared database system. It is organized so that it can perform efficiently in its role. The Directory System is oriented toward high-performance query and low-performance modification.

The nature of leaf entities should generally conform to the following guidelines:

- A leaf entity could describe the location of a resource, or contain information which has no other natural home. An example of the former is the location of a print service, and of the latter, a membership list.

- Leaf entities should contain data that change slowly, rather than rapidly; and a leaf entity should contain information that is expected to be accessed frequently.

- There should be no problems with the various copies of a leaf entity being transiently incorrect. Because of the distributed nature of the database, it may take an appreciable length of time for a change to a leaf entity to be propagated to all the copies of that leaf entity. The fact that some copies may be out-of-date should present no confusion to users.

2.5 Search Order in the Directory Information Base

Leaf entities in the DIB may be created and deleted while DSAs are responding to queries. Consequently, the user must assume that the content of the DIB may change between operations. There are Directory Access Service operations which, although they allow a pattern to be supplied instead of a name, will operate on only one entity. In these cases, the Directory System will operate on the first entity it finds that matches the pattern. The order in which entities are examined is implementation dependent. Unless the pattern matches only one name in the entire DIB, the user cannot assume that subsequent calls using the same pattern will cause the Directory System to select the same entity. In the description of the various Directory Access Service operations in the next section, reference to the term first entity found refers to these conditions.

2.6 Authentication

This Technical Report supports a first simple, optional level of authentication:

- Level 1, where a user password is conveyed along with the user's name. The password is transmitted unencrypted, in clear. Thus this level of authentication is exposed to the threat of an eavesdropper intercepting the password and using it to masquerade. However, this level provides a simple mechanism which is sufficient for many applications and for the needs of most users of a DS.

Note 2
Because authentication is optional and some operations need to be informed of the name of the user, this Technical Report specifies a facility to convey an unauthenticated user name using the authentication parameter.

(Blank page)

3. SERVICE AND PROTOCOL DEFINITIONS

3.1 General

The Directory Access Service is defined in the form of Remote Operations (cf. CCITT Rec. X.410).

Each Operation description includes a declaration of the Operation in X.409/410 standard notation, a description of the Operation's arguments and results, and occasionally an example of its use.

Use of the X.409/410 notation results in a simultaneous definition of Service Elements and Directory Access Protocol Data Units (DAPSU).

Table 1 lists the Service Elements/Operations described in this section.

This section is organized as follows:

- Clause 3.2 defines the basic date types used in the specification;
- Clause 3.3 defines conventions for the use of context specific tags;
- Clause 3.4 defines the operations applicable to all entities;
- Clause 3.5 defines the operations applicable to non-leaf entities only;
- Clause 3.6 defines the operations on leaf entities;
- Clause 3.7 defines the operations applicable to all properties (property operations are applicable to leaf entities only);
- Clause 3.8 defines operations on the members of group properties;
- Clause 3.9 defines the remote errors used by the operations on Clauses 3.4-3.8;
- Clause 3.10 defines the use of underlying services to carry the DAPDUs.

3.2 Frequently-Used Data Types

3.2.1 Data Types for Names

This clause defines the data types required for naming. A name is composed of name parts. The concepts of name pattern and name specification relate to the search and pattern matching features of the Directory Service.

3.2.1.1 Wildcard Character

It is possible in Directory Service names to specify an arbitrary sequence of characters. This is done by inserting the wildcard character into the name where one wishes to specify zero or more occurrences of "I don't care" characters:

```
wildcard IA5String ::= "*"
```

The circumstances in which a wildcard character may be specified are restricted. The allowed uses of wildcard characters are defined in the following clauses.

Set	Subset	Service Elements/Operations
Name Elements	For Leaf and Non-Leaf Entities:	Read Distinguished Name Read Aliases Of Create Alias Delete Alias
	For Non-Leaf Entities Only:	Read Children (with or without Filter) Read Children Types Read Alias Children
	For Leaf Entities Only:	Create Leaf Entity Delete Leaf Entity
Property Elements	For All Properties:	Read Properties Read Property Types Add Property Change Property Delete Property
	For Members in Group Properties:	Read Group Members (with or without Filter) Is Member Add Group Member Add Self Delete Group Member Delete Self
Common Feature Elements		Patterns Using Wildcards Authentication Filters (see above)

Table 1. Overview of Service Elements/Operations

3.2.1.2 Name Parts

```
NamePartType :: = INTEGER

NamePartValue :: = IA5String

NP :: = SEQUENCE {
   nptype [0] IMPLICIT NamePartType,
   npvalue [1] IMPLICIT NamePartValue}

NamePart :: = NP --the npvalue shall not contain a wildcard--

NamePartPattern:: = NP --the npvalue may contain a wildcard--
```

A **NamePart** and a **NamePartPattern** are identical, except that the later may contain wildcard characters.

3.2.1.3 Vertex Names in General

```
Name :: = SEQUENCE of NamePart

NamePattern :: = SEQUENCE {
   nonfinal [0] IMPLICIT SEQUENCE OF NamePart,
   final [1] IMPLICIT NamePartPattern}

NameSpecification :: = SEQUENCE OF NamePartPattern
```

A **Name**, a **NamePattern** and a **NameSpecification** are identical, except that: a **Name** cannot have any wildcard characters; a **NamePattern** may have wildcard characters only in its final name part; and a **NameSpecification** may have wildcard characters in any or all of its name parts. Name patterns are used as arguments in may operations and this usually means that the operation is performed on the first object found that matches the pattern (see clauses 2.5 and 3.4.1). Name specifications have a special use in membership lists, where membership is tested against the specification as if it was a pattern (see the **IsMember** operation).

3.2.1.4 Names of Leaf Vertices

```
LeafName :: = Name

LeafNamePattern :: = NamePattern
```

3.2.1.5 Names of Intermediate Vertices

```
NonLeafName :: = Name

NonLeafNamePattern :: = NamePattern
```

3.2.2 Data Types for Properties

A Property consists of its Property Type and its Property Value.

There are two kinds of property: item and group. Whether a property is an item or group is determined at the time it is added to a leaf entity. (A property of a given property type could be of one kind in one leaf entity, and of a different kind in another, although doing this is strongly discouraged.)

The value of an item property is not inspected by the Directory System. The value of a group property is a set of patterns. where wildcards are allowed in any of their name parts (including the degenerate case where the pattern contains no wildcard characters). Group property values are manipulated by the Directory System.

3.2.2.1 Property Types

```
PropertyType :: = INTEGER
```

3.2.2.2 Property Value

```
PropertyValue :: = CHOICE {
    item [0] IMPLICIT OCTETSTRING,
    group [1] IMPLICIT SET OF NameSpecification }
```

3.2.2.3 Property

```
Property :: = SEQUENCE { PropertyType, PropertyValue }
```

3.2.3 Filters

Some search operations screen a set of elements against one or more of the following criteria:

I) having a name part with a specified type;

II) having a name part with a specified type and value;

III) having a property with a specified type;

IV) having a property with a specified type and value;

V) combinations of the above by means of the Booleans **and**, **or**, **not**.

The basic constituent of a filter is the filter component, which models numbers I-IV above.

```
FilterComponent :: = CHOICE {
                        [0]          NamePartType,
                        [1] IMPLICIT NamePart,
                        [2]          PropertyType,
                        [3] IMPLICIT Property }
```

The following recursive definition uses prefix notation to model filters as boolean combinations. In a filter type, the and and or parameters must comprise, where they occur, at least two elements in the sequence.

```
Filter :: = CHOICE {
    [0] FilterComponent,
    and [1] IMPLICIT SEQUENCE OF Filter,--shall contain at least two elements--
    or [2] IMPLICIT SEQUENCE OF Filter, --shall contain at least two elements--
    not [3] Filter }
```

3.2.4 Authentication Parameters

All operations of the Directory Access Service carry a parameter, which is usually optional, which identifies the user and may also include some means, such as a password, to authenticate this identity. This parameter is defined as a CHOICE type to allow the support of various levels of authentication:

```
Authenticator :: = CHOICE {

                    [0] IMPLICIT UserName,

                    [1] IMPLICIT SimpleCredentials }

UserName :: = LeafName

UserPassword :: = OCTETSTRING

SimpleCredentials :: = SET {

                        [0] IMPLICIT UserName,

                        [1] IMPLICIT UserPassword }
```

Note 3
Future Technical Reports for Directory Access may define further options in the Authenticator CHOICE.

3.3 Conventions for Tag Values

In the Operation specifications of this Technical Report, certain conventions have been adopted regarding the allocation of tags to arguments and result parameters, in order to facilitate some aspects of implementation and testing. These conventions are as follows.

3.3.1 Arguments

Tag values 0-3 are used for the argument which names the entity of primary interest in this operation or its parent, as follows:

- 0 : the entity itself is named and the name is not a pattern;
- 1 : the entity itself is named and the name may be a pattern;
- 2 : the parent is named and the name is not a pattern;
- 3 : the parent is named and the name may be a pattern.

Tag values 4-6 are used for arguments which are Property Types, as follows:

- 4 : if the Property is assumed not to exist before this operation;
- 5 : if the Property is assumed to exist before this operation;
- 6 : for a second Property Type in some perations.

Tag value 7 is used for a Filter.

Tag value 8 is used for an Authenticator.

Tag values 9-15 are reserved.

Tag values from 16 upwards are allocated serially on an operation-specific basis.

3.3.2 Result Parameters

Tag value 0 is used for the result parameter which returns a Distinguished Name.

Tag value 1 is used for the result parameter which returns a Name that matches a Pattern argument.

Tag values 2-15 are reserved.

Tag values from 16 upwards are allocated serially on an operation-specific basis.

3.4 Operations Applicable to All Entities

The operations in this clause may be applied both to Leaf Vertices and to Intermediate Vertices.

3.4.1 Read Distinguished Name

Operation **readDistinguishedName** returns a distinguished name given a name or name pattern. This procedure is useful when a user knows a partial spelling of a name, or an alias (or both) and wishes to find the distinguished name of the entity. If the pattern contains wildcard characters, the DIB is searched for a match. The searching stops with the first name (alias or distinguished n ame) that matches. If the result of the search is an alias, it is de-referenced.

```
readDistinguishedName OPERATION
   ARGUMENT SET {
      name [1] IMPLICIT NamePattern,
      agent [8] Authenticator OPTIONAL },
   RESULT SET {
      distinguishedName [0] IMPLICIT Name,
      match [1] IMPLICIT Name OPTIONAL }
   ERRORS { authenticationError,
      serviceError, accessControlError, nameError }
   ::= 4
```

Arguments: **name** is the pattern which is to be looked up (the last name part may contain wildcard characters). **agent** contains the client's credentials and authentication data.

Results: **match** is the name or alias that first matched the search. **distinguishedName** is the distinguished entity that corresponds to name.

3.4.2 Read Aliases of a Name

Operation **readAliasesOf** lists the aliases that point to an entity. The list of aliases produced applies to the first name that matched the pattern. If that name is an alias, it is de-referenced first.

```
readAliasesOf OPERATION
   ARGUMENT SET {
      pattern [1] IMPLICIT NamePattern,
      agent [8] Authenticator OPTIONAL }
```

```
RESULT SET {
    distinguishedName [0] IMPLICIT Name,
    match [1] IMPLICIT Name OPTIONAL
    aliases [16] IMPLICIT SET OF Name }
ERRORS { authenticationError,
    serviceError, accessControlError, nameError }
:: = 9
```

Arguments: **pattern** defines the string that candidate vertex-names must match; the first match found will determine the entity to be listed, agent contains the user's credentials and authentication data.

Results: **match** is the name or alias that first matched the pattern. **distinguishedName** is the distinguished name of the entity to which the aliases point. **aliases** returns the data, i.e., a set of alias names (with no wildcard characters in them).

3.4.3 Create Alias

Operation **createAlias** adds a new alias to the DIB and points it at the specified entity. If the name supplied is itself an alias, it is de-referenced, and the newly created alias is pointed at the distinguished entity.

```
createAlias OPERATION
    ARGUMENT SET {
        alias [0] IMPLICIT Name,
        sameAs [16] IMPLICIT Name,
        agent [8] Authenticator OPTIONAL }
    RESULT SET {
        distinguishedName [0] IMPLICIT Name }
    ERRORS { authenticationError,
        serviceError, updateError, accessControlError, nameError }
    :: = 10
```

Arguments: **alias** is the name of the alias to be created. **sameAs** is the name of, or an alias of, the entity to which the alias is to point. **agent** contains the user's credentials and authentication data.

Results: **distinguishedName** is the distinguished name of the entity to which **alias** points.

3.4.4 Delete Alias

Operation **deleteAlias** deletes an existing alias from the DIB and returns the distinguished name to which the alias was pointing. The specified name must be an alias and not a distinguished name.

```
deleteAlias OPERATION
  ARGUMENT SET {
    alias [0] IMPLICIT Name,
    agent [8] Authenticator OPTIONAL }
  RESULT SET {
    distinguishedName [0] IMPLICIT Name }
  ERRORS { authenticationError,
    serviceError, updateError, accessControlError, nameError }
  ::= 11
```

Arguments: **alias** is the name of the alias to be deleted; it must be an alias and not a distinguished name. **agent** contains the user's credentials and authentication data.

Results: **distinguishedName** is the distinguished name that was pointed to by **alias**.

3.5 Operations on Non-Leaf Vertices

The operations defined in this clause are applicable to Non-Leaf Vertices but not to Leaf Vertices.

Note 4
The operations on Non-Leaf Vertices defined in this Technical Report are for enquiry only. It is anticipated that future Technical Reports will also define administrative operations on Non-Leaf Vertices, for example to create and delete them. Until such Technical Reports are defined, administration of Non-Leaf Vertices must be achieved by non-standard means defined by each DSA implementation.

3.5.1 Read Children

Operation **readChildren** lists the name parts (type and value) emanating from a specified non-leaf vertex. This vertex can also be specified by means of a pattern, in which case the operation will be performed on the first match found. The operation will list either all the name part values or those of a specified type that match a given pattern. If the second argument, pattern, is not present, the operation returns the list of all name parts of children; if present, it returns all the values of name parts that match the pattern, in particular, use of "*" obtains all values of a certain name part type.

```
readChildren OPERATION
  ARGUMENT SET {
    parent [3] IMPLICIT NonLeafNamePattern,
    pattern [16] IMPLICIT NamePartPattern OPTIONAL,
    filter [7] Filter OPTIONAL,
    agent [8] Authenticator OPTIONAL }
  RESULT SET {
    distinguishedName [0] IMPLICIT NonLeafName,
    match [1] IMPLICIT NonLeafName OPTIONAL,
    children [16] IMPLICIT SET OF NamePart }
  ERRORS { authenticationError, serviceError, accessControlError, nameError }
  ::= 5
```

Arguments: **parent** defines the parent vertex, which must be a non-leaf. **pattern**, if present, defines the name parts of the children to be listed; if it is not present, all children must be listed. **filter** further specifies the children to be listed.

Results: **match** is the name or alias that first matched the parent. **distinguishedName** is the distinguished name of the parent. **children** returns the list of matching name parts.

3.5.2 Read Children Types

Operation **readChildrenTypes** list all the different types of name parts emanating from a vertex. The search for a parent vertex stops with the first match found.

```
readChildrenTypes OPERATION
   ARGUMENT SET {
      parent [3] IMPLICIT NonLeafNamePattern,
      agent [8] Authenticator OPTIONAL }
   RESULT SET {
      distinguishedName [0] IMPLICIT NonLeafName,
      match [1] IMPLICIT NonLeafName OPTIONAL
      types [16] IMPLICIT SET OF NamePartType }
   ERRORS { authenticationError, serviceError, accessControlError, nameError }
   :: = 6
```

Argument: **parent** is a name or pattern specifying a non-leaf vertex.

Results: **match** is the name or alias that first matched the parent. **distinguishedName** is the distinguished name of the non-leaf vertex specified by parent. **types** lists the different types of name components found directly underneath that vertex.

3.5.3 Read Alias Children

Operation **readAliasChildren** lists only the name parts emanating from a particular vertex which are aliases.

```
readAliasChildren OPERATION
   ARGUMENT SET {
      parent [3] IMPLICIT NonLeafNamePattern,
      pattern [16] IMPLICIT NamePartPattern OPTIONAL,
      filter [7] Filter OPTIONAL,
      agent [8] Authenticator OPTIONAL }
   RESULT SET {
      distinguishedName [0] IMPLICIT NonLeafName,
      match [1] IMPLICIT NonLeafName OPTIONAL,
      children [16] IMPLICIT SET OF NamePart }
   ERRORS {
      authenticationError, serviceError, accessControlError, nameError }
   :: = 8
```

Arguments: **parent** defines the parent vertex, which must be a non-leaf. **pattern** specifies the aliases which are to be listed. **filter** further specifies the children to be listed. **agent** contains the client's credentials and authentication data.

Results: **match** is the name or alias of the parent vertex that first matched the pattern. **distinguishedName** returns the name of the parent vertex. **children** returns the data, i.e., set of values of name parts.

3.6 Operations on Leaf Vertices

3.6.1 Create Leaf Entity

Operation **createLeafEntity** creates a new leaf entity in the DIB. This is created with an initial list of properties. The arguments specify the name or an alias of the parent entity, and a name part to form the last part of the distinguished name of the new leaf entity. The distinguished name is returned.

```
createLeafEntity OPERATION
  ARGUMENT SET {
    parent [2] IMPLICIT NonLeafName,
    distinguishedArcLabel [16] IMPLICIT NamePart,
    properties [17] IMPLICIT SET OF Property,
    agent [8] Authenticator OPTIONAL }
  RESULT SET {
    distinguishedName [0] IMPLICIT LeafName }
  ERRORS { authenticationError,
    serviceError, propertyError, updateError, accessControlError, nameError }
  ::= 2
```

Arguments: **parent** specifies the name or an alias of the parent vertex, which must be a non-leaf entity. **distinguishedArcLabel** specifies the last name part of the distinguished name of the new entity. **properties** specifies the initial set of properties. **agent** contains the client's credentials and authentication data.

Results: **distinguishedName** is the distinguished name of the newly created leaf entity.

3.6.2 Delete Leaf Entity

Operation **deleteLeafEntity** deletes a leaf entity from the DIB, including all its Properties. If the leaf name is an alias, it is first de-referenced. As a result of this operation, all aliases that point to the specified leaf entity are also deleted.

```
deleteLeafEntity OPERATION
  ARGUMENT SET {
    name [0] IMPLICIT LeafName,
    agent [8] Authenticator OPTIONAL }
  RESULT SET { }
  ERRORS { authenticationError,
    serviceError, updateError, accessControlError, nameError }
  ::= 3
```

Arguments: **name** is the leaf-name or alias of the leaf entity to be deleted. **agent** contains the client's credentials and authentication data.

3.7 General Property Operations

The Operations defined in this clause operate on one or more Properties of a Leaf Entity. They apply both to item and to group Properties.

3.7.1 Read Properties

Operation **readProperties** returns a list of selected properties associated with a leaf entity. The type and value of each property is returned. If the empty list of property types is given, then all properties of the entity are read.

```
readProperties OPERATION
   ARGUMENT SET {
      pattern [1] IMPLICIT LeafNamePattern,
      propertyTypes [16] IMPLICIT SET OF PropertyType,
      agent [8] Authenticator OPTIONAL }
   RESULT SET {
      distinguishedName [0] IMPLICIT LeafName,
      match [1] IMPLICIT LeafName OPTIONAL,
      properties [16] IMPLICIT SET OF Property }
   ERRORS { authenticationError,
      serviceError, propertyError, accessControlError, nameError }
   :: = 16
```

Arguments: **pattern** specifies the leaf entity. **propertyTypes** specifies the list of property types whose values are requested (the empty list means that all properties are requested). **agent** contains the client's credentials and authentication data.

Results: **match** is the name or alias that first matched pattern. **distinguishedName** is its distinguished name. **properties** is the set of properties (i.e., types and values).

3.7.2 Read Property Types

Operation **readPropertyTypes** returns the property types associated with a leaf entity whose name matches the specified name or pattern.

```
readPropertyTypes OPERATION
   ARGUMENT SET {
      pattern [1] IMPLICIT LeafNamePattern,
      agent [8] Authenticator OPTIONAL }
   RESULT SET {
      distinguishedName [0] IMPLICIT LeafName,
      match [1] IMPLICIT LeafName OPTIONAL,
      propertyTypes [16] IMPLICIT SET OF PropertyType }
   ERRORS {
      authenticationError, serviceError, accessControlError, nameError }
   :: = 15
```

Arguments: **pattern** identifies the leaf entity whose properties will be returned. **agent** contains the client's credentials and authentication data.

Results: **match** is the name or alias that first matched pattern. **distinguishedName** is its distinguished name. **propertyTypes** is the list of the property types assigned to **distinguishedName**.

3.7.3 Add Property

Operation **addProperty** adds a new property to a leaf entity and gives that property an initial value. If an attempt is made to add a property of the same property type as an existing property in this entry, the operation fails returning **updateError** with a value of **noChange**.

```
addProperty OPERATION
   ARGUMENT SET {
      name [0] IMPLICIT LeafName,
      newProperty [16] IMPLICIT Property,
      agent [8] Authenticator OPTIONAL }
   RESULT SET {
      distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
      serviceError, propertyError, updateError, accessControlError, nameError }
   ::= 13
```

Arguments: **name** is a name of the leaf entity to which the property will be added. **newProperty** specifies the property (type and value) to be added. **agent** contains the client's credentials and authentication data.

Results: **distinguishedName** is the distinguished name to whose leaf entity the property was added.

3.7.4 Change Property

Operation **changeProperty** is the same as **AddProperty** except that it changes an already existing property.

```
changeProperty OPERATION
   ARGUMENT SET {
      name [0] IMPLICIT LeafName,
      property [16] IMPLICIT Property,
      agent [8] Authenticator OPTIONAL }
   RESULT SET {
      distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
      serviceError, propertyError, updateError, accessControlError, nameError }
   ::= 1
```

Arguments: **name** is a name of the leaf entity whose property is to be changed. **property** identifies the specific property (type and value) to be assigned to that property. **agent** contains the client's credentials and authentication data.

Results: **distinguishedName** is the distinguished name of the leaf entity whose property was changed.

3.7.5 Delete Property

Operation **deleteProperty** deletes the specified property (both type and value) from the leaf entity. The leaf name may be an alias. The property may be item or group.

```
deleteProperty OPERATION
   ARGUMENT SET {
```

```
    name [0] IMPLICIT LeafName,
    property [5] IMPLICIT PropertyType,
    agent [8] Authenticator OPTIONAL }
  RESULT SET {
    distinguishedName [0] IMPLICIT LeafName }
  ERRORS { authenticationError,
    serviceError, propertyError, updateError, accessControlError, nameError }
  :: = 14
```

Arguments: **name** is a leaf name of the leaf entity whose property will be deleted. **property** is the Property Type of the Property to be deleted. **agent** contains the client's credentials and authentication data.

Results: **distinguishedName** is the distinguished name from whose leaf entity the property was deleted.

3.8 Operations On the Membership of Group Properties

The operations in this clause apply to Group Properties only.

3.8.1 Read Group Members

The **readGroupMembers** operation retrieves the membership of a group property. It has both a recursive and a non-recursive mode of operation.

```
readGroupMembers OPERATION
  ARGUMENT SET {
    name [1] IMPLICIT LeafNamePattern,
    primary [5] IMPLICIT propertyType,
    secondary [6] IMPLICIT propertyType OPTIONAL,
    filter [7] Filter OPTIONAL,
    agent [8] Authenticator OPTIONAL }
  RESULT SET {
    distinguishedName [0] IMPLICIT Name,
    match [1] IMPICIT LeafName OPTIONAL,
    members [16] IMPLICIT SET OF NameSpecification }
  ERRORS { authenticationError,
    serviceError, propertyError, accessControlError, nameError }
  :: = 18
```

Arguments: **name** is the name pattern specifying the desired leaf entity. **primary** is the property type that specifies the desired group property. **secondary** indicates whether the recursive mode of operation is required: if **secondary** is omitted, the result is the set of members of the **primary** Property of the object; if **secondary** is present, then the operation returns the same set plus all members of the secondary property of this set, and so on. Duplications are removed from the returned list, and wildcards are not substituted, so the operation is guaranteed to terminate eventually.

Results: **match** is the name or alias that first matched **name**. **distinguishedName** is the distinguished name of the leaf that holds the group. **members** list the members without repetitions.

3.8.2 Is Member

Operation **isMember** determines if a name is a member of some group property. This procedure has two modes of operation. In the normal mode, it will examine only the members of the property specified. In the group mode, it will extend the search to the membership of groups listed in the original group property.

In normal mode, the specified name is compared with each entry in a group property of some leaf entity. If the name is found, comparison stops. In any event, the procedure indicates whether or not the name was found.

In group mode, the search is extended to the group properties of leaf entities whose names were in the initial group property. Thus, for the original object, the search proceeds as in the normal mode. However, if the name is not found in this property, the following procedure is invoked.

I) Each member of the property is assumed to name a leaf entity in the database. That entity is examined to see if it has a specified secondary (group) property. If the member doesn't name a leaf entity or the leaf entity doesn't have the required property, the next member of the group is examined.

II) Otherwise, the members of that secondary property are examined for the original name. If the sought-for name is not a member of this group, each member of this group is examined according to this algorithm--that is, each is examined to see if it has the secondary property.

III) This process continues recursively (but always using the same secondary property) until either a match is found, or all members referenced by the original list, either directly or indirectly, have been examined. Consequently, the procedure will always terminate.

In either mode, the name comparison proceeds in a manner somewhat the opposite of the other Directory Service operations. First, the name whose membership is being checked may have wildcard characters in any of its component parts. However, the wildcards will not be interpreted as such, but, rather, act as just another character. Second, the members of the group property are considered to be "super" patterns; wildcard characters can occur in any or all of the components. The comparison process treats the member name as a pattern when comparing with the sought-for name. It is important to note that in a group mode operation, when a pattern is a member of a group, it does not define a set of entities whose properties are to be searched; such a pattern cannot define further entities to be investigated. The pattern attribute of a member is employed only for purposes of pattern matching with the sought-for name.

```
isMember OPERATION
  ARGUMENT SET {
    memberOf [1] IMPLICIT LeafNamePattern,
    primary [5] IMPLICIT PropertyType ,
    secondary [6] IMPLICIT PropertyType OPTIONAL,
    name [16] IMPLICIT NameSpecification,
    agent [8] Authenticator OPTIONAL }
  RESULT SET {
    distinguishedName [0] IMPLICIT LeafName,
    match [1] IMPLICIT LeafName OPTIONAL,
    isMember [16] IMPLICIT BOOLEAN }
```

```
ERRORS { authenticationError,
   serviceError, propertyError, accessControlError, nameError }
:: = 20
```

Arguments: **memberOf** is the name of the leaf entity whose specified group property is to be examined. **primary** indicates the group property to be searched. **name** is the name whose membership is being determined. **secondary** indicates whether or not the group mode should be invoked: if it is absent, only the members of **property** are examined for **name**; if it is present, then, for each member of **property**, if there is a leaf entity associated with the member, in this leaf entity the property identified by **secondary** is examined for **name**, and for each member of that group property the search is extended to the same group property, and so forth until every leaf entity referenced has been examined, or until **name** is found. **agent** contains the user's credentials and authentication data.

Results: **isMember** is **TRUE** if **name** was found, **FALSE** otherwise. **distinguishedName** is the distinguished name designated by **memberOf**. **match** is the name or alias that first matched **memberOf**.

Example: Suppose a user wants to determine if "Charlie" is the owner of "object". "object" has a group property, called "owners", that lists its owners. Suppose also that there is a kind of leaf entity called a "name list" which has a property called "subordinates" which contains the names of other "name list"s. Assume that the names of "name list" leaf entities can appear as owners of "object"s. To ascertain if "Charlie" is an owner of the object, a client would make the following (stylized) call:

```
isMember {
   memberOf :: = "object",
   primary :: = "owners",
   secondary :: = "subordinates",
   name :: = "Charlie",
   agent :: = ... }
```

What happens is this: the members of the "owners" property of "object" are examined to see if any one of them is "Charlie". If "Charlie" is not a member of "owners", then for each member of "owners" which is a "name list" and, therefore, has a property of "subordinates", that property is examined to see if "Charlie" is a member. If not, then each member which is itself a "name list" leaf entity has its "subordinates" property checked for "Charlie". This process goes on recursively until the entire set of names, the closed set, pointed to by the "owners" property has been examined. If "Charlie" is found in one of the group properties, searching stops and an indication of found is returned.

3.8.3 Add Group Member

The **addGroupMember** operation adds a member to a group property.

```
addGroupMember OPERATION
   ARGUMENT SET {
      name [1] IMPLICIT LeafName,
      group [5] IMPLICIT PropertyType,
      member [16] IMPLICIT NameSpecification,
      agent [8] Authenticator OPTIONAL }
```

```
    RESULT SET {
        distinguishedName [0] IMPLICIT Name }
    ERRORS { authenticationError,
        serviceError, propertyError, updateError, accessControlError, nameError}
    ::= 12
```

Arguments: **name** is the name specifying the desired leaf entity. **group** is the property type that specifies the desired group property, to which the member is to be added. **member** is the name of the member to be added; wildcards are allowed in any of its name parts.

Results: **distinguished** is the distinguished name of the leaf to which the member was added.

3.8.4 Add Self

The **addSelf** operation adds the user as a member to a group property.

```
addSelf OPERATION
    ARGUMENT SET {
        name [1] IMPLICIT LeafName,
        group [5] IMPLICIT PropertyType,
        agent [8] Authenticator }
    RESULT SET {
        distinguishedName [0] IMPLICIT LeafName }
    ERRORS { authenticationError,
        propertyError, accessControlError, serviceError, updateError, nameError}
    ::= 17
```

Arguments: **name** is the name specifying the desired leaf entity. **group** is the property type that specifies the desired group property, to whose membership the user is to be added. The name of the user is taken from **agent**.

Results: **distinguished** is the distinguished name of the leaf to whose membership the user was added.

3.8.5 Delete Group Member

The **deleteGroupMember** operation deletes a member of a group property.

```
deleteGroupMember OPERATION
    ARGUMENT SET {
        name [1] IMPLICIT LeafName,
        group [5] IMPLICIT PropertyType,
        member [16] IMPLICIT NameSpecification,
        agent [8] Authenticator OPTIONAL }
    RESULT SET {
        distinguishedName [0] IMPLICIT Name }
    ERRORS { authenticationError,
        serviceError, propertyError, updateError, accessControlError, nameError }
    ::= 7
```

Arguments: **name** is the name specifying the desired leaf entity. **group** is the property type that specifies the desired group property, from which the member is to be deleted.

member is the name of the member to be deleted; wildcards are allowed in any of its name parts, but there is no search implied: only an exact replica of the string **member** will be deleted from the group.

Results: **distinguishedName** is the distinguished name of the leaf from which the member was added.

3.8.6 Delete Self

The **deleteSelf** operation deletes the user from the membership of a group property.

```
deleteSelf OPERATION
   ARGUMENT SET {
      name [1] IMPLICIT LeafName,
      group [5] IMPLICIT PropertyType,
      agent [8] Authenticator }
   RESULT SET {
      distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
      serviceError, propertyError,updateError, accessControlError, nameError }
   ::= 19
```

Arguments: **name** is the name specifying the desired leaf entity. **group** is the propertyType that specifies the desired group property, from whose membership the user is to be deleted. The name of the user is taken from **agent**.

Results: **distinguishedName** is the distinguished name of the leaf from whose membership the user was deleted.

3.9 Remote Errors

3.9.1 General

When a remote operation completes successfully, it returns results as specified in the definition of the operation. However, conditions can arise before or during execution of the operation that make successful completion of the request impossible. For example, the user may have specified incorrect arguments in a remote operation call, or some required resource may be unavailable.

When such conditions occur, an error is reported to communicate to the user the nature of the problem. Each error encompasses an entire class of possible conditions, and the specific problem is further described by the arguments of the error. For example, **serviceError** indicates that the DSA could not complete the requested operation. The particular problem is specified by the argument to **serviceError**, which is of type **ServiceProblem**.

All Directory Access Service errors are defined below. Each error definition includes a declaration of the error in CCITT Rec. X.409/410 standard notation, and a description of its arguments.

3.9.2 Argument Resolution

Many of the Directory Access Service operations have two arguments of the same type. To avoid duplication of essentially identical errors or error types, some errors have an argument indicating which of the two identically-typed arguments caused the error. This resolving argument in the error definition is defined as follows:

```
WhichArgument :: = INTEGER {
   first(1),
   second(2) }
```

first and **second** correspond to the lexical order of the arguments in the procedure definition. If the procedure which reported the error only has one argument of the indicated type, **first** is always reported.

3.9.3 Authentication Errors

A DSA may refuse to perform an operation if the client's authenticator is invalid. The refusal is reported by the error:

```
authenticationError ERROR
   PARAMETER SET {
      problem [0] IMPLICIT AuthenticationProblem }
   :: = 6

AuthenticationProblem :: = INTEGER {
   userNameInvalid (50),
   userPasswordInvalid (51),
   inappropriateCredentials (52) }
```

userNameInvalid indicates that the purported user name was not a valid leaf name.

userPasswordInvalid indicates that the purported user password did not match the value of the user password as expected by the Directory System.

inappropriateCredentials indicates that the user credentials specified in an authentication operation were inappropriate for the authentication policy in effect.

3.9.4 Service Errors

Service errors model a number of circumstances that prevent a DSA from satisfying a service request. These circumstances include, but are not limited to, the following: the DSA was too busy or could not allocate resources; the DSA does not hold the desired data and returns a "hint;" the DSA tried to obtain the data but the other DSA that holds it could not be contacted.

```
serviceError ERROR
   PARAMETER SET {
      problem [0] IMPLICIT ServiceProblem,
      hint [1] IMPLICIT LeafName OPTIONAL }
   :: = 5

ServiceProblem :: = INTEGER {
   dSATooBusy (40),
   wrongDSA (42),
   chainingFailed (43),
   unsupportedFilter (48),
   recursionNotImplemented (49) }
```

The problem parameter indicates the reason for the inability of the DSA to provide the requested service.

dSATooBusy means that the service is processing so many other user requests that it presently cannot handle the operation. The client should pause a short while and retry the operation, or try the operation on some other DSA.

wrongDSA means that the DSA does not hold the desired information in its local information base. This response shall be accompanied by the **hint** parameter in order to direct the user to another DSA (see section 3.9.4.1).

chainingFailed means that the service could not complete the operation because another Directory Service had to be accessed and was found to be unavailable.

unsupportedFilter means that the user specified a filter which the DSA does not support for this operation.

recursionNotImplemented means that this DSA does not support recursive group expansion.

3.9.4.1 The Use of "Hint"

A DSA may return a value of the optional parameter **hint** along with any of the values of the parameter ServiceProblem, because this information may be cached or otherwise used by some DUAs. However, when a DSA returns the **ServiceProblem** parameter with the value **wrongDSA**, the **hint** parameter must be also present.

The value of **hint** is used in order to initiate a series of queries to the original DSA to ascertain the addresses of other DSAs that may be able to process the client's original request. The following four conditions must apply:

I) **hint** is a **LeafName**;

II) **hint** points to a group property, whose property type is "members";

III) the members of this group property are distinguished names of DSAs;

IV) the DSA that returned the **hint** holds in its local database the group property (to which **hint** points), as well as the leaf entity for each of the DSA names specified in this group property.

The entry for a DSA holds, among other things, the addresses of the DSA (like other Application Processes, a DSA may be reachable through more than one address).

Thus, after receiving a hint from a DSA, the DUA can perform the following sequence of operations:

I) Call the last contacted DSA with the request **readGroupMembers { LeafName : hint, Property :** "members" }. This step yields a list of distinguished names of DSAs.

II) Select one or more of the names returned and obtain the list of addresses for each one of these selected names. This step generates a list of addresses of DSAs.

III) Now select one among these addresses obtained in the last step, and send to that address the original service requested (this step can be repeated with another address in case of failure to make contact; it is even possible to back up to Step II to try some untried names, if there are any). When successful contact is established with any of these addresses, if the newly contacted DSA satisfies the request the

algorithm terminates. Or else, the DSA returns a **hint** and the DUA goes back to Step I.

The four conditions above guarantee that Steps I) and II) will never produce a **wrongDSA** error. This ECMA Technical Report does not specify the method by which the implied selections in Steps II) and III) are performed by the DUA. Some clients may wish to select all names; others may prefer to go back and forth between Steps III) and II) until all names are exhausted. In Step III), some DUAs will select an address at random, while others may have a way of determining which addresses are "nearer" to them.

Any implementations of this ECMA Technical Report must guarantee that the process will converge regardless of how DUAs perform the selections in Steps II) and III); and each implementation must specify how many times a DUA may be expected, in the worst case, to have to perform Step I).

3.9.5 Property Errors

The error **propertyError** is reported when an error associated with the Property argument of an operation prevents the operation from successfully completing.

```
propertyError ERROR
   PARAMETER SET {
      problem [0] IMPLICIT PropertyProblem,
      which [2] IMPLICIT WhichArgument,
      distinguishedName [3] IMPLICIT LeafName }
   ::= 3

PropertyProblem ::= INTEGER{
   missing(20),
   wrongType(21) }
```

The first argument indicates the nature of the problem. A value of **missing** means that the designated leaf entity does not have any property with the specified **PropertyID**. **wrongType** indicates that the leaf entity has such a property, but that its type is item when a group type was required, or vice versa.

The second argument, **distinguishedName**, indicates the distinguished name of the leaf entity to which the **propertyError** refers. **distinguishedName** is useful if the client specified an alias or pattern as the name of the entity.

3.9.6 Update Errors

The **updateError** can be reported by any operation that can modify (add to, delete from, or otherwise change) the DIB.

```
updateError ERROR
   PARAMETER SET {
      problem [0] IMPLICIT UpdateProblem,
      found [1] IMPLICIT BOOLEAN,
      which [2] IMPLICIT WhichArgument,
      distinguishedName [3] IMPLICIT LeafName }
   ::= 4
```

```
UpdateProblem :: = INTEGER{
   noChange(30),
   entryOverflow (32),
   databaseOverflow(33) }
```

The **problem** argument indicates why the update failed. The following values are specified.

noChange indicates that the operation would not change the DIB. This can occur, for example, with a request to add a property that already exists, or a request to change or delete something that does not exist.

entryOverflow means that too much data is associated with the leaf entity being modified. The overflow can occur for two reasons: the user tried to assign to the leaf entity more properties than the implementation can handle; or the total storage associated with a leaf entity is larger than the DSA can handle. The amount of data that can be associated with a leaf entity is implementation dependent.

databaseOverflow means that there is no room on the DSA in which to put the requested update. The size of a DSA local information base is implementation dependent.

The argument **found** indicates the existence of the leaf entity or property indicated by **which**. If **found** is **TRUE**, the leaf entity or property exists in the DIB; if **found** is **FALSE**, it does not. **distinguishedName** indicates the distinguished name of the leaf entity to which **updateError** refers. It is useful if an alias or pattern was specified.

3.9.7 Access Control Errors

If the user does not have sufficient rights to invoke the desired operation, the **accessControlError** error is returned. (This situation may occur due to private or local access control policies in effect at a DSA).

```
accessControlError ERROR
   PARAMETER SET {
      problem [0] IMPLICIT AccessControlProblem }
   :: = 1

AccessControlProblem :: = INTEGER {
   accessRightsInsufficient (1) }
```

The problem parameter describes the problem in greatr detail. Other Technical Reports, or future versions of this Technical Report, may specify some further value of this parameter.

3.9.8 Name Error

The **nameError** is reported when the name verification process is unable to completely resolve the name part list to a vertex name of the appropriate type.

```
nameError ERROR
   PARAMETER SET {
      problem [0] IMPLICIT NameProblem,
      which [2] IMPLICIT WhichArgument }
   :: = 2
```

```
NameProblem :: = INTEGER {
    overspecified (1),
    erroneous (3),
    underspecified (4),
    illegalNonLeaf (5) }
```

overspecified means that an initial sequence of the name parts of the purported name is a valid **LeafName** of an existing entry, while the complete purported name is not valid.

erroneous means that the name is invalid or non-existent.

underspecified means that this is a valid **NonLeafName** of an existing non-leaf entity, while a **LeafName** was required for this operation.

illegalNonLeaf means that the name designates a leaf entity while a **NonLeafName** was required for this operation.

3.10 Mapping onto Underlying Services

The Operation specifications and Error specifications shall be encoded as Directory Access Protocol Data Units (DAPDU) by means of the application of the CCITT Rec. X.409 encoding rules to the CCITT Rec. X.409/X.410 notation used in Section 3 and Appendix D of this ECMA Technical Report. These Directory Access Protocol Data Units shall be communicated between the DUA and the DSA, using the Remote Operations Service defined in ECMA TR/31, Section 5, in the following manner:

- All Operations of the Directory Access Protocol are Operation Class 2 (Asynchronous with Result or Error). (Thus any number of Operations may be outstanding at one instant.)

- When it requires to invoke any operation, a DUA shall send the Invoke DAPDU for that operation by means of the RO-INVOKE Service Element.

- On receiving an Invoke DAPDU for any operation, a DSA shall take action appropriate to the requested operation and then shall respond in one and only one of the following ways:

  : send a Return Result DAPDU by means of RO-RESULT;

  : send an Error DAPDU by means of RO-ERROR;

  : send a Reject DAPDU by means of RO-REJECT-U.

- A DUA shall be capable of receiving all the above responses, and also of receiving RO-REJECT-P in response to an Invoke DAPDU.

- The response DAPDU (Return Result, Error or Reject) shall be communicated on the same ROS-association as the Invoke DAPDU.

- The ROS-association shall be created by the DUA and shall use RO-BEGIN parameters as follows:

  : "reference" is the Presentation Service Access Point address of the DSA.

  : "application protocol" is "D1" to denote Directory Access Protocol.

- No use shall be made of the concept of Operation Priority.

4. CONFORMANCE REQUIREMENTS

4.1 General

This section describes the conformance requirements for the Directory Access Service and Protocol as defined in Section 3 and Appendix D of this Technical Report.

An implementation of the Directory Access Protocol shall use the Remote Operations Service in the manner specified in clause 3.10 of this Technical Report, and in conformance to Section 5 of ECMA TR/31.

Support of the Directory Access Protocol has the following conformance levels:

- Systems embodying a DUA:

  Any combination of Operations (service elements) is allowed as long as all corresponding Remote Errors can be handled.

- Systems embodying a DSA:

  Shall be able to perform all Operations (service elements) as specified in Section 3 of this Technical Report, except that the following features are optional:

- Authentication parameters must be accepted by the DSA, but an authentication process may not be present, in which case no restrictions are applied.

- Recursive search on Members of a Group Property may not be supported. In this case, the **secondary** parameter in the Operations **readGroupMembers** and **isMembers** causes the operation to be refused with a **RecursionNotImplemented** error.

- Filters may not be supported or may be only partially supported by the DSA.

- The return of the **match** result parameter by the DSA may not be supported.

4.2 Equipment

The conformance requirement is for equipment which consists of hardware and/or software claimed to be in conformance with this Technical Report. The equipment may also have other purposes.

4.3 Protocol Subsets

No protocol subset is defined by this Technical Report.

4.4 Additional Protocols

In addition to the protocols defined in this Technical Report, the equipment may also implement other Application Layer Protocols. Such additional provisions are themselves not in conformance with this Technical Report, but do not prejudice conformance with this Technical Report provided that they are separate and do not prevent use of the protocols defined within this Technical Report.

4.5 Conformance to the Directory Access Protocol

The subject equipment:

- Shall accept all correct DAPDUs received.

- Shall generate only correct DAPDUs.

The conformance statement shall specify:

- In the case of a DUA Entity:

  Which operations the equipment supports.

- In the case of a DSA Entity:

  Which optional features, if any, the equipment supports.

APPENDIX A

BRIEF DESCRIPTION OF THE REFERENCE MODEL OF OPEN SYSTEMS INTERCONNECTION

A.1 Introduction and Scope

This Appendix is not part of the Technical Report.

This Appendix provides a brief description of the Reference Model of Open Systems Interconnection.

A.2 General Description

A.2.1 Introduction

The Reference Model of Open Systems Interconnection provides a common basis for the co-ordination of the development of new standards for the interconnection of systems and also allows existing standards to be placed within a common framework. The model is concerned with systems comprising terminals, computers and associated devices and the means for transferring information between these systems.

A.2.2 Overall Perspective

The model does not imply any particular systems implementation, technology or means of interconnection, but rather refers to the mutual recognition and support of the standardized information exchange procedures.

A.2.3 The Open Systems Interconnection Environment

Open Systems Interconnection is not only concerned with the transfer of information between systems (i.e., with communication), but also with the capability of these systems to interwork to achieve a common (distributed) task. The objective of Open Systems Interconnection is to define a set of standards which allow interconnected systems to co-operate.

The Reference Model of Open Systems Interconnection recognizes three basic constituents (see Figure A.1):

- application processes within an OSI environment;
- connections which permit information exchange;
- the systems themselves.

Note A.1
The application processes may be manual, computer or physical processes.

A.2.4 Management Aspects

Within the Open Systems Interconnection architecture there is a need to recognize the special problems of initiating, terminating, monitoring on-going activities and assisting in their harmonious operations as well as handling abnormal conditions. These have been collectively considered as the management aspects of the Open Systems Interconnection architecture. These concepts are essential to the operation of the interconnected open systems and therefore are included in the comprehensive description of the Reference Model.

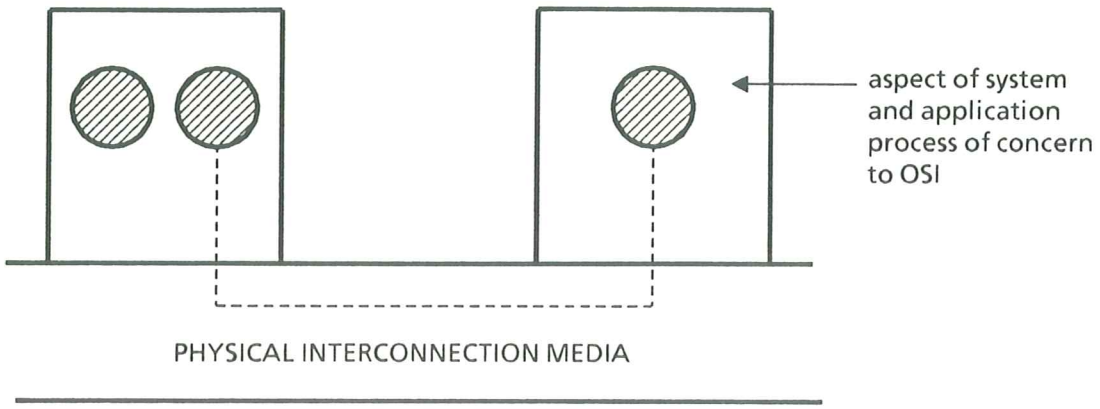

Figure A.1. General schematic diagram illustrating the basic elements of Open Systems Interconnection

A.2.5 Concepts of a Layered Architecture

The open systems architecture is structured in Layers. Each system is composed of an ordered set of sub-systems represented for convenience by Layers in a vertical sequence. Adjacent subsystems communicate through their common inter4face.

A Layer consists of all subsystems with the same rank. The operation of a layer is the sum of the co-operation between entities in that Layer. It is governed by a set of Protocols specific to that Layer.

The services of a Layer are provided to the next higher Layer, using the functions performed within the Layer and the services available from the next lower Layer.

An entity in a Layer may provide services to one or more entities in the next higher Layer and use the services of one or more entities in the next lower Layer.

A.3 The Layered Model

The seven-Layer Reference Model is illustrated in Figure A.2.

A.3.1 The Application Layer

As the highest layer in the Reference Model of Open Systems Interconnection, the Application Layer provides services to the users of the OSI environment, not to a next higher layer.

The purpose of the Application Layer is to serve as the window between communicating users of the OSI environment through which all exchange of meaningful (to the users) information occurs.

The user is represented by the application-entity to its peer.

All user specifiable parameters of each communication instance are made known to the OSI environment (and, thus, to the mechanisms implementing the OSI environment) via the Application Layer.

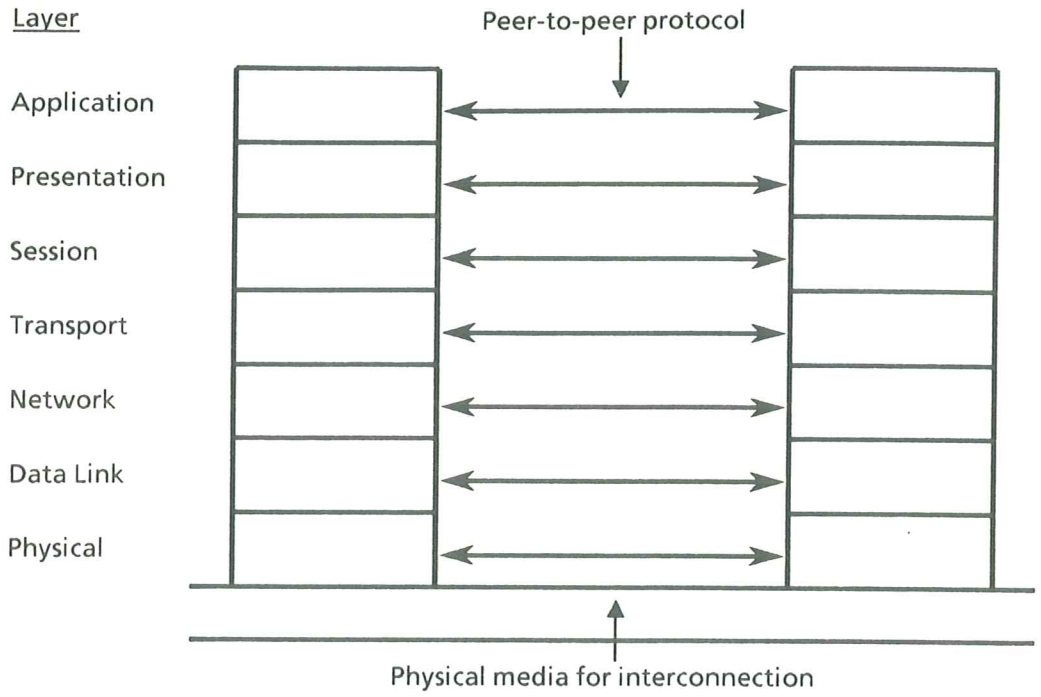

Figure A.2. The seven-layer Reference Model and peer-to-peer protocol

A.3.2 The Presentation Layer

The purpose of the Presentation Layer is to represent information to communicating application-entities in a way that preserves meaning while resolving syntax differences.

A.3.3 The Session Layer

The purpose of the Session Layer is to provide the means necessary for co-operating presentation-entities to organize and synchronize their dialogue and manage their data exchange. To do this, the Session Layer provides services to establish a session-connection between two presentation entities, and to support their orderly data exchange interactions.

To implement the transfer of data between the presentation-entities, the session-connection is mapped onto and uses a transport-connection.

A.3.4 The Transport Layer

The Transport Layer exists to provide the transport-service in association with the underlying services provided by the supporting layers.

The transport-service provides transparent transfer of data between session entities. Transport Layer relieves the transport users from any concern with the detailed way in which reliable and cost effective transfer of data is achieved.

The Transport Layer is required to optimize the use of the available communication resources to provide the performance required by each communicating transport user at minimum cost. This optimization will be achieved within the constraints imposed by considering the global demands of all concurrent transport users and the overall limit of resources available to the Transport Layer.

Since the network service provides network connections from any transport entity to any other, all protocols defined in the Trasnport Layer will have end-to-end significance, where the ends are defined as the correspondent transport-entities.

The transport functions invoked in the Transport Layer to provide requested service quality will depend on the qualify of the network service. The quality of the network service will depend on the way the network service is achieved.

A.3.5 The Network Layer

The Network Layer provides the means to establish, maintain and terminate network connections between systems containing communicating application-entities and the functional and procedural means to exchange network service data units between two transport entities over network connections.

A.3.6 The Data Link Layer

The purpose of the Data Link Layer is to provide the functional and procedural means to activate, maintain and deactivate one or more data link connections among network entities.

The objective of this layer is to detect and possibly correct errors which may occur in the Physical Layer. In addition, the Data Link Layer conveys to the Network Layer the capability to request assembly of data circuits within the Physical Layer (i.e., the capability of performing control of circuit switching).

A.3.7 The Physical Layer

The Physical Layer provides mechanical, electrical, functional and procedural characteristics to activate, maintain and deactivate physical connections for bit transmission between data link entities possibly through intermediate systems, each relaying bit transmission within the Physical Layer.

APPENDIX B

GLOSSARY

B.1 General

This Appendix is part of the Technical Report.

The terminology used in this Technical Report consists of:

- terminology defined in the Reference Model for Open Systems Interconnection,
- terminology for Directory model, services and protocol, which are defined in this Appendix.

B.2 Glossary

Alias: Alternative name for an object, which is not its distinguished name.

Authentication Error: Error indication given by the Directory System when the user's authenticator is inadequate.

Authentication Problem: Detailed problem indication on Authentication error.

Child: In the Name tree, a vertex is called a "child" of the vertex to which it is connected at the next higher level of the tree. All vertices except the Root Vertex have exactly one parent.

Directory Access Protocol: The protocol which governs communications between the Directory User Agent and the Directory Service Agent.

Directory Access Service: Service which enables a user to read and to store the properties associated with leaf-entities designated by Names.

Directory Information Base: Set of all the data managed by Directory System.

Directory Service Agent: OSI Application Process which is a part of the Directory System.

Directory System: Set of DSAs cooperating together to provide the Directory Service.

Directory System Naming Authority: Authority in charge of managing naming inside a Naming Domain.

Directory User Agent: OSI Application Process resides in the user's End System and handles the Directory Access Protocol on his behalf.

Distinguished Name: Name by which a leaf entity is created in the Directory System; any other names used to access this leaf entity are aliases.

Entry: Leaf entity (see leaf entity).

Filter: Criteria used during a search operation in the Directory Information Base.

Hint: Argument given, in case of wrong server error, by the original Directory Service Agent, in order to initiate a series of queries to the original Directory Service Agent, to ascertain the addresses of other Directory Service Agents that can process the user's original request.

Leaf Entity: A complete Name and its associated Properties. It is at the lowest level of the Name Tree.

Name: A name is a linguistic construct that singles out a particular object from among a collection of objects. It is constructed as a sequence of Name Parts.

Name Part: A part of a Name, consisting of a Name Part Type and a Name Part Value.

Name Part Type: Component of the Name Part giving the meaning of the Name Part Value (for instance, whether the Name Part Value is a Country Name or an Application Entity Title).

Name Part Value: Component of the Name Part giving the actual information associated with the Name Part.

Name Tree: The set of names of a Directory System, modelled as an oriented tree.

Naming Authority: See Directory System Naming Authority.

Naming Domain: The complete sub-tree of the Name Tree that is rooted at a given non-leaf vertex.

Non-Leaf Entity: Any vertex in the name tree, which is not on the lowest level of the tree.

Object: Anything that can be named.

Parent: In the NameTree, a vertex is called a "parent" of the vertices to which it is connected at the next lower level of the tree. A parent can have many children.

Pattern: Argument which can be used in some operations, having the structure of a name, in which partial specification may be effected by the use of wildcards.

Property: Component of a leaf entity giving information about that leaf entity. It consists of a property type and a property value.

Property Error: Error indication given by the Directory System in relation to a property argument of an operation.

Property Type: Part of property giving the meaning of the property value (for instance, whether the property value is a phone number or a telex number).

Property Value: Part of property giving the actual information included in this property (for instance, a phone number).

Root Vertex: The root (highest level) of the Name tree. This vertex is the only one that does not have a parent.

Service Error: Error reporting the inability, for various reasons, of a Directory Service Agent to satisfy a correctly formulated request.

Update Error: Error reporting the failure of an attempt to modify the Directory Information Base.

B.3 Acronyms

AP	=	Application Process
APDU	=	Application Protocol Data Unit
DAPDU	=	Directory Application Protocol Data Unit
DIB	=	Directory Information Base
DS	=	Directory System
DSA	=	Directory Service Agent
DUA	=	Directory User Agent
NA	=	Naming Authority
ND	=	Naming Domain
OSI	=	Open Systems Interconnection
PDU	=	Protocol Data Unit

(Blank page)

APPENDIX C

SUMMARY OF PRESENTATION TRANSFER SYNTAX AND PROTOCOL SPECIFICATION METHOD

C.1 Introduction and Scope

This Appendix is not part of the Technical Report.

The protocol specified by this Technical Report is based upon the transfer syntax defined in the CCITT Rec. X.409 (identical with ANS.1-ISO DP 8824) and the notation defined in the CCITT Rec. X.410, Chapter 2. This Appendix describes briefly that syntax, that notation, and some of the associated concepts.

C.2 Data Types, Data Values and Standard Notation

CCITT Rec. X.409 defines a transfer syntax for various kinds of information. Each piece of information is considered to have a type as well as a value. A data type is a class of information (for example, numeric or textual). A data value is an instance of such a class (for example, a particular number or a fragment of text). CCITT Rec. X.409 defines a number of generally useful data types from which application-specific data types are constructed in this Technical Report and in others that make use of the CCITT Rec. X.409 transfers syntax. Among the generally useful data types defined by CCITT Rec. X.409 are Integer, Octet String, Sequence and Set.

The standard notation defined in CCITT Rec. X.409 is a formal description method that allows data types relevant for an application to be specified in terms of other data types, including the generally useful data types of CCITT Rec. X.409. This notation is used in Section 3 of this Technical Report, where the Protocol Element data types are specified in terms of Sets and Sequences of more elementary data types which in turn are specified in terms of others, and finally in terms of basic data types such as Integer and Octet String.

C.3 Standard Representation

This Technical Report representation for a data type is the set of rules for encoding values of that type for transmission as a sequence of octets. The representation of a value also encodes its type and length, and is completely implied by the standard notation of the data type.

The standard representation of a data value is a data element having three components, which always appear in the following order: The Identifier designates the data type and governs the interpretation of the Contents. The Length specifies the length of the Contents. The Contents is the value of the data element, encoded as specified in CCITT Rec. X.409. The Identifier and the Length each consist of one or more octets; the Contents consists of zero or more octets.

C.3.1 Identifier

Four classes of data types are distinguished by means of the Identifier: universal, application-wide, context-specific and private-use. Universal types are generally useful, application-independent types; they are defined in CCITT Rec. X.409. Applicaiton-wide types are more specialized, being peculiar to a particular application; they are defined in this Technical Report and in others using CCITT Rec. X.409, by means of the standard notation. Context-specific types, like application-wide types, are peculiar to an application and defined using the standard notation. However, they are used only within an even more limited context - for example, that of a Set - and their identifiers are assigned so as to be distinct only within that limited context. Private-use types are reserved for private use;

the assignment of specific private-use identifiers can be accomplished by means of the standard notation but is outside the scope of CCITT Rec. X.409 or this Technical Report.

Two forms of data elements are distinguished by means of the Identifier: primitive and constructor. A primitive element is one of the Contents which is atomic. A constructor element is one of the Contents which is itself a data element, or a series of data elements. Constructor elements are thus recursively defined.

C.3.2 Length

The Length specifies the length in octets L of the Contents and is itself variable in length. It may take any of three forms: short, long and indefinite.

- The short form is used when L is less than 128.
- The long form is used when L is greater than 127.
- The indefinite form may (but need not) be used when the element is a constructor. When this form is employed, a special end-of-contents (EOC) element terminates the Contents.

C.3.3 Contents

The Contents is variable in length ancdis interpreted in a type-dependent way. If the data element is a constructor, the Contents itself comprises zero or more data elements; data elements are thus recursively defined.

C.4 Built-in Types and Defined Types

The generally useful data types defined by CCITT Rec. X.409 consist of built-in types and defined types.

Built-in types are used to construct all other data types. They include Integer, Octet String, Sequence, Set and Tagged. Integer is a primitive data type. Octet String can be either primitive or constructor. Sequence and Set are constructor data types. Identifiers for these data types are of the universal class and are specified in CCITT Rec. X.409. A Tagged data type is a data type for which the Identifier can be specified using the standard notation, as is done in Section 3 of this Technical Report.

Defined types are specified in CCITT Rec,. X.409 using the standard notation. They include Numeric String and Printable String, all of which are defined in terms of the built-in type Octet STring. They can be either primitive or constructor; their identifiers are of the universal class and are specified in CCITT Rec. X.409.

C.5 The Remote Operations Notation

Chapter 2 of CCITT Rec. X.410 uses CCITT Rec. X.409 macros to define a notation for specifying the abstract syntax of a protocol as a set of request-response pairs of APDUs. This also implicitly assumes a rule of procedure in which each response is bound to a request and each request-response pair is independent of all others. This assumption about the Protocol can usefully be tied to an assumption about the corresponding Service: the Protocol request relates to a Service request/indication, and the Protocol reply to a Service response/confirm.

When the assumptions apply--as they do in the case of the protocol specified in this Technical Report--the use of the CCITT Rec. X.410, Chapter 2, notation results in an unambiguous and complete definition of:

- the Service,
- the Abstract Syntax of the Application Protocol Data Units (APDUs),
- the cause-effect relationship between Service primitives and APDUs.

C.5.1 Operation and Error Data Types

The building blocks of this specification method are two data types defined in CCITT Rec. X.410 Chapter 2: Operation and Error.

A data value of type Operation represents the identifier for an operation that an Application Entity (AE) in one open system may request be performed by a peer AE in another open system. A single data value, the argument of the operation, accompanies the request. Some operations report their outcome, whether success or failure. Other operations report their outcome only if they succeed, others only if they fail, and still others never at all. A single data value, the result of the operation, accompanies a report of success; a report of failure identifies the exceptional condition that was encountered.

The notation for an Operation type is the keyword OPERATION, optionally followed by the type of the operation's argument, the reference name optionally assigned to it, and the nature of the operation's outcome reporting (if any). If the operation reports success, the type of its result and the reference name optionally assigned to it are specified. If the operation reports failure, the reference names of the errors it reports are specified. The notation for an Operation value is the operation's numeric identifier.

A data value of type Error represents the identifier for an exceptional condition that an AE in one open system may report to a peer AE in another open system as the outcome of a previously requested operation. A single data value, the parameter of the error, accompanies the report.

The notation for an Error type is the keyword ERROR, optionally followed by the type of the error's parameter and the reference name optionally assigned to it. The notation for an Error value is the error's numeric identifier.

C.5.2 Invocation, Success, Failure and Rejection

CCITT Rec. X.410 Chapter 2 also uses the abstract syntax of CCITT Rec. X.409 to define the four classes of Protocol Data Units (PDUs): Invoke, ReturnResult, ReturnError, and Reject. The following summarizes the function of these PDUs.

The Invoke PDU requests that an operation be performed. It is sent whenever one AE desires assistance from another, and has three components. The invoke identifier component is used to correlate the request with its subsequent response. The operation component identifies the operation to be performed. The argument component is the operation's argument.

The ReturnResult PDU reports the successful completion of an operation. It is sent in eventual response to an Invoke PDU if the operation succeeds, and has two components. The invoke identifier component identifies the original request. The result component is the operation's result.

The ReturnError PDU reports the unsuccessful completion of an operation. It is sent in eventual response to an Invoke PDU if the operation fails, and has three components. The invoke identifier component identifies the original request. The error component identifies the error being reported. The parameter component is the error's parameter.

The Reject PDU reports the receipt and rejection of a malformed PDU. It is sent in eventual response to a malformed PDU whose type is other than Reject.

C.6 Remote Operation Service

CCITT Rec. X.410 does not specify all the details of how these PDUs are mapped onto the next lower layer. Instead, Chapter 2 states that the details of this mapping are application protocol specific.

This Technical Report uses the Remote Operations Service defined in ECMA TR/31, which provides an application protocol independent way of mapping the four PDUs--Invoke, ReturnResult, ReturnError and Reject--onto any suitable lower layer services.

APPENDIX D

FORMAL SPECIFICATION

D.1 Introduction

This Appendix is part of the Technical Report.

This Appendix formally specifies the OSI Directory Access Service and Protocol as a module in the notation of CCITT Rec. X.409/X.410.

D.2 Formal Specification

```
DirectoryService DEFINITIONS ::=
BEGIN

-- This module specifies the Directory Service Protocol's remote operations and
-- errors, as well as the types of their arguments, results, and parameters.

-- This module cites the following data types defined in other modules:

--  RemoteOperations.OPERATION -- see CCITT Rec. X.410 --
--  RemoteOperations.ERROR -- see CCITT Rec. X.410 --

OPERATION ::= RemoteOperations. OPERATION

ERROR ::= RemoteOperations.ERROR

-- TYPES AND CONSTANTS DESCRIBING NAMES --

NamePartType ::= INTEGER

NamePartValue ::= IA5String

NP ::= SEQUENCE {
    nptype [0] IMPLICIT NamePartType,
    npvalue [1] IMPLICIT NamePartValue}

wildcard IA5String ::= "*"  -- the wildcard character (asterisk) --

NamePart ::= NP --the npvalue shall not contain a wildcard--

NamePartPattern:: = NP --the npvalue may contain a wildcard--

-- There can be no wildcard characters in any of the following types
--

Name ::= SEQUENCE of NamePart

LeafName ::= Name

NonLeafName ::= Name

-- Wildcard characters are permitted in the final name part of the
following types --
```

```
NamePattern :: = SEQUENCE {
    nonfinal [0] IMPLICIT SEQUENCE OF NamePart
    final [1] IMPLICIT NamePartPattern}

LeafNamePattern :: = NamePattern

NonLeafNamePattern :: = NamePattern

-- Wildcard characters are permitted in any name parts of the
following type --

NameSpecification :: = SEQUENCE OF NamePartPattern

-- TYPES AND CONSTANTS DESCRIBING PROPERTIES --

PropertyType :: = INTEGER

PropertyValue :: = CHOICE {
        item [0] IMPLICIT OCTETSTRING,
        group [1] IMPLICIT SET OF NameSpecification }

Property :: = SEQUENCE {PropertyType, PropertyValue}

-- OTHER TYPES AND CONSTANTS --

-- The definition of filter, recursively from its components --

FilterComponent :: = CHOICE {
    [0]     NamePartType,
    [1] IMPLICIT NamePart,
    [2]     PropertyType
    [3] IMPLICIT Property }

Filter :: = CHOICE {
        [0] FilterComponent,
        and [1] IMPLICIT SEQUENCE OF Filter, -- at least two elements --
        or [2] IMPLICIT SEQUENCE OF Filter, -- at least two elements --
        not [3] Filter }

-- How the client identifies itself to the service --

Authenticator  :: = CHOICE {
                    [0] IMPLICIT UserName,
                    [1] IMPLICIT SimpleCredentials }

UserName :: = LeafName
UserPassword :: = OCTETSTRING

SimpleCredentials :: = SET {
                    [0] IMPLICIT UserName,
                    [1] IMPLICIT UserPassword }
```

```
-- OPERATIONS --

-- OPERATIONS APPLICABLE TO ALL ENTITIES--

readDistinguishedName OPERATION
    ARGUMENT SET {
        name [1] IMPLICIT NamePattern,
        agent [8] Authenticator OPTIONAL },
    RESULT SET {
        distinguishedName [0] IMPLICIT Name,
        match [1] IMPLICIT Name OPTIONAL }
    ERRORS { authenticationError,
        serviceError, accessControlError, nameError }
    ::= 4

readAliasesOf OPERATION
   ARGUMENT SET {
        pattern [1] IMPLICIT NamePattern,
        agent [8] Authenticator OPTIONAL }
   RESULT SET {
        distinguishedName [0] IMPLICIT Name,
        match [1] IMPLICIT Name OPTIONAL
        aliases [16] IMPLICIT SET OF Name }
   ERRORS { authenticationError,
        serviceError, accessControlError, nameError }
    ::= 9

createAlias OPERATION
    ARGUMENT SET {
        alias [0] IMPLICIT Name,
        sameAs [16] IMPLICIT Name,
        agent [8] Authenticator OPTIONAL }
    RESULT SET {
        distinguishedName [0] IMPLICIT Name }
    ERRORS { authenticationError,
         serviceError, updateError, accessControlError, nameError }
    ::= 10

deleteAlias OPERATION
    ARGUMENT SET {
        alias [0] IMPLICIT Name,
        agent [8] Authenticator OPTIONAL }
    RESULT SET {
            distinguishedName [0] IMPLICIT Name }
    ERRORS { authenticationError,
        serviceError, updateError, accessControlError, nameError }
    ::= 11
```

```
-- OPERATIONS APPLICABLE TO NON-LEAF VERTEXES ONLY --

readChildren OPERATION
    ARGUMENT SET {
        parent [3] IMPLICIT NonLeafNamePattern,
        pattern [16] IMPLICIT NamePartPattern OPTIONAL,
        filter [7] Filter OPTIONAL,
        agent [8] Authenticator OPTIONAL }
    RESULT SET {
        distinguishedName [0] IMPLICIT NonLeafName,
        match [1] IMPLICIT NonLeafName OPTIONAL,
        children [16] IMPLICIT SET OF NamePart }
  ERRORS { authenticationError, serviceError, accessControlError, nameError }
  :: = 5

readChildrenTypes OPERATION
  ARGUMENT SET {
        parent [3] IMPLICIT NonLeafNamePattern,
        agent [8] Authenticator OPTIONAL }
  RESULT SET {
        distinguishedName [0] IMPLICIT NonLeafName,
        match [1] IMPLICIT NonLeafName OPTIONAL
        types [16] IMPLICIT SET OF NamePartType }
  ERRORS { authenticationError, serviceError, accessControlError, nameError }
  :: = 6

readAliasChildren OPERATION
  ARGUMENT SET {
        parent [3] IMPLICIT NonLeafNamePattern,
        pattern [16] IMPLICIT NamePartPattern OPTIONAL,
        filter [7] Filter OPTIONAL,
        agent [8] Authenticator OPTIONAL }
  RESULT SET {
        distinguishedName [0] IMPLICIT NonLeafName,
        match [1] IMPLICIT NonLeafName OPTIONAL,
        children [16] IMPLICIT SET OF NamePart }
  ERRORS {
        authenticationError, serviceError, accessControlError, nameError }
  :: = 8
```

```
-- OPERATIONS APPLICABLE TO LEAF VERTEXES ONLY--

createLeafEntity OPERATION
   ARGUMENT SET {
         parent [2] IMPLICIT NonLeafName,
         distinguishedArcLabel [16] IMPLICIT NamePart,
         properties [17] IMPLICIT SET OF Property,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
         serviceError, propertyError, updateError, accessControlError, nameError }
   :: = 2

deleteLeafEntity OPERATION
   ARGUMENT SET {
         name [0] IMPLICIT LeafName,
         agent [8] Authenticator OPTIONAL }
   RESULT SET { }
   ERRORS { authenticationError,
         serviceError, updateError, accessControlError, nameError }
   :: = 3

-- OPERATIONS APPLICABLE TO ALL PROPERTIES --

readProperties OPERATION
   ARGUMENT SET {
         pattern [1] IMPLICIT LeafNamePattern,
         propertyTypes [16] IMPLICIT SET OF PropertyType,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName,
         match [1] IMPLICIT LeafName OPTIONAL,
         properties [16] IMPLICIT SET OF Property }
    ERRORS { authenticationError,
         serviceError, propertyError, accessControlError, nameError }
    :: = 16

readPropertyTypes OPERATION
   ARGUMENT SET {
         pattern [1] IMPLICIT LeafNamePattern,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName,
         match [1] IMPLICIT LeafName OPTIONAL,
         propertyTypes [16] IMPLICIT SET OF PropertyType }
   ERRORS {
         authenticationError, serviceError, accessControlError, nameError }
   :: = 15
```

```
addProperty OPERATION
   ARGUMENT SET {
         name [0] IMPLICIT LeafName,
         newProperty [16] IMPLICIT Property,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
         serviceError, propertyError, updateError, accessControlError, nameError }
   :: = 13

changeProperty OPERATION
   ARGUMENT SET {
         name [0] IMPLICIT LeafName,
         property [16] IMPLICIT Property,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
         serviceError, propertyError, updateError, accessControlError, nameError }
   :: = 1

deleteProperty OPERATION
   ARGUMENT SET {
         name [0] IMPLICIT LeafName,
         property [5] IMPLICIT PropertyType,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
         serviceError, propertyError, updateError, accessControlError, nameError }
   :: = 14

-- OPERATIONS APPLICABLE TO MEMBERS OF GROUPS --

readGroupMembers OPERATION
   ARGUMENT SET {
         name [1] IMPLICIT LeafNamePattern,
         primary [5] IMPLICIT propertyType,
         secondary [6] IMPLICIT propertyType OPTIONAL,
         filter [7] Filter OPTIONAL,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT Name,
         match [1] IMPICIT LeafName OPTIONAL,
         members [16] IMPLICIT SET OF NameSpecification }
   ERRORS { authenticationError,
         serviceError, propertyError, accessControlError, nameError }
   :: = 18
```

```
isMember OPERATION
   ARGUMENT SET {
         memberOf [1] IMPLICIT LeafNamePattern,
         primary [5] IMPLICIT PropertyType ,
         secondary [6] IMPLICIT PropertyType OPTIONAL,
         name [16] IMPLICIT NameSpecification,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName,
         match [1] IMPLICIT LeafName OPTIONAL,
         isMember [16] IMPLICIT BOOLEAN }
   ERRORS { authenticationError,
         serviceError, propertyError, accessControlError, nameError }
   :: = 20

addGroupMember OPERATION
   ARGUMENT SET {
         name [1] IMPLICIT LeafName,
         group [5] IMPLICIT PropertyType,
         member [16] IMPLICIT NameSpecification,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT Name }
   ERRORS { authenticationError,
         serviceError, propertyError, updateError, accessControlError, nameError}
   :: = 12

addSelf OPERATION
   ARGUMENT SET {
         name [1] IMPLICIT LeafName,
         group [5] IMPLICIT PropertyType,
         agent [8] Authenticator }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
         propertyError, accessControlError, serviceError, updateError, nameError}
   :: = 17

deleteGroupMember OPERATION
   ARGUMENT SET {
         name [1] IMPLICIT LeafName,
         group [5] IMPLICIT PropertyType,
         member [16] IMPLICIT NameSpecification,
         agent [8] Authenticator OPTIONAL }
   RESULT SET {
         distinguishedName [0] IMPLICIT Name }
   ERRORS { authenticationError,
         serviceError, propertyError, updateError, accessControlError, nameError }
   :: = 7
```

```
deleteSelf OPERATION
   ARGUMENT SET {
         name [1] IMPLICIT LeafName,
         group [5] IMPLICIT PropertyType,
         agent [8] Authenticator }
   RESULT SET {
         distinguishedName [0] IMPLICIT LeafName }
   ERRORS { authenticationError,
         serviceError, propertyError,updateError,  accessControlError, nameError }
   ::= 19

-- REMOTE ERRORS AND THEIR ARGUMENTS --

-- GENERAL PARAMETER FOR ARGUMENT RESOLUTION--

WhichArgument ::= INTEGER {
     first(1), -- concerns the first name or property argument --
     second(2) } -- concerns the second name or property--
                 -- argument--

-- ERRORS CONCERNING AUTHENTICATION --

AuthenticationError  ERROR
   PARAMETER SET {
         problem [0]  IMPLICIT  AuthenticationProblem }
   ::= 6

AuthenticationProblem ::= INTEGER {
   userNameInvalid (50),
   userPasswordInvalid (51),
   inappropriateCredentials (52) }

-- ERRORS INDICATING INABILITY TO FULFILL REQUEST --

serviceError ERROR
   PARAMETER SET {
         problem [0] IMPLICIT ServiceProblem,
         hint [1] IMPLICIT LeafName OPTIONAL }
   ::= 5

ServiceProblem ::= INTEGER {
   dSATooBusy (40),
   wrongDSA (42),
   chainingFailed (43),
   unsupportedFilter (48),
   recursionNotImplemented (49) }

-- PROPERTY ERRORS --

propertyError ERROR
   PARAMETER SET {
         problem [0]  IMPLICIT  PropertyProblem,
         which [2] IMPLICIT WhichArgument,
         distinguishedName [3]  IMPLICIT  LeafName }
   ::= 3
```

```
PropertyProblem ::= INTEGER {
   missing(20),
   wrongType(21) }

-- ERRORS REPORTING FAILURE OF AN UPDATE REQUEST--

updateError ERROR
   PARAMETER SET {
    problem [0] IMPLICIT UpdateProblem,
    found [1] IMPLICIT BOOLEAN,
    which [2] IMPLICIT WhichArgument,
    distinguishedName [3] IMPLICIT LeafName }
   ::= 4

UpdateProblem ::= INTEGER{
   noChange(30),
   entryOverflow (32),
   databaseOverflow(33) }

-- ERRORS DUE TO ACCESS RESTRICTIONS --

accessControlError ERROR
   PARAMETER SET {
    problem [0] IMPLICIT AccessControlProblem }
   ::= 1

AccessControlProblem ::= INTEGER {
   accessRightsInsufficient (1) }

-- ERRORS CONCERNING NAMES --

nameError ERROR
   PARAMETER SET {
    problem [0] IMPLICIT NameProblem,
    which [2] IMPLICIT WhichArgument }
   ::= 2

NameProblem ::= INTEGER {
   overspecified (1),
   erroneous (3),
   underspecified (4),
   illegalNonLeaf (5) }

END. --of Directory Service--
```

D.3 Standard Property Types

```
-- PROPERTIES DEFINED IN THIS TECHNICAL REPORT--

members PropertyType ::= 64

pSAPaddress PropertyType ::= 65
```

(Blank page)

APPENDIX E

STANDARD PROPERTY TYPES

E.1 Introduction

This Appendix is part of the Technical Report.

This Technical Report defines standard properties which are likely to be needed very commonly in OSI Directories.

Note E.1
Further standard Properties may be defined in other Technical Reports or in future version of this Technical Report.

Note E.2
The section on Conformance (Section 4) does not define conformance to standard property types, because this matter does not affect the operation of either a DUA or a DSA. Rather, it relates to conventions adopted (or not adopted) by users of the Directory Access Service. However, such users are strongly advised to treat the standard property types as reserved for the uses indicated, and not to use them for other purposes.

E.2 Internal-Use Properties

Property Types 0-63 inclusive are reserved for internal use by the Directory System.

E.3 Members Property

Reference Name of Property:	"members"
Property Type:	64
Kind of Property:	group
Semantics:	An entity which has this Property is considered to be a "group object" and the members of this property are the names of the members of the group.
Examples:	distribution list, list of employees of a department

E.4 PSAP Addresses Property

Reference Name of Property:	"PSAP addresses"
Property Type:	65
Kind of Property:	item
Value of Property:	The item value (OCTETSTRING) has the Abstract Syntax SEQUENCE OF OCTETSTRING and is encoded by the encoding rules of CCITT recommendation X.409. Each OCTETSTRING in the sequence is an OSI PSAP Address in the form to be defined by OSI addressing standards.

(Blank page)

APPENDIX F

ACCESS CONTROL

F.1 Introduction

This Appendix is not part of the Technical Report.

This Appendix describes an access control mechanism that implementors may provide.

F.2 Directory Service Access Controls

Access control means the ability of the Directory Service to determine, for each specific request for service, whether the purported client has the right to perform that operation on the specified data.

The Directory Service should provide two types of access control on update operations: administration and self-administration. These controls distinguish classes of user privilege, defined as the ability to perform certain operations.

Given the intended use of the Directory Service, read access controls are much more costly, and also less critically necessary, than the administrative ones. Some of the problems inherent in read access controls are briefly discussed. It is suggested that an in-depth study of the subject should precede standardization of these controls.

A DSA may provide access control by means of user groups, called access control lists This means that an access control list consists of a list of the names of entities (i.e., Application Processes that are registered in the Directory Service) that are permitted to pass this control. There are administration control lists and self-administration control lists.

Access control lists may have patterns among the members.

F.3 Administration

Administration access controls apply to all operations that modify the database, with the exception of those operations governed by the weaker type of self-administration.

Administration access controls can apply to non-leaf and/or leaf entities and/or properties.

A Directory System Naming Domain (ND) specific administration control list applies to the entire ND information base, except where it is overridden by a lower level access control. It is not necessary for every ND to have an ND specific administration control list, because it can inherit that of its parent. It should, however, be possible to attach such a list to any ND.

Access to a leaf entity is ruled by the controls that apply to its parent. If an implementation provides leaf entity specific access controls, the corresponding list will, where present, override the effect of the list belonging to the parent and rule control to the entry.

A Property-specific access control list may, in principle, be attached to any property overriding the effect of any higher level lists on that specific entry component. The absence of such a list indicates that access is ruled by the next higher level control list in effect.

F.4 Self-Administration

Self-administration access controls apply only to group properties (properties of the group type).

The effect of a self-administration control list is to allow users to add and remove themselves from the member list attached to the group property. The privilege granted by this list is in addition to any privilege granted by the administration control list (Clause F.3).

If no self-administration control list is present, or if the list is empty, no users are allowed to add or remove themsleves from the group (except those that have administrator privilege according to Clause F.3).

F.5 Read

The basic problem with read access controls is that they tend to defeat one of the principal design considerations of Directory Systems: that it must perform highly efficiently, and be highly available, under conditions where update frequency is low and query frequency is high.

Because read access controls are so sensitive in terms of overall systems performance, ECMA recommends that an in-depth analysis of the requirements and technical constraints must be performed before such controls are included in a future version of this Technical Report.

The most costly read controls are those at the leaf entity and/or Property level It seems likely that controls for non-leaf entities would be less costly, mainly because they would tend to be swapped in core all the time--perhaps also because these wider access control lists would tend to be much less complex than leaf entity-specific ones. At the same time, the wider controls would be much less powerful than those of finer granularity, and even these simple controls would imply the need to check every incoming request against the access control list.

Of all the Property-specific access controls, probably the most useful, and possibly the least costly in terms of overall systems performance, would be controls that can be applied only to groups. The structure of these read access controls could be roughly the same as in Clause F.4.