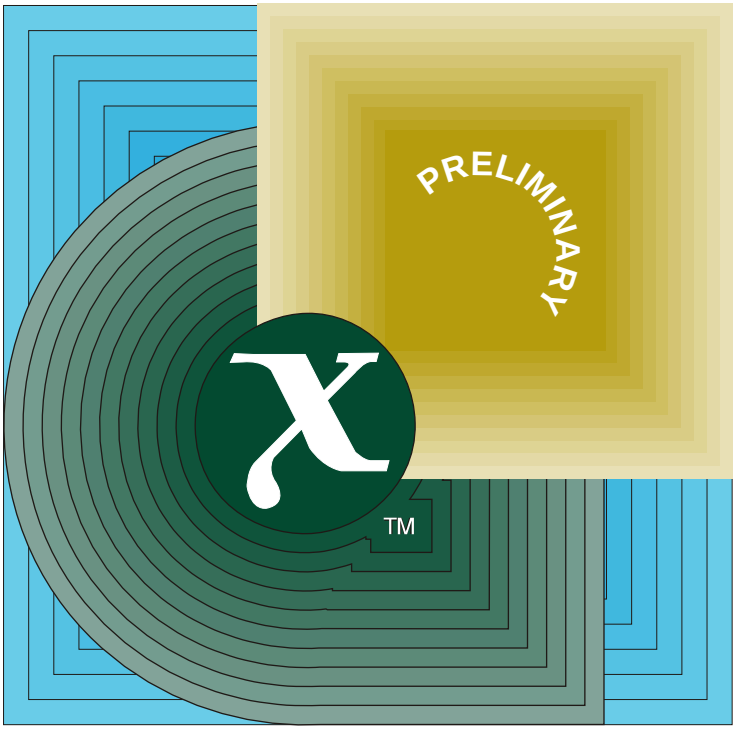
Preliminary Specification

Systems Management: Topology Service



THE Open GROUP

[This page intentionally left blank]

Preliminary Specification

Systems Management:

Topology Service

The Open Group

Preliminary Specification

Systems Management: Topology Service

ISBN: 1-85912-127-6
Document Number: P701

Published in the U.K. by The Open Group, March 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

One of the fundamental needs of an automated management system is to be able to keep track of where resources are located and how they are related to other resources in the managed environment. Topology provides a service to applications for managing topological relationships (associations) among managed objects in a distributed workgroup and up through the Enterprise environments.

A Topology Service relieves most of the burden that applications have in managing associations, by providing storage of topological data, by maintaining semantic integrity of associations, and by providing query support to clients interested in retrieving topological information. As such, Topology serves as a fundamental integration point for management applications in determining how managed objects are organized, what their inter-dependencies are, and how underlying aspects of the managed environment blend together to form the resources which administrators manage. Classes of applications which benefit from this integration include fault management applications, map generators, correlators, and configuration management tools.

Structure

- **Chapter 1** examines the purpose, scope and requirements of a topology service.
- **Chapter 2** describes the high-level architecture, key concepts and development process involved in this topology service.
- **Chapter 3** gives the data definitions use in this topology service.
- **Chapter 4** describes how to populate the Topology Data Store, and once populated, how to manage it.
- **Chapter 5** describes how to manage a collection of Topological Types.
- **Chapter 6** desribes the Topology Query interface.
- **Appendix A** provides more information on Topology Query.

Typograpgical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — variable names, for example, substitutable argument prototypes, parameters and environment variables
  - — options in text
  - — function calls are shown as follows: *name*( )
- Normal font is used for the names of constants and literals.
- CORBA IDL code fragments are shown in **helvetica bold** font.

Trade Marks

Motif®, OSF/1® and UNIX® are registered trademarks and the ‘‘X Device’’™ and The Open Group™ are trademarks of The Open Group.

Referenced Documents

The following documents are referenced in this specification:

OODFM

D. Coleman, et al., Object-Oriented Development: The Fusion Method, published by Prentice-Hall.

XCMFv1

Preliminary Specification, June 1995, Systems Management: Common Management Facilities, Volume 1 (ISBN: 1-85912-047-4, P421).

Federated Naming

CAE Specification, July 1995, Federated Naming: The XFN Specification (ISBN: 1-85912-052-0, C403).

Index

Chapter 1

Introduction

1.1 Purpose

With the move to re-engineer and automate tasks and processes, coupled with the move to distributed systems environments, we have seen the need to have more robust and cooperating network and systems management and administration tools.

One of the fundamental needs of an automated management system is to be able to keep track of where resources are located and how they are related to other resources in the managed environment. Therefore, a well designed Topology information management service will be of increasing importance. Topology provides a service to applications for managing topological relationships (associations) among managed objects in a distributed workgroup up through the enterprise environments.

A Topology Service relieves most of the burden applications have in managing associations by providing storage of topological data, by maintaining semantic integrity of associations, and by providing query support to clients interested in retrieving topological information. As such, Topology serves as a fundamental integration point for management applications in determining how managed objects are organized, what their inter-dependencies are, and how underlying aspects of the managed environment blend together to form the resources administrators manage. Classes of applications which benefit from this integration include fault management applications, map generators, correlators, and configuration management tools.

As management environments become increasingly complex, and due to the integral role that topological data has in management, a Topology Service necessarily must:

- scale well to address the broad range of size seen today in customer environments
- be extensible to accommodate new technologies and unanticipated topologies.

Accordingly, major operational goals pertaining to management that the Topology Service supports include:

- the provision of an environment in which applications can integrate with one another, via topological information, by means of consistent paradigms
- the enabling of scalable solutions, whereby an application can usefully participate in managing hundreds to millions of objects.

1.2 Scope

This specification addresses Topology information management services for network and systems management purposes. However, this technology may well be applicable outside of its original charter (for example: relationship management of other applications).

It is not the function of the Topology Service to ‘‘discover’’ topological data or to manage assets. It is also not the function of this service to ‘‘display’’ information to the user. This service cooperates with other services such as Discovery Services, Event Services, Inventory Management or Asset Management Services, and User Presentation Services such as map services or map generation services, etc.

Likewise, this service does not specifically address ‘‘interchange’’ of information between implementation-specific topology managers. However, this service could support an additional component that performs interchange functions.

1.3 Requirements

1.3.1 Association Management

A Topology Service must provide a general, distributed service for managing topological associations that exist among managed objects in a consistent manner.

It must manage the lifecycle of associations as directed by clients using the service. Operations include the ability to:

- make managed objects known to Topology (manage and unmanage them) so associations can be established with them
- create, delete, and query associations
- notify clients of such association changes
- represent associations of arbitrary degree (relating multiple objects with a single association) in a manner consistent with the overall Topology paradigm.

It must provide the ability to validate that associations may be allowed and the validation process must be client-definable and extensible so that the Topology Service can be made to manage arbitrarily-defined topologies, newly and dynamically defined by clients of the service. The extensible aspects which the service must support include:

- configuration of the kinds of associations that can be established with what types of objects
- specification of the minimum and maximum number of a given kind of association that an object type can participate in
- dynamic alteration of topology semantics if and when required
- policy imposition and removal, enforced by Topology as topological edits take place, as needed.

Semantics of the *associated objects* are unaffected by their involvement in topological associations.

Validation must also be enforced when a client may request an edit to the topological information. This enforcement may allow or prohibit the client to make the edit, based on the enforcement process results.

1.3.2 Scalability and Performance

The Topology Service must scale well to be able to support very large environments that may require management of the order of *millions* of objects, yet be able to also not be so heavyweight and burdensome as to not be appropriate for small workgroup environments who may only have *hundreds* of objects to manage.

The Topology Service should allow the storage of relationships between CORBA objects and non-CORBA objects.

Topological data, internal to the service, should not be required to be exposed in the form of directly addressable objects in and of themselves (such as directly addressable CORBA objects).

The Topology Service should not restrict implementations from being federated in order to support improved performance in large-scale, distributed environments.

1.3.3 Query Support

The Topology Service should support arbitrary queries which are client-defined. Example queries might include:

- Which clients are currently mounting file systems from NFS server Q?
- Which users are currently running shared Application X?
- What systems are dependent on router A for connectivity to P, Q and R?
- What departments are responsible for managing the PC inventory in Building 6?

Queries should be able to be stored, re-used and referenced as sub-queries from within other queries.

1.3.4 Application Integration

The Topology Service must provide semantics that are well defined and established so that all clients (both internal within a product and third-party supported clients) can edit topology data.

The Topology Service must allow for clients to be able to build on the work of others by creating and manipulating associations involving objects that have been defined and managed by other applications.

The Topology Service should support extensions to new topologies based on existing objects may participate in. This must be done in such a way that the semantics and implementations of existing objects are unaffected by new associations they are called to participate in via the Topology Service.

The Topology Service must provide an integrated, comprehensive view of both logical and physical worlds.

The Topology Service should provide direct support for *integrated* management of logical and physical worlds where these worlds are tied together. In particular, Topology should support integrated views of resources that actually consist of cooperating logical and physical objects, while retaining individuality and direct management access to the underlying objects:

> The Topology Service should directly support integrated management of systems, networks and distributed applications. Queries should be able to span linkages *transparently* among these topologies through its integrated views and common storage.

The Topology Service should allow applications to query all topological data (subject to security restrictions). It should integrate the various topologies that objects participate in such that queries can be written to retrieve *unified* views (spanning associations defined by separate and distinct topologies).

The Topology Service should allow for queries to be named and shared (re-used) by applications other than those which defined the queries.

The Topology Service should integrate with external event services so applications can subscribe to change notifications pertaining to topological data:

- Topology should define and generate events to notify interested clients of changes (such as creations, deletions and updates) that occur on the basis of:
  - — object types
  - — object state

— association instances

— the management and unmanagement of object instances by topology.

The Topology Service should maintain a persistent store of its topological data. If the topological data is stored in an RDBMS, the underlying tables should be published to support read access by independent report generators. However, the form of the data in the relational store may not be conducive to simple or efficient access.

1.3.5 Interoperability

The Topology Service should support the interchange of topological data with heterogeneous implementations, including:

- the exchange of associations
- the exchange of objects involved in the associations
- ideally, the semantics which apply to valid topological representations.

1.3.6 Usability

The Topology Service should support a usable, intuitive interface:

- provide a paradigm that hides Topology-internal representations from the client to relieve the burden from the client of dealing with Topology internals.

The Topology Service interface and querying capabilities should support:

- comprehensive views of cooperating components, to provide individual views of aggregated components and provide queries based on either comprehensive or individual views
- provide transparency between comprehensive and individual component views that allows comprehensive representations to take on the characteristics of all of their individual components.

1.4 Relationship to Existing Services

1.4.1 Relationship Service

The OMG Relationship Service has considerable overlap in its functionality with the proposed Topology Service. In particular they both address the following key requirement:

- Manage relationships between objects of which the objects themselves are unaware.

However, Topology has some additional requirements that yield a different service:

- Scale up to the management of relationships of hundreds of thousands of objects.
- Provide a mechanism to integrate relationship data from disparate application spaces and present a unified view of the complete set of relationships. This is the aggregation concept which is explained in the abstract portion of the Topology submittal.
- Manage relationships between things that are not CORBA objects.

The particular interface defined for Topology yields some additional benefits beyond the basic requirements outlined above:

- The distinction between *relationships* and *entities* is blurred. This is important as an integration enabler, because what one application views as a relationship may be an entity to another.

  Consider the example in which one application manages *People that are Married* to each other, and a different application manages *Marriages which are Performed by Clergy*. In the first application, *Marriage* is a relationship between persons, but in the second application *Marriage* is an entity that participates in a different sort of relationship with a clergy.

- The Topology Service maintains information about the kinds of entities and relationships that may be managed. This information can be requested via the defined interfaces and may also be augmented at run-time on the fly.

  The OMG Relationship Service provides no such standard mechanism for a client to gather this information, and new relationships and roles can only be defined by creating new CORBA interfaces — generally a compile-time activity.

1.4.2 Transaction Service

Topology does not present any interfaces that overlap or attempt to augment the Transaction Service interfaces. However, Topology is defined to operate in an environment where a transaction service can help maintain consistency of the topology. Thus, some transaction service is needed in order to implement Topology as well as to use it. The OMG Topology Service is an appropriate service to use for this functionality. While it would be feasible to implement a Topology Service using a different transaction service, this would be counter productive since the clients of the service must also use the transaction interfaces.

Topology provides a powerful declarative query mechanism using a formal language specifically designed for querying the relationships that are managed by Topology. It would have been appealing to simply use the OMG Query Service interface for querying topology, however, the Query Service requires the use of the OQL/SQL which is not suitable for use in an environment where the objects being queried have no knowledge of the relationships in which they participate and in which the relationship store has no knowledge of the objects being managed.

1.4.3 Factory Service

Topology does not provide any general lifecycle management services. Thus, there is no overlap with the GenericFactory interface.

1.4.4 Persistence Service

Topology does not present any interfaces that overlap or attempt to augment the Persistence interface. A Topology implementation will usually keep a persistent store of its data and may use whatever mechanism the implementor chooses to use.

1.4.5 XCMF Policy Service

There is some overlap of functionality between certain features of the Topology Service and the ‘‘Common Management Facilities’’ (XCMF) Policy Management Service (see referenced document **XCMFv1**). Topology has features that enable clients of the service to define rules that apply to relationships between topological entities, or arbitrary rules applied to the state of topological entities that are involved in changes to e underlying topological relationships. Topology enforces these rules automatically whenever a change to the topology being managed is made. Similarly, XCMF Policy Management allows clients of the service to define rules that apply to any portion of the state of each managed object. The state information to which these rules may apply could include, but is in no way limited to, information about the relationships the managed objects participate in. XCMF Policy Management does not automatically enforce policies defined on the state of managed objects. Instead, policy enforcement must be requested by clients of the service.

It would not be feasible to implement XCMF Policy Management in terms of the support for policy provided by the Topology Service, since the latter requires a state change between topological relationships to trigger policy enforcement, and the former allow policy enforcement at any arbitrary time — which may not be preceded by a change in the relationships in which managed objects participate.

It may be possible to implement the support for policy enforcement provided by Topology in terms of XCMF Policy Management. For instance, whenever relationship changes occur CORBA proxies could be created for the topological entities affected using Instance Managers, and Validation Policies could be invoked corresponding to all rules defined by Topology. But, this solution may be sub-optimal, since it requires creation of first class CORBA objects corresponding to all topological entities involved in a state change.

Whether or not the support for policy management provided by Topology is implemented in terms of XCMF Policy Management is an issue which should be left as an implementation detail within the Topology Service, since the choice of whether or not to do this would not impact the interoperability of two implementations of Topology. Also, too little is known at this time about whether or not this is an optimal solution to make the relationship between this services a requirement to an implementor. Instead, it makes more sense to view this as an implementation detail at this time, opening up the possibility of both integrated, and non-integrated solutions, which will ultimately provide for more direct comparison between the various approaches.

1.4.6 XCMF Instance Management

There is no obvious relationship between Topology and XCMF Instance Management. Topology manages the relationships between entities (objects), while XCMF Instance Management manages the lifecycle of object instances by type, along with the policies that may be applied to each type of object instance. Most notably, Topology does *not* manage any aspect of the lifecycle of the entities it manages.

1.4.7 XCMF Managed Sets

There is clearly some functionality overlap between the Topology Service and XCMF ManagedSets. Essentially, Topology is capable of representing and managing any type of relationship between managed entities. XCMF ManagedSets is capable of managing one particular type of relationship that can exist between managed objects: the member/set relationship.

Since the type of relationship supported by ManagedSets is a subset of the types of relationships that are supported by Topology, it makes no sense to consider implementing Topology in terms of ManagedSets.

Conversely, however, it may be possible to implement XCMF ManagedSets in terms of Topology. To do so, the implementation would likely be such that a topological entity of type **Member** would be created for each XCMF object supporting the Member interface, and a topological entity of type **Set** would be created for each XCMF object supporting the Set interface. Whenever an XCMF Member object is added to an XCMF Set object, a relationship would be established between the corresponding Topological entities within Topology. All queries to the XCMF ManagedSet objects for things such as membership of a Set of which Sets a Member belongs to would be delegated as queries to Topology.

We do not feel, however, there has been sufficient implementation experience with either XCMF or Topology to make the implementation of XCMF ManagedSets in terms of Topology a strict requirement at this time. The implementation described above could add appreciable overhead to an implementation of XCMF ManagedSets not integrated with Topology, for example, since the proposed implementation essentially requires delegation of requests on ManagedSets interfaces to Topology. As with the issue of overlap between the XCMF Policy Management Service and the support for Policy Management supported by Topology, we feel the issue of whether one could/should be implemented in terms of the other should be left as an implementation detail at this time. If integrated implementations emerge that demonstrate clear advantages over non-integrated implementations, then issues such as this can be revisited at this time.

In general, there is a strong argument for specifying XCMF and Topology in such a way as there are no dependencies between them. Specifying these as two orthogonal sets of services allows more implementation options, and hopefully will produce more total implementations. Making the requirement that one depends on the other places a heavier-weight requirement on any organization considering the implementation of one or the other, and thus may prohibit implementations of either.

Chapter 2

Topology Service Overview

The Topology Service provides a general, distributed service for storing and querying topological associations between objects. The Topology Service is extensible to handle any developer-defined topologies, including, for example, those found among users and computer systems, any variety of networks, the hardware arrangements of computing devices, I/O cards and peripherals, or shared software and the departments that manage its use.

The Topology Service is highly useful in the handling of management problems, as it removes most of the burden applications have in storing, manipulating and querying associations among the resources being managed. This not only reduces application development cost, it also exposes a key integration point for applications: a rendezvous point for determining how managed resources are organized, configured, or connected, and for how separate, underlying aspects of the managed environment blend together to form the resources administrators manage.

Major operational goals pertaining to management that the Topology Service addresses include:

- the provision of an environment in which applications can integrate with one another via topological data through consistent paradigms
- the enabling of scalable solutions, whereby an application can usefully participate in managing environments that range in size from an order of hundreds up to an order of millions objects
- the ability to establish new associations among objects without affecting the semantics or implementations of those objects.

The Topology Service is configurable and extensible. It allows applications to define what constitutes valid associations that characterize a particular topology. This is done by specifying the rules that the associations must obey which the Topology Service will then enforce. The Topology Service then provides storage of the associations and querying capabilities for retrieval.

2.1 High-level Architectural View

Figure 2-1 shows the major components of a Topology Service being submitted for standardization. These are:

- Semantics Manager API
  which is used by a client to define the topological rules which form the semantics for a given topology
- Data Manager API
  which is used by a client to populate the Topology Service with associations that conform to the semantics of previously-defined topology rules
- Query Manager API
  which is used by a client to retrieve topological data based on registered (and sharable) queries.

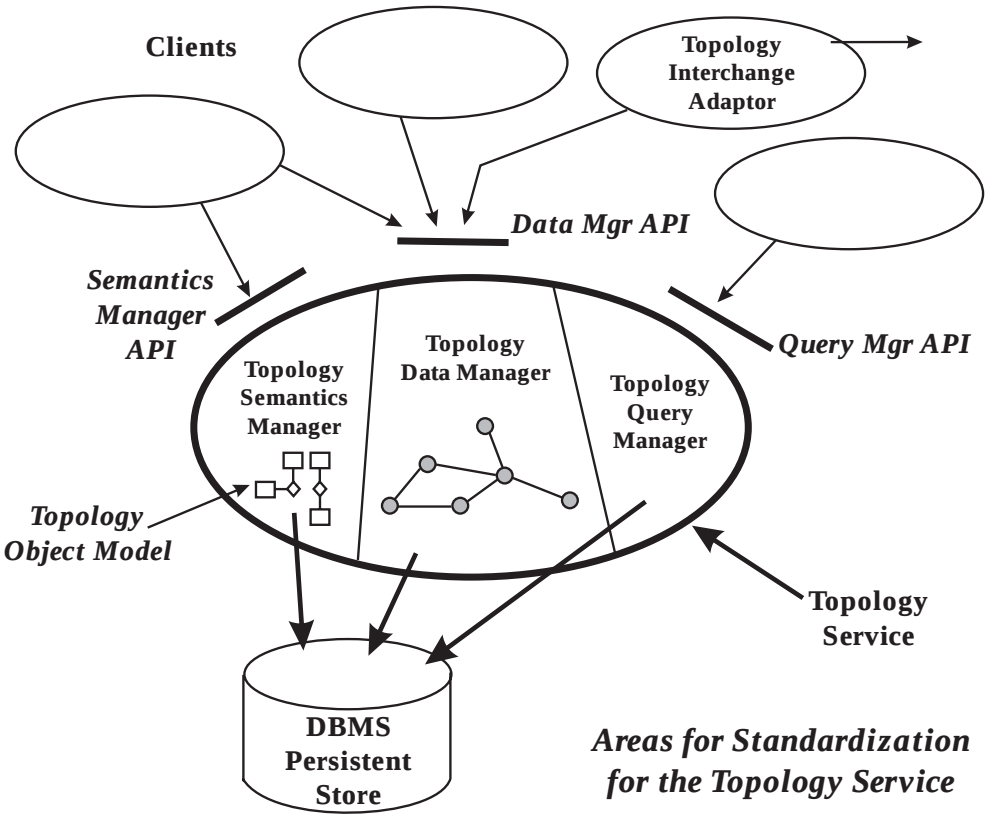


Figure 2-1 High-level Architcture View of Topology Service

2.2 Key Concepts

2.2.1 Topology

A topology is a set of valid associations between objects. In the Topology Service the set of valid associations in a topology is constrained by a set of *topology rules*. As an example in network management, a TCP/IP network topology can be established which is governed by rules dictating conditions such as:

- all IP network addresses must be unique
- a host can be connected to a network only if one of its network interface cards has an IP address which matches the address of the network.

As an example in system management, topology rules can be established which define a topology of users and distributed computing resources. Such a topology would consist of associations that indicate how users map to various login accounts and what corresponding access each has to particular system resources. The corresponding topology rules might dictate how many users can simultaneously have access to a given type of resource, and under what conditions.

2.2.2 Entities and Aggregate Entities

The Need for Aggregates

Management interfaces to resources in a distributed network and system environment are typically very complex. Although often viewed by their users as single components, managed resources often consist of several, possibly tens of smaller, individually managed objects.

These smaller objects are often a mixture of logical and physical components which work together to offer functional services to which applications can subscribe. Any or all of the smaller components can change (due to upgrades, repairs, migration, physical moves, etc.) Yet as these changes to underlying components take place, the combined resource itself (usually) persists. Such a combined resource is what users and administrators generally know by a common name and perceive in terms of the services its combined, underlying components provide.

Consider the example of a file system. While the file system is typically viewed as a whole, it actually consists of a logical component and a physical component. The logical component is a hierarchical file system (hfs) which defines the structure of contained files and directories. The physical component, is a hard disk device which has its media organized by the hfs. Together they form the file system. Both are required, yet each can be managed individually. For example:

1. by taking the appropriate steps, the hard disk can be replaced with another, while still retaining the file system as it is known to its users
2. asset management of the physical hard drive is handled without any consideration given to the hfs implemented over it
3. by logically mounting the hfs to another location in the computer system's complete file system structure, the file system has moved, yet physically it has been unaffected.

Consider now the administrator's desire to export the file system to clients via NFS. To do so, another logical component is needed: an *exported file system* which provides a remote access point for mounting and which retains a list of clients who may have access to the exported file system. There are several points worth noting in this example:

1. The *exported file system* component also represents the file system, along with the hfs and hard drive components — yet it has a very different lifecycle.
2. Not every file system gets mounted. Thus, only those being exported require the additional *exported file system* component.
3. It is the *exported file system* which participates in an NFS topology which in turn identifies the clients and servers forming the *distributed* file system. The other file system components, such as the hfs and the disk drive, neither have knowledge of NFS nor should they.
4. If this file system were exported instead via non-NFS protocols (such as those provided by Netware), a variation of the NFS *exported file system* would be required to adequately manage the export.
5. If this file system were exported via both NFS and Netware, it would be appropriate and correct to have an *exported file system* component for *each*, so that the facets of each exporting environment can be managed individually and independently.

So, in this example, we see a single resource — a file system — which is nevertheless based on multiple, underlying components. Some of these components always exist while others may come and go more frequently, and yet all require some degree of individual management.

This example illustrates a frequently-encountered situation:

- Resources and services are often viewed as units because they operate as units, yet they are truly a collection of cooperating components that must be individually managed. Furthermore, it is the underlying components that characterize a resource's participation in various topologies.

This situation raises two issues for resource management:

- How does one best maintain the perspective, identity, and longevity of a combined resource, while allowing direct management access to its underlying components as individual entities?
- How does one appropriately model a resource to allow it to participate in a variety of topologies, some of which are physical, some of which are logical, some of which are seemingly in conflict with one another, and some of which are not even conceived yet?

A solution to this problem would make sufficiently transparent the key underlying components that intrinsically form a resource as well as the topologies those components participate in. Such a solution becomes highly useful in:

- bridging the gaps between logical and physical worlds that interact together to form resources
- enabling traversals through and across separate topologies in which resource components participate.

Both these situations characterize the nature of the networks and systems that make up resource management. Consequently, applications providing *integrated* management of:

- networks

- systems
- networks *and* systems
- distributed applications which depend on both networks and systems

would all benefit fundamentally from such a solution.

The Topology Service provides a solution to this problem through means of a special relationship, called aggregation[1]. Internal to Topology, individual managed objects, such as the hfs, the hard disk, and the exported file system in the above example, are represented by *topological entities*. When topological entities are recognized to be part of a larger resource (such as the file system in the example above) the Topology Service can be told to join them into an *aggregate entity* (AE).

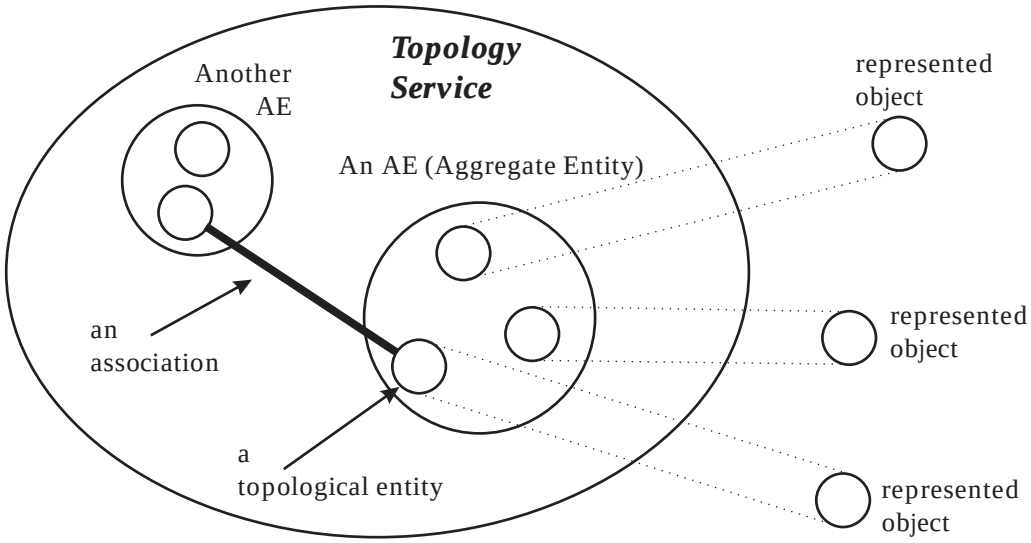


Figure 2-2 Topological Entities and Aggregate Entities

As a result, Topology provides aggregate views, as well as individual entity views, of the components that participate in the topologies it stores[2].

In the above file system example, aggregation means that access can be made to the file system AE, whereby information pertaining to its topological entities — its hard disk, its hfs, and any exported file systems — are immediately available. From the AE, any of the topological associations that topological entities participate in can be seen and traversed. Figure 2-3 illustrates the concept using the File System example for UNIX systems.

1. The Aggregation relationship is considered ''special'' to Topology because it has certain characteristics that set it apart from the administration of associations in general, as will become evident.
2. Note that objects are referenced, but not managed, by Topology. *Applications* manage objects, while *Topology* manages the topological characteristics pertaining to objects — that is, the *Topological Entities*, the *Associations* between them, and the *Aggregate Entities* which provide a complete view of the entities being managed.

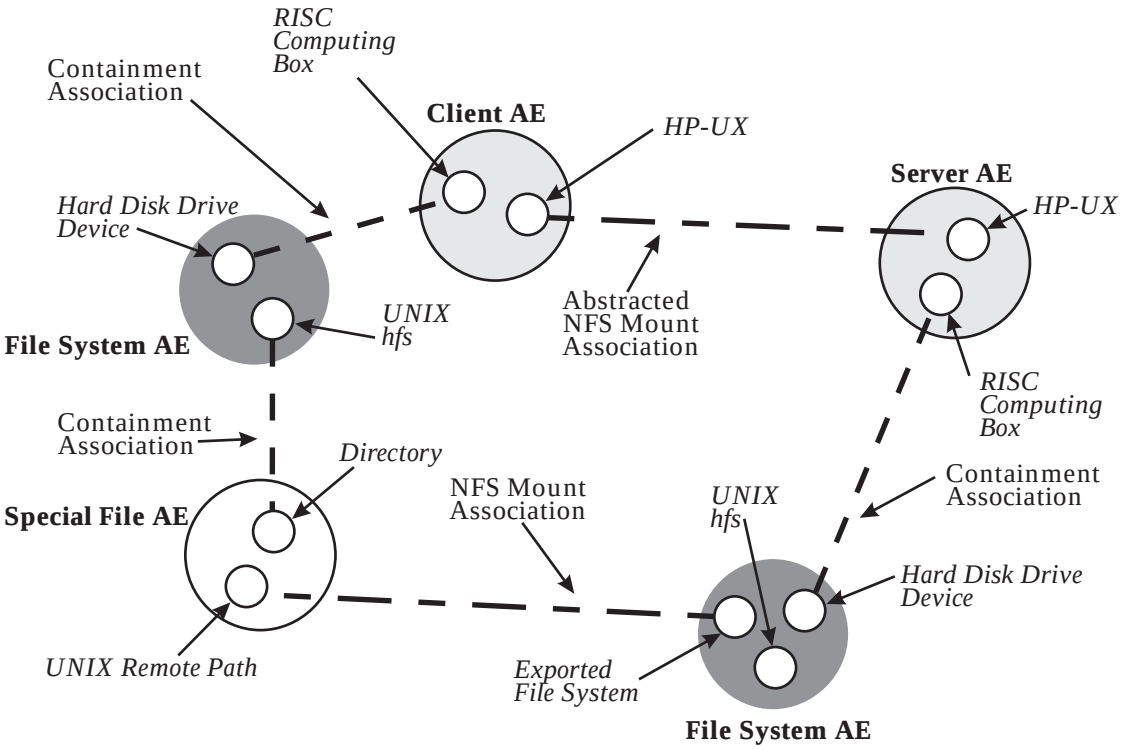


Figure 2-3 AEs and Topological Entities in the File System Example

This benefit of entity aggregation becomes evident when navigating through the topology tree. AEs provide a degree of transparency whereby traversals from one topology to another is readily accomplished via the topological entities that form an AE. In the above example, the ‘‘Abstracted NFS Mount Association’’ between the Client and Server is in fact derived from navigating through three types of topologies: physical equipment containment, logical file system structure, and the NFS topology. The Topology Service query subsystem takes advantage of the integration and transparency provided by AEs to make such a query easy to specify and execute. Without the aggregation provided by AEs, such a query would be far more cumbersome to develop.

2.2.3 Topology Rules

The Topology Service allows developers to dynamically specify *topology rules* which must be met for every instantiated association. As part of these rules, the object-oriented concept of **type** is used to distinguish different entities that make up an AE. Every topological entity is assigned such a *topological type*. Based on these types, topology rules express the kinds of associations allowed with various other topological types[3].

3. It is useful to note that the use of types allows the expression of ‘‘topologies’’ to follow easily from the developer’s Object-Oriented Analyses (OOA) of their respective problem domains. See Section 2.4 on page 23 for an example of this.

Examples of topological types include NFS Server and NSF Client for an NFS topology. For an SDH topology, topological examples include Virtual Circuit, Trail, Add-Drop Multiplexer and Synchronous Digital Crossconnect.

Note that topological types are managed purely as a Topology Service concept. Developers can define new types and assign them to entities in whatever manner makes sense for their applications and for the topologies being represented. In practice, however, topological types will typically correspond directly to the *interfaces* exposed by the *objects* represented by topological entities.

The topological types assigned to entities are transitive to their AEs. Thus, an AE takes on all the topological types of its entities.

Associations are characterized by *roles* and by *cardinality*. Association *roles* dictate the parts played by topological types on either end of an association. Association *cardinality* indicates the minimum and maximum number of these associations that can be established with an entity of a given topological type. Associations in Topology are strictly binary in nature, that is, each association links exactly two topological entities. (Relationships of degree n are modelled by representing the relationships as Topological Entities themselves.)

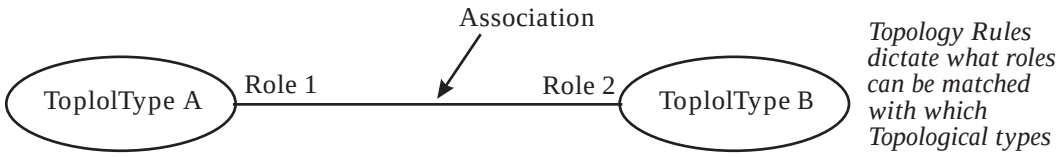


Figure 2-4 Matching Topology Roles with Topological Types

2.2.3.1 Topology Rule Format

Topology rules are assigned to specific topological types. They dictate the kinds and quantity of associations that a specified topological type can participate in. A Topology Rule for topological type T is defined by the following 4-tuple:

```
{role, min-cardinality, max-cardinality, associated_topological_type}
```

where

- *role* specifies the role played by the topological type T in the association
- the *minimum* and *maximum* cardinality dictate how many of these kinds of associations can be applied to an entity of this topological type T
- the *associated_topological_type* dictates the type of the entity on the other side of the (binary) association.

In the File System example, two rules would be defined for the *NFS mount* association shown in Figure 2-3 on page 14, one for the topological entity of type *UNIX remote path*, and one for the topological entity of type *exported file system*. Before specifying the rules, the roles that characterize the *NFS mount* association need to be defined. Using *mounter* for the client side and *exporter* for the server side, we have the following topology rule for the *UNIX remote path* topological type:

```
{mounter, 1, 1, exported file system}
```

which indicates that a remote path, if it exists, must be associated with exactly one *exported file system* entity. The corresponding topology rule for the *exported file system* topological type is then:

```
{exporter, 0, *, UNIX remote path}
```

In this case, the existence of the *exported file system* is not dependent on any mounting clients. Also, a maximum cardinality is specified as being infinite. In practice, the implementer of this topology might really want to specify some maximum number to represent a finite limitation on system resources.

2.2.3.2 Further Characteristics

The Topology Service allows topological types to be specialized. For example, the *personal computer* and *mainframe* topological types are specializations of topological type *computer*; an *ethernet LAN card* topological type is a specialization of *network interface card*. A particular type may be a specialization of multiple parent types.

The Topology Service considers that a specialized type will inherit all the characteristics of its more generalized types. Therefore, an entity of a specialized type must not only obey the rules defined specifically for that type, but also all the topological rules defined for its parent types.

2.2.3.3 Summary of Topology Constraints

There are 3 types of constraints which the Topology Service enforces on a topology. These are:

- configurable constraints
- built-in constraints
- extensible constraints.

Configurable constraints are Topology Rules as described above, which are configured by the developer and enforced directly by the Topology Service.

Built-in constraints are built into Topology and require that each entity be uniquely represented by one topological type, and that each entity belong to only one AE. This ensures uniqueness of each entity represented in Topology and that all associations in which an entity participates are readily accessible and apparent (not hidden or difficult to find).

Extensible constraints allow the developer to supplement built-in and configurable constraints with additional semantics that cannot be expressed using topology rules. These extensible constraints are supported by allowing the developer to supply code in the form of object methods, called *enforcers*, that implement the supplementary constraints. When changes to topology data are proposed by any client, the Topology Service invokes any registered enforcers that apply to validate the changes. Only fully-validated changes are allowed to proceed.

2.2.4 Object Mode

Figure 2-5 describes the relationships between the concepts used within Topology. This is expressed as a *fusion* (see Section 2.2.4.1 on page 18) object model. Note that this form of expression does not imply or require that these objects exist as CORBA objects in an implementation.

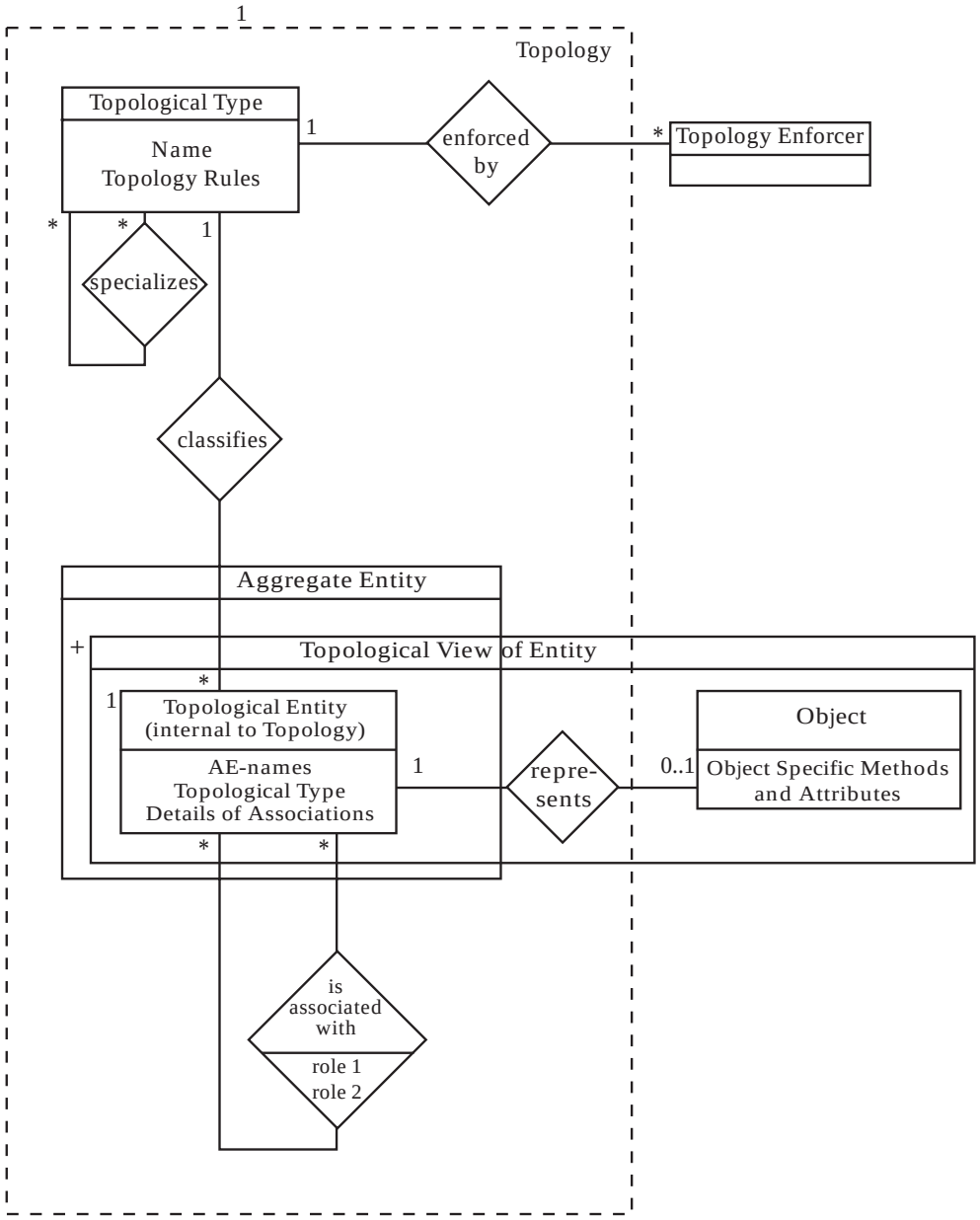


Figure 2-5 Topology Object Model

2.2.4.1 Fusion Method of Object Modelling

The topology object model (see Figure 2-5 on page 17) is based on the *fusion* method of object modelling, as explained in referenced document **OODFM**. Conventions in the Fusion method include the representations shown in Figure 2-6.

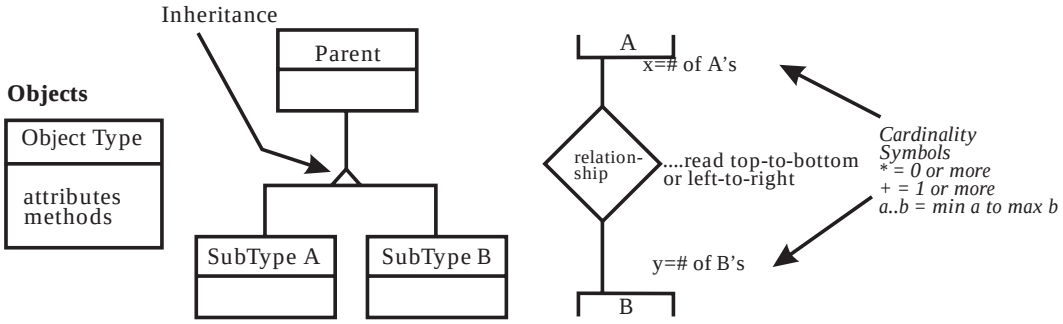


Figure 2-6 Conventions in Fusion Object Modelling Method

2.2.5 Topology Queries

Topology allows a user to make arbitrary queries of topological data. A query always returns a boolean value indicating whether or not a match was found and optionally the actual AEs that matched the query.

A query is expressed in the form of a *navigation path* which is used by Topology to navigate through the topology graph in search for matching AEs. A navigation path includes one or more *AE-patterns* interconnected by *association-patterns*. *association-pattern*.

An *AE-pattern* is a string expression of AE selection criteria. The AE-pattern selection criteria consists of an AE name or an expression of topological types.

An *association-pattern* interconnects AE-patterns to designate the criteria for selecting associations while navigating through the topology graph from one AE to another. An association-pattern uses an expression of role names as a basis for selecting associations.

Textual examples of queries which can be formulated include:

- *''get all the laser printers that are associated via serial cables with this computer''*
  (assuming that *laser printer*, *computer* and *serial cable* are types of AEs whose valid associations conform to a topology managed by the Topology Service)
- *''list all the programs that can be executed by the users who can execute Program A''*
  (assuming that a *Program A* is an existing topological entity and that *person* and *can execute* are AEs whose valid associations conform to a topology managed by the Topology Service).

Queries are expressed textually using a *Topology Query Language (TQL)*. Queries can be parameterized so their execution point in the topology graph can be determined at execution time.

2.3 Development Process

2.3.1 Service Development Process

Figure 2-7 shows a typical process for developing a sub-product which uses the Topology Service.

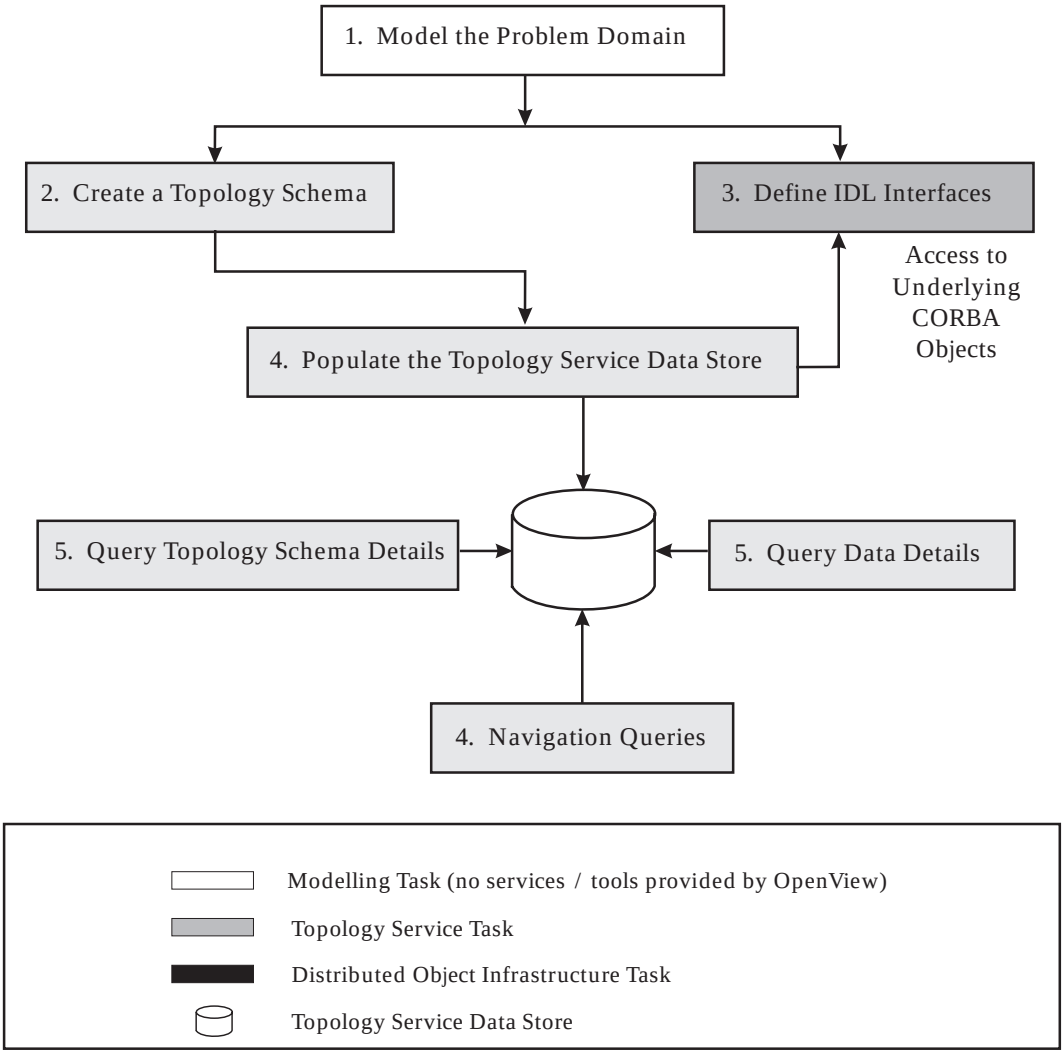


Figure 2-7 Typical Topology Service Development Process

2.3.2 Creating a Topology Schema

Figure 2-8 shows a typical process for creating a topology schema for a client which uses the Topology Service.

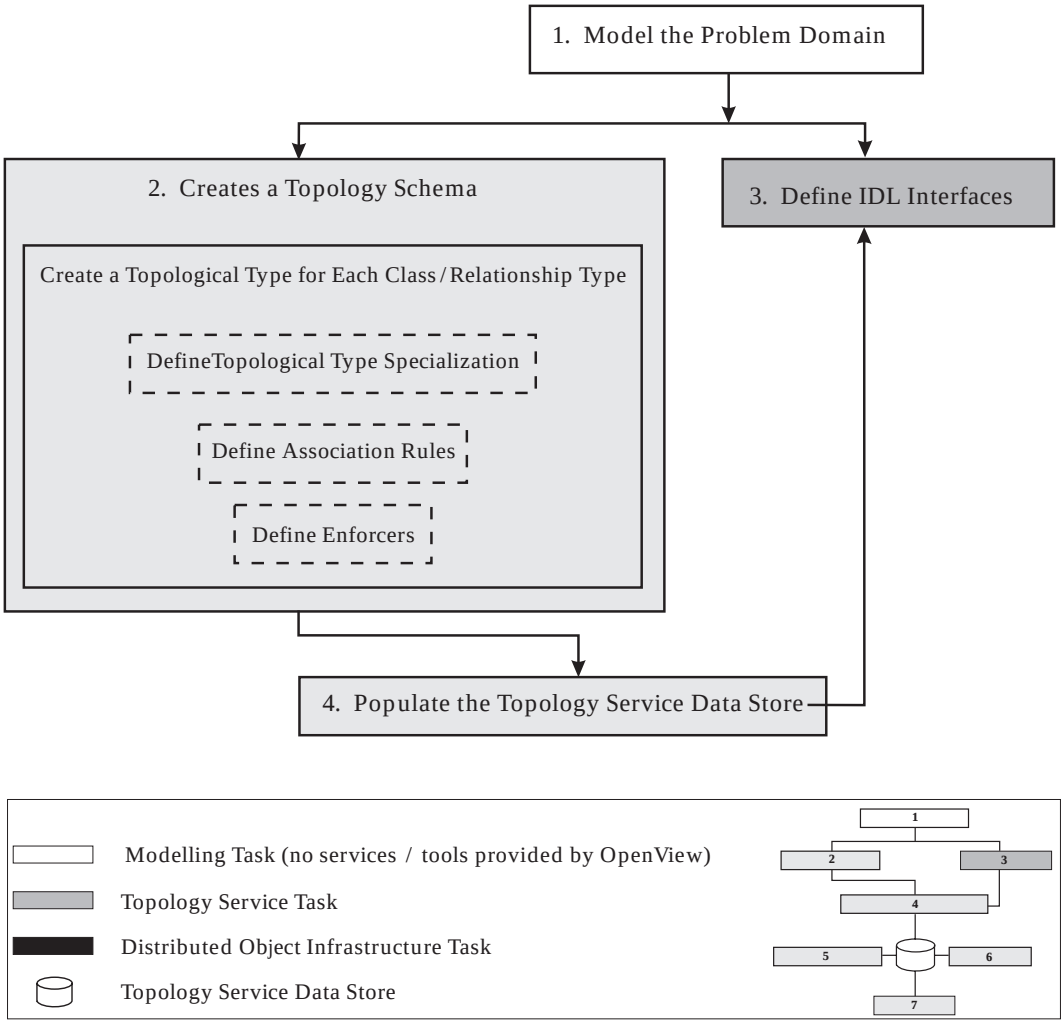


Figure 2-8 Typical Process for Creating a Topology Schema

2.3.3 Typical Process for Defining IDL Interfaces

Figure 2-9 shows a typical process for defining IDL interfaces for a client which uses the Topology Service.

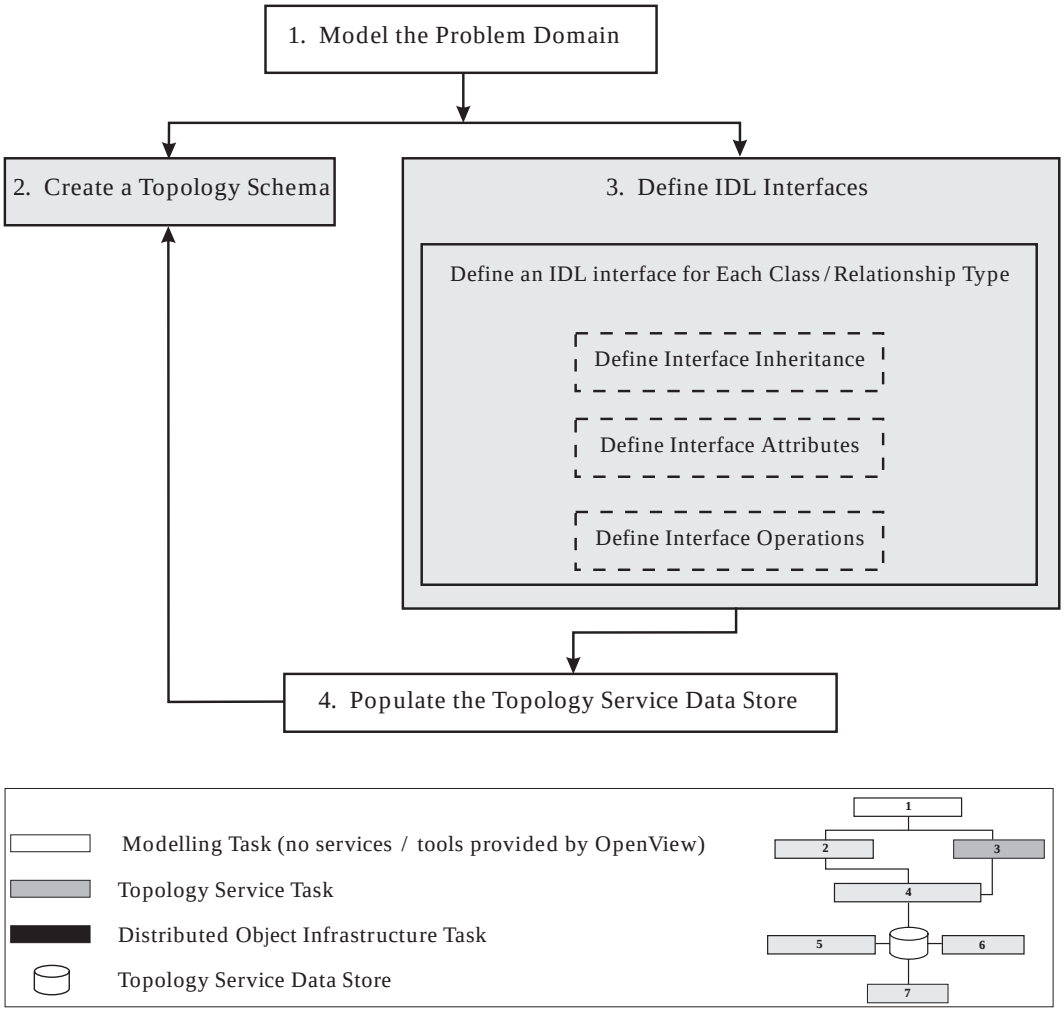


Figure 2-9 Typical Process for Defining IDL Interfaces

2.3.4 Populating the Topology Service Data Store

Figure 2-10 shows a typical process for populating the Topology Service data store.

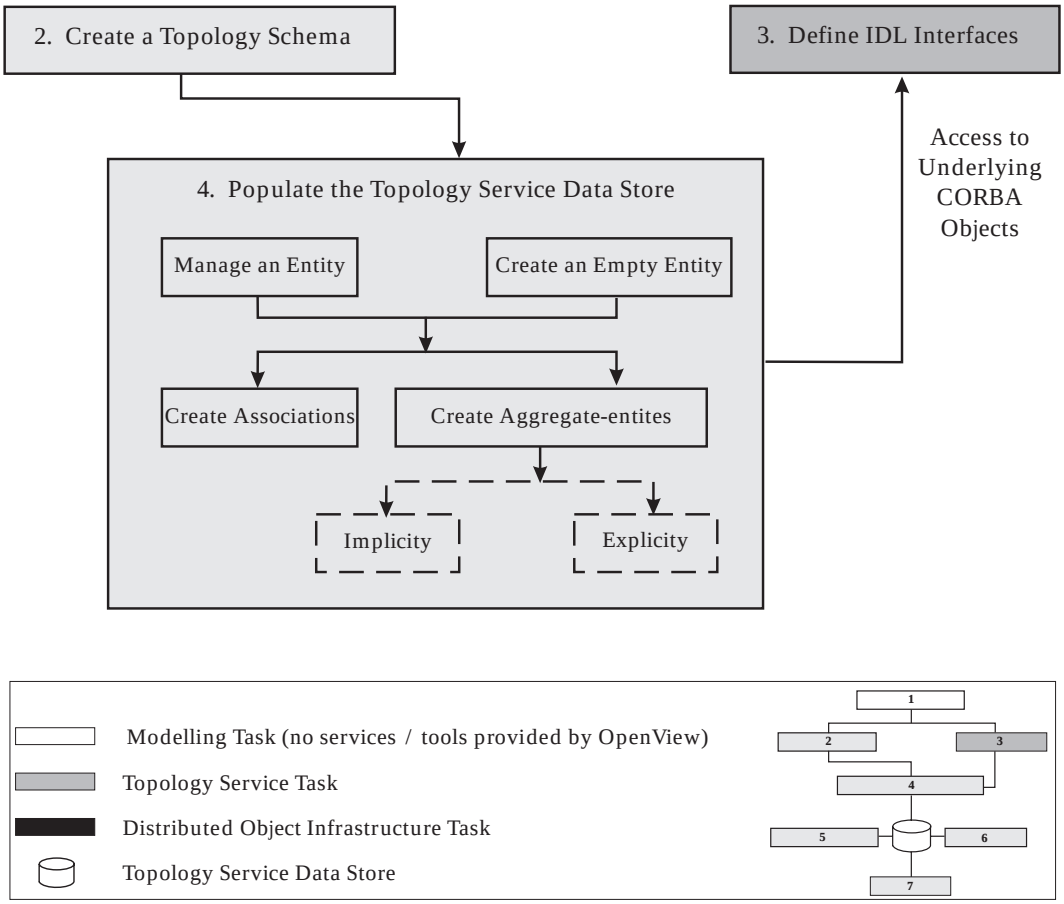


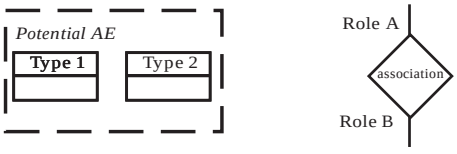
Figure 2-10 Typical Process for Populating Topology Service Data Store

2.4 Development Example

As an example for defining a topology to be managed by the Topology Service, consider an extension to the File System example which incorporates PC NFS clients. Figure 2-11 on page 24 shows an example subgraph of what such a set of topologies might look like within the Topology Service. Note that the example assumes some extensions to the object model, which includes new topological types (for example, PC Box, DOS Disk hfs, DOS network drive, and a Windows95 Operating System). A corresponding object model on which the example topologies are based is shown in Figure 2-12 on page 26. [4]

4. This object model is also based on the *fusion* method of object modelling, as explained in referenced document **OODFusion**. The following extensions to fusion are used in the figure in this footnote, since AEs and association roles are not fusion concepts.

Extensions to Fusion for the Example



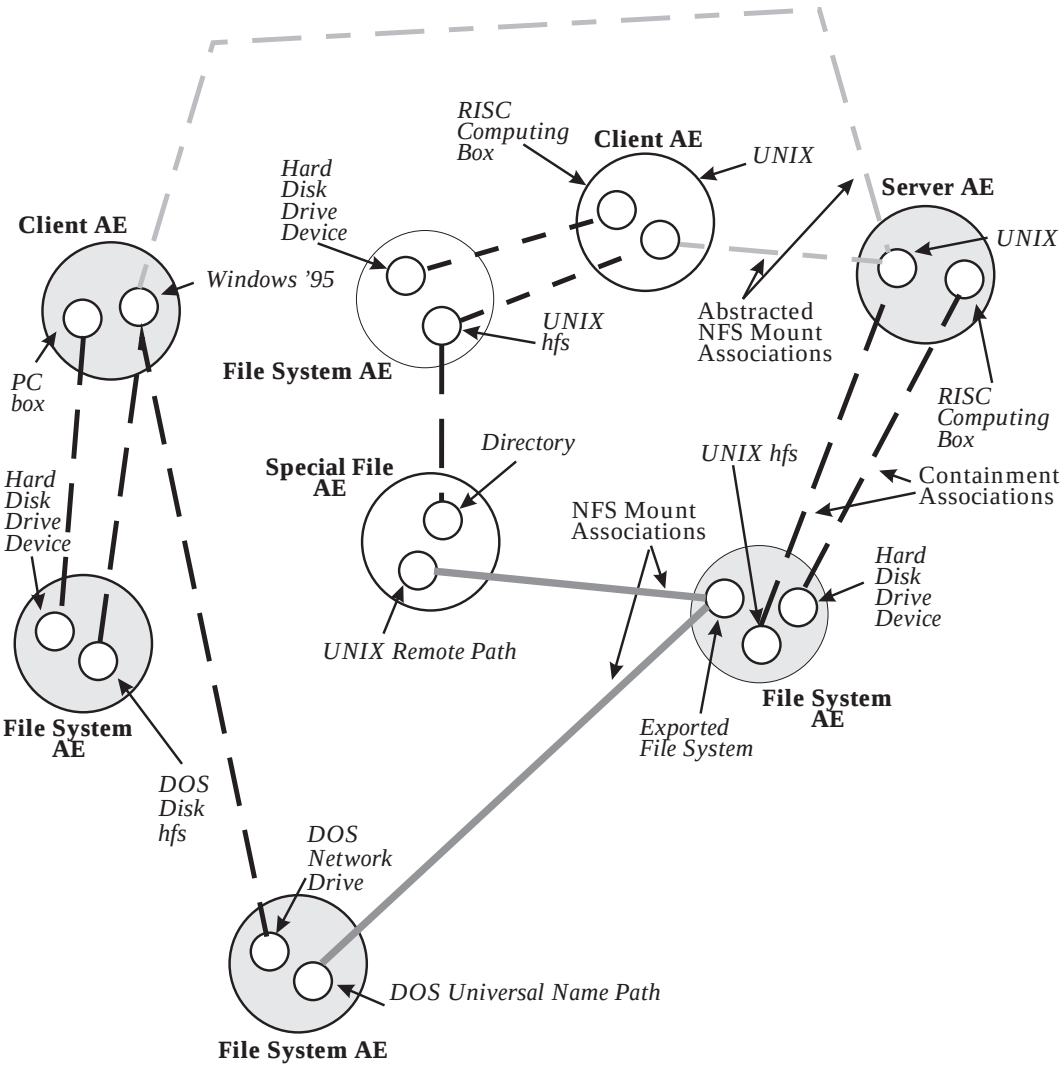


Figure 2-11 Topological Entities and Associations for PC NFS Example

In developing the example, we establish topology rules for two topologies: NFS, and OS-hierarchical file systems. The topological types these entail include:

- OS subtypes:
  - — HP-UX
  - — Windows '95
- logical volume and its subtypes:
  - — UNIX hfs
  - — DOS disk hfs
  - — DOS network drive

- file system path and its subtypes:
  - directory
  - special file subtypes:
    - remote path (UNIX)
    - universal name path (DOS)
- exported file system
- physical computing equipment
  - RISC computing box
  - PC box
- physical peripheral devices
  - hard disk drive.

These are shown as shaded in Figure 2-12.

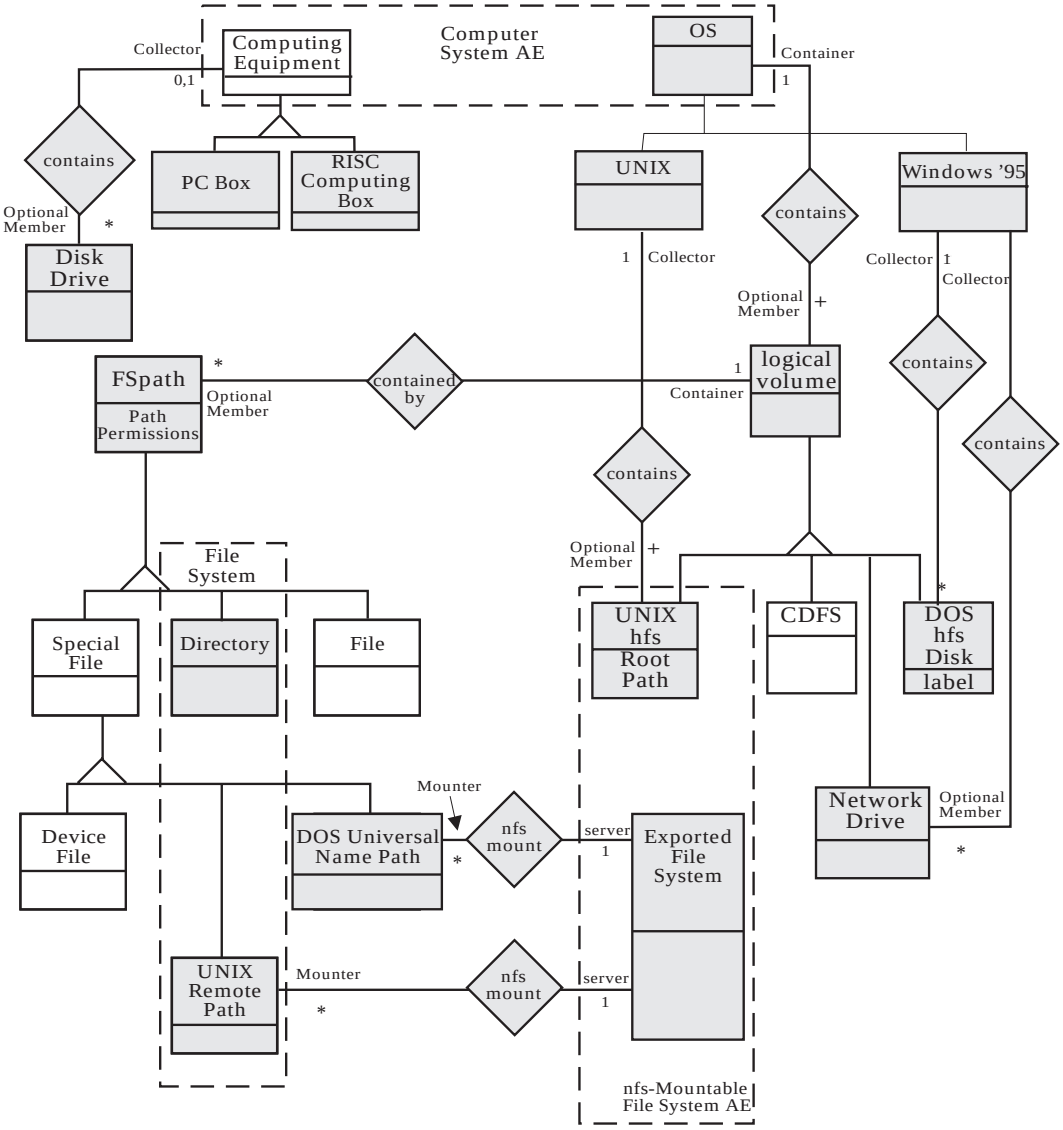


Figure 2-12 Portion of Object Model for Development Example

Once the object model is defined, with association roles and cardinality labeled, the topology rules can be written down. The rules are developed for each topological type to express the kinds of associations that topological type can participate in.

Each topology rule specifies a role name, and a minimum and maximum number of associations in which the entity can participate in that role, along with the topological type of the *associated* entity.

Table 2-1 shows the topology rules used for this example for the eight types of associations (denoted by the diamond-shaped boxes in Figure 2-12).

Association	Topological Type	Topology Rules			
		Role	Min	Max	Associated Entity's Topological Type
1	OS	container	1	infinite	Logical Volume
	Logical Volume	optional member	1	1	OS
2	Windows '95 OS	collector	0	26	DOS disk hfs
	DOS disk hfs	optional member	1	1	Windows '95 OS
3	Windows '95 OS	collector	0	26	DOS network drive
	DOS network drive	optional member	1	1	Windows '95 OS
4	UNIX OS	collector	1	infinite	UNIX hfs
	UNIX hfs	optional member	1	1	UNIX OS
5	Logical Volume	container	0	infinite	FSpath
	FSpath	optional member	1	1	Logical Volume
6	UNIX remote path	mounter	0	1	Exported File System
	Exported File System	server	0	infinite	UNIX remote path
7	DOS universal name path	mounter	0	1	Exported File System
	Exported File System	mounted	0	infinite	DOS universal name path
8	Computing Equipment	collector	0	32	Disk Drive
	Disk Drive	optional member	0	1	Computing Equipment

Table 2-1 Topology Rules

With the Topology Rules defined and made known to the Topology Service, topological associations can be realized. Figure 2-13 shows a sample topology, whose associations are represented in Table 2-2.

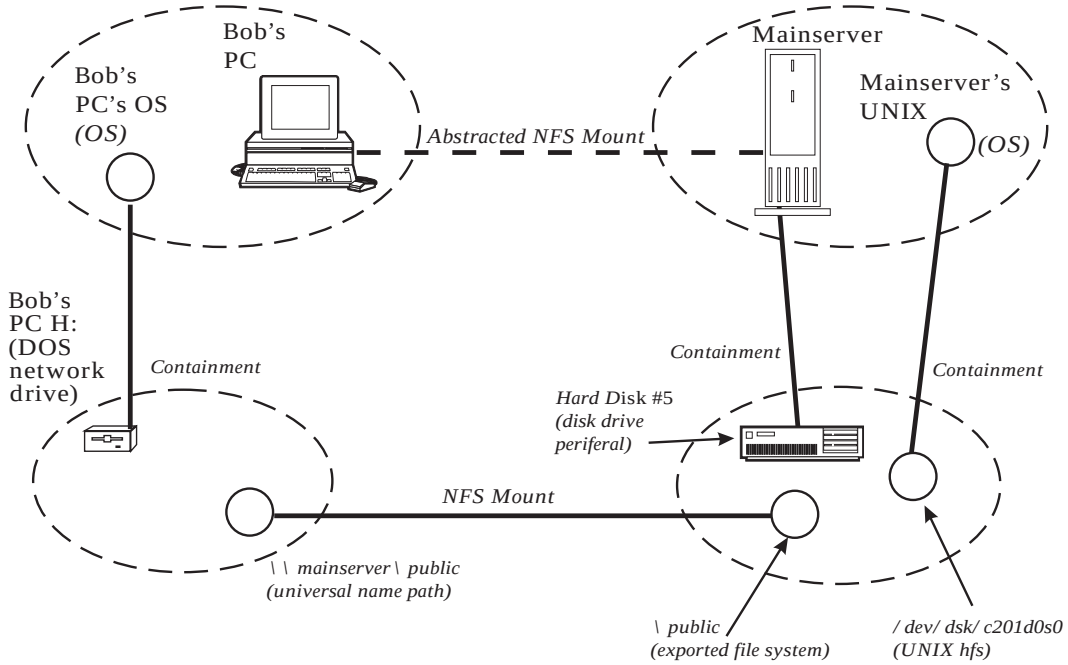


Figure 2-13 Sample Topology with Entities

With the specified constraints, the developer can create the associations identified in Table 2-2.

Entity	Entity's Role	Associated Entity's Role	Associated Entity
Windows '95 OS on Bob's PC	Container	Optional Member	Drive H: on Bob's PC
\\hpcndfs1\public universal name path on Bob's PC	Mounter Server	/public exported file system on hpcndfs1	
/dev/dsk/c201d0s0 on hpcndfs1	Optional Member	Collector	HP-UX OS on hpcndfs1
hard disk drive #5	Optional Member	Collector	mainserver RISC Computing Box

Table 2-2 Valid Associations

Note that the *Abstracted NFS Mount* association between Bob's PC and the main server can be added as well (provided the topological rules are established first[5]). This NFS Mount association is an abstraction derived from underlying physical and logical associations, brought together through AE aggregation.

2.4.1 Adding Enforcers

If it is to be modelled correctly, there is at least one aspect of the NFS topology semantics that the Topology rules cannot adequately describe and hence require an enforcer. Specifically, an *exported file system* entity has as attribute which indicates which remote clients are allowed to establish a mount with it. This attribute consists of a list of hostnames with the designated access each is allowed to have. The Topology rules cannot enforce this, but a registered enforcer can. Note that this check by the enforcer requires information beyond the universal name path and the exported file system, which are directly associated in the NFS Mount. Validating the association for Bob's PC in the example requires following the association from the DOS network drive to its containing OS (and its AE) to determine the hostname for Bob's PC.

2.4.2 Queries

The Topology Service also provides a query language, as noted earlier, which allows Topology clients to obtain topological information (such as the Abstracted NFS Mount association described above and the path from which it is derived). The queries supported provide transparency across AEs and thereby tie separate topologies together into seemingly one, providing an powerful data integration paradigm.

---

5. These would consist of the following:
   - for the Computing Equipment Topological Type (a):
     `\{NFS client, 0, infinite, Computing Equipment\}`
   - for the Computing Equipment Topological Type (b):
     `\{NSF Server, 0, infinite, Computing Equipment\}.`

Chapter 3

Data Definitions

This chapter contains a description of the data definitions used in the Topology Service modules.

3.1 Topology.idl Definitions

```
#if !defined (TOPOLOGY_IDL)
#define TOPOLOGY_IDL
module Topology {

        // NOTE: it is hoped that the PartitionedIdentifier will be
        // incorporated into the standard COS ObjectIdentity module

        struct PartitionedIdentifier {
                string id_context;
                string id_base;
        };

        typedef sequence<PartitionedIdentifier> IdList;

        struct TopologicalEntity {
                PartitionedIdentifier id;
                Object obj;
        };

        typedef sequence <TopologicalEntity> TopologicalEntityList;
        typedef string TopologicalTypeName;
        typedef sequence <TopologicalTypeName> TopologicalTypeNameList;

        struct Association {
                TopologicalEntity entity1;
                string role1;
                TopologicalEntity entity2;
                string role2;
        };

        typedef sequence <Association> AssociationList;

        exception SchemaViolation {};
        exception AggregationConstraintViolation{};
        exception NotManaged{};
        exception TransactionWillRollback{};
        exception ServerNotContactable{};
        exception NoSuchType{};
        exception RoleNotSupported{};
        exception InconsistentData{};
        exception InvalidRoleName{};
        exception InvalidTypeName{};
};
#endif  // TOPOLOGY_IDL
```

Chapter 4

Populating Topology Interface

This chapter describes how to populate the Topology Data Store, and once populated how then to manage it.

All interfaces are expressed in CORBA IDL. They are grouped under TopologyData.idl.

4.1 EntityManager: Managing Entities

These operations are designed to manage the topological data for a collection of entities within the Topology Service.

This interface provides access to the data that the Topology Service maintains about topologically managed entities.

An EntityManager is a collection manager whose components are the *topological aspects* of entities. These *topological aspects* are never accessed directly by users of the Topology Service. Instead an EntityManager acts as a *proxy* to implement the operations that are conceptually performed on the entities themselves.

The entity (or entities) on which an operation is to be performed is identified by a PartitionedIdentifier structure. This structure contains two strings which together uniquely identify the entity and are immutable for this entity. Currently this structure is defined as part of the Topology module, but it is hoped that this will become part of the standard CORBA Services (COS) ObjectIdentity module in the future.

When first announcing the entity to the Topology Service (via the **manage**( ) operation) the entity is indicated by a TopologicalEntity structure which contains a PartitionedIdentitifer and also contains an object reference (which may be nil). Topology will store the object reference and will return it to the clients when referring to the entity, but Topology itself does not examine or use the object reference.

Many of the operations are designed such that it is not an error to do the same operation on the same entity (or entities) that has previously been done. In these cases the operation returns a boolean value which will indicate whether the data was actually changed (TRUE) or not (FALSE).

The purpose of this section is to describe the interfaces associated with managing entities with the Topology Service. Once an entity is managed by the Topology Service, numerous tasks can be performed, such as creating associations between the entity and other entities, and managing the entity as part of an aggregate entity.

If the entity to be managed does not exist (where no underlying implementation exists), it is necessary to create an empty entity — see the reference manual page for **CreateEmptyEntity**( ).

NAME — DESCRIPTION

Manage — This operation is used to manage an existing entity and treat it as a topological entity.

SIGNATURE

```
boolean manage (
                in TopologicalEntity entity,
                in Topology::TopologicalTypeName topo_type)
        raises (TypeChanged,
                Topology::SchemaViolation,
                Topology::InvalidTypeName,
                Topology::NoSuchType,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

entity
The given entity to be managed.

topo_type
The topological type for the new entity, typically, same as the entity name.

Return
A boolean which indicates whether or not this operation modified the topology data.

SEMANTICS

This operation creates a new topological entity of the specified topological type *topo_type*. The topological type name must conform to the Federated Name syntax (see reference **XFN**).

If the entity specified by the *entity* parameter value is already topologically managed, and it is of the same topological type as the one specified by the *topo_type* parameter value, no exception is raised. In this case the operation returns FALSE to indicate that the Topology data store was not altered.

Topology saves the object reference portion of the TopologicalEntity but does not examine it or use it. This object reference may be nil.

EXCEPTIONS RAISED

[TypeChanged]
Raised if this operation attempts to change the type of an entity which is already managed in Topology.

[Topology::InvalidTypeName]
Raised if the given type name does not conform to the X/Open Federated Name syntax.

[Topology::SchemaViolation]
Raised if this operation is not part of a transaction and the type specified for the entity being managed requires that the entity be part of an association.

[Topology::NoSuchType]
Returned if the specified topological_type does not exist.

[Topology::TransactionWillRollback]
Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Unmanage — This operation is used when you want to have Topology stop managing the topological data for an entity.

SIGNATURE

```
boolean unmanage (
            in Topology::PartitionedIdentifier entity)
    raises (Topology::SchemaViolation,
            Topology::ServerNotContactable,
            Topology::TransactionWillRollback);
```

PARAMETERS

Input

entity
The given entity for which management of its topology data is to stop.

Return
A boolean which indicates whether or not this operation modified the topology data.

SEMANTICS

This operation is used to inform Topology to stop managing the topological data for a given entity. This operation deletes the topological data associated with the entity.

If the given entity, *entity* has not been managed created via the operation **manage**( ), then no exception will be raised. In this case, the operation does not modify the data and returns a FALSE to indicate that the topology database was not modified.

If the given entity was constrained to be part of the same aggregate as another entity, the Topology Service will undo (*untie*) this constraint.

EXCEPTIONS RAISED

[Topology::NotManaged]
Raised if the entity specified is not managed by Topology.

[Topology::SchemaViolation]
Raised if this operation is not part of a transaction and the operation results in a conflict between the Topology Schema definition and the new state of the topology data. This occurs if the Topology Type rules require that an entity be part of an association, but the operation unmanaged an entity which was participating in one such association (thereby removing the association).

[Topology::TransactionWillRollback]
Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Create_Empty_Entity — This operation is used to create an ‘‘empty-entity’’. Typically this is for managing *relationship types* where there are no actual entity implementations behind these Topology Service internal structures.

SIGNATURE

```
Topology::PartitionedIdentifier
create_empty_entity (
                in Topology::TopologicalTypeName topo_type)
        raises (Topology::NoSuchType,
                Topology::InvalidTypeName,
                Topology::SchemaViolation,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

topo_type
The topological type representing this entity.

Return

This operation returns a *PartitionedIdentifier* which identifies the internal Topological entity structure.

SEMANTICS

This operation creates an empty entity, that is, a topologically managed entity which has no underlying object implementation. The empty entity becomes an instance of the topological type specified by the *topo_type* parameter value.

EXCEPTIONS RAISED

[NoSuchType]
Returned if the specified topological_type does not exist.

[Topology::SchemaViolation]
Raised if this operation is not part of a transaction and the type specified for the entity being managed requires that the entity be part of an association.

[Topology::TransactionWillRollback]
Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

[ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Associate — This operation is used to create an association between two entities..

SIGNATURE

```
boolean associate (
                in Topology::Association assoc );
        raises (Topology::SchemaViolation,
                Topology::NotManaged,
                Topology::InvalidRoleName,
                Topology::RoleNotSupported,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

assoc

The structure that includes the TopologicalEntity identifiers and their roles in the association being proposed.

Output/Return

A boolean which indicates whether or not this operation modified the topology data.

SEMANTICS

This operation forms a topological association between the two entities specified in *assoc*.

An association is specified by two sets of `<entity, role>` pairs.

If the association specified in *assoc* already exists, no error is returned. In this case, the operation does not modify the data and returns FALSE.

The object reference in the TopologicalEntity identifiers (passed in the Association structure) is disregarded for this operation and may be nil.

EXCEPTIONS RAISED

[Topology::NotManaged]

Raised if one of the entities specified in the association is not managed by Topology.

[Topology::SchemaViolation]

Raised if this operation is not part of a transaction and the operation results in a conflict between the Topology Schema definition and the new state of the topology data. This occurs if the Topology Type rules express a maximum cardinality for an association but the operation adds an association which exceeds this maximum.

[Topology::InvalidRoleName]

Raised if a role specified in the association does not conform to the X/Open Federated Name syntax.

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable

[Topology::RoleNotSupported]

Raised if a role which was specified which is not supported by the entities.

[Topology::TransactionWillRollback]

Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

NAME — DESCRIPTION

Disassociate — This operation is used to delete an association between two entities.

SIGNATURE

```
boolean disassociate (
                in Topology::Association assoc)
        raises (Topology::NotManaged,
                Topology::InvalidRoleName,
                Topology::SchemaViolation,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

assoc

The structure that includes the entities and the roles within an association which are to be deleted.

Return

A boolean which indicates whether or not this operation modified the topology data.

SEMANTICS

This operation destroys a topological assocation between the two entites specified in the *assoc* parameter.

If the association specified in *assoc* did not previously exist, no error is returned. In this case, the operation does not modify the data and returns a FALSE.

The object reference in the TopologicalEntity identifiers (passed in the Association structure) is disregarded for this operation and may be nil.

EXCEPTIONS RAISED

[Topology::NotManaged]

Raised if one of the entities specified in the association is not managed by Topology.

[Topology::InvalidRoleName]

Raised if a role specified in the association does not conform to the X/Open Federated Name syntax.

[Topology::SchemaViolation]

Raised if this operation is not part of a transaction and the operation results in a conflict between the Topology Schema definition and the new state of the topology data. This occurs if the Topology Type rules require that an entity be part of an association, but the operation removed an association which the rules require.

[Topology::TransactionWillRollback]

Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Tie_Entities — This operation is used to constrain two entities so that they are forced to represent the same Aggregate Entity (AE).

SIGNATURE

```
boolean tie_entities (
                in AggregationTie tie)
        raises (Topology::NotManaged,
                Topology::AggregationConstraintViolation,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

tie

Indicates the two entities to be tied and a string identifier to distinguish the tie from other ties between these two entities.

Output/Return

A boolean which indicates whether or not this operation modified the topology data.

SEMANTICS

This operation constrains two given entities, to represent the same Aggregate Entity (AE).

If the tie (between these two entities and with the given identifier) already exists no exception is raised. In this case, the operation does not modify the data and returns FALSE to indicate that the data was not modified.

The object reference in the TopologicalEntity identifiers (passed in the AggregationTie structure) is disregarded for this operation and may be nil.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]

Raised if one of the entities specified in the tie is not managed by Topology.

[Topology::AggregationConstraintViolation]

Returned if this operation would result in an Aggregate Entity containing two entities with the same Topological Type, or two entities where the type of one is a child of the type of another.

[Topology::TransactionWillRollback]

Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

NAME — DESCRIPTION

Untie_Entities — This operation is used to stop two entities from being constrained so that they are no longer forced to represent the same Aggregate Entity(AE).

SIGNATURE

```
boolean untie_entities (
                in AggregationTie tie)
        raises (Topology::NotManaged,
                Topology::TransactionWillRollback,
                Topology::ServiceNotContactable);
```

PARAMETERS

Input

tie

Indicates the two entities to be tied and a string identifier to distinguish the tie.

Output/Return

A boolean which indicates whether or not this operation modified the topology data.

SEMANTICS

This operation removes the specified tie from constraining the two entities to represent the same Aggregate Entity(AE). The tie to be removed is indicated by the two entities and the identifier.

If the entities were not previously constrained to represent the same AE, no error is returned. In this case, the operation does not modify the data and returns FALSE.

The object reference in the TopologicalEntity identifiers (passed in the AggregationTie structure) is disregarded for this operation and may be nil.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]

Raised if one of the entities specified in the tie is not managed by Topology.

[Topology::TransactionWillRollback]

Raised if the operation is being performed within a transaction and inconsistencies are discovered in the current data. Topology will not let the transation successfully commit.

NAME — DESCRIPTION

In_Same_Aggregate — This operation is used to verify whether a list of entities belong to the same Aggregate Entity.

SIGNATURE

```
boolean in_same_aggregate (
                in Topology::IDList entity_list)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

entity_list
List of entity identifiers.

Return

A boolean which indicates whether the entities belong to the same Aggregate Entity.

SEMANTICS

If all of the entities in the entity_list belong to the same Aggregate Entity, then a TRUE is returned. If any one of the entities in the *entity_list* do not belong to the same Aggregate Entity then FALSE is returned.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]
Returned if any one or more of the entities in *entity_list* are not managed by Topology.

NAME — DESCRIPTION

Is_Managed — This operation is used to determine whether or not a certain entity is managed by Topology.

SIGNATURE

```
boolean is_managed (
                in Topology::PartitionedIdentifier entity)
        raises (Topology::ServerNotContactable);
```

PARAMETERS

Input

entity

The given entity to be verified if managed by Topology.

Return

A boolean which indicates whether the entity is managed by Topology.

SEMANTICS

If the entity is managed by Topology a TRUE is returned, else a FALSE is returned.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

State_Has_Changed — This operation is used to advise the Topology Service that the non-topological state of an entity has changed.

SIGNATURE

```
void state_has_changed (
                in Topology::PartionedIdentifier entity)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

entity

The given entity whose non-topological state has changed.

SEMANTICS

An entity with an enforcer which is interested in its non-topological state must arrange for the correct invocation of this operation. Entities using this operation allow greater integration with other sub-products.

This allows an application to place appropriate code for verifying the semantics of a relationship where it belongs — that is, in an enforcer. For example, an application which manages relationships between modems might want to ensure that the baud rate of communicating modems are the same. It can have an enforcer which gets triggered whenever the modems are associated or when the modems have a state change (when their baud rate is changed). The enforcer can then verify that the two modems have the same baud rate. However, since Topology cannot on its own be aware of state changes it must be informed of them via this call.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]

Raised if the entity specified is not managed by Topology.

NAME — DESCRIPTION

Get_Associations — This operation is used to get the associations that involve an entity.

SIGNATURE

```
Topology::AssociationList
get_associations (
                in Topology::PartitionedIdentifier entity)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

entity
The entity for a list of associations is wanted.

Return

A list of associations, *Topology::AssociationList*, that involve the given entity, *entity*.

SEMANTICS

This operation returns a list of associations, *Topology::AssociationList* that represents all the associations that involve the entity.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]
Raised if the entity specified is not managed by Topology.

NAME — DESCRIPTION

Get_Topological_Type — This operation gets the topological type for a given entity.

SIGNATURE

```
Topology::TopologicalTypeName
get_topological_type (
                in Topology::PartitionedIdentifier entity)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

entity
The entity of interest.

Return

The topological type, *Topology::TopologicalTypeName*, of the entity.

SEMANTICS

This operation gets the topological type of the topological entity specified by the *entity* parameter value.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]
Raised if the entity specified is not managed by Topology.

NAME — DESCRIPTION

Get_Ties — Get the list of aggregation ties that involve an entity.

SIGNATURE

```
AggregationTieList
get_ties (
                in Topology::PartitionedIdentifier entity)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

entity
The entity of interest.

Return

Returns a list of ties that involve the given entity. If the entity is not tied to any other entities then the returned list is empty.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]
Raised if the entity specified is not managed by Topology.

4.2 Managing Aggregate Entities Interface

This set of operations provides access to the data that the Topology Service maintains about Aggregate Entities. The Aggregate Entity Manager, *AEManager*, is a collection manager whose components are Aggregate Entities. These Aggregate Entities are never accessed directly by users of the Topology Service, instead an AEManager acts as a *proxy* to implement the operations that are conceptually performed on individual Aggregate Entities. The Aggregate Entity on which the operation is to be performed is identified by a PartitionedIdentifier of any one of the entities of the Aggregate.

NAME — DESCRIPTION

Get_Entities — This operation gets the topological entities that are part of an Aggregate Entity.

SIGNATURE

```
Topology::TopologicalEntityList
get_entities (
                in Topology::PartitionedIdentifier ae)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

ae

An identifier for an entity within the Aggregate Entity of interest.

Return

A list of entity identifiers which are in the Aggregate Entity.

SEMANTICS

This operation gets the topological entities that are part of the Aggregate Entity specified by the *ae* parameter value. The value of the *ae* parameter may identify any one of the entities in the Aggregate Entity. The list will always include the entity identifier of the input parameter.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]

Raised if the entity specified is not managed by Topology.

NAME — DESCRIPTION

Get_Topological_Types — This operation gets the topological types of the entities that are part of an Aggregate Entity.

SIGNATURE

```
Topology::TopologicalTypeNameList
get_topological_types (
                in Topology::PartitionedIdentifier ae)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

ae

An entity identifier within the Aggregate Entity.

Return

A list of Topological Types.

SEMANTICS

This operation gets the topological types of the topological entities that are part of the Aggregate Entity specified by the *ae* parameter value. The value of the *ae* parameter may identify any one of the entities in the aggregate.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]

Raised if the entity specified is not managed by Topology.

NAME — DESCRIPTION

Get_Entity_of_Type — This operation gets the entity of a particular topological type which is part of an Aggregate Entity.

SIGNATURE

```
Topology::TopologicalEntity
get_entity_of_type (
                in Topology::PartitionedIdentifier ae,
                in Topology::TopologicalTypeName topo_type)
        raises (Topology::ServerNotContactable,
                Topology::NotManaged);
```

PARAMETERS

Input

ae
An entity within an Aggregate Entity.

topo_type
The topological type of the entity being sought.

Return
The identifier of an entity with the specified topological type.

SEMANTICS

This operation gets the entity of a particular topological type which is part of the Aggregate Entity specified by the *ae* parameter value. The value of the *ae* parameter may identify any one of the entities in the Aggregate Entity. The operation will only get an entity of the specified topological type *topo_type*, and not any specializations of that topological type.

If an entity of the topological type specified by the *topo_type* parameter value is found, the entity identifier is returned.

If no entity of the topological type specified by the *topo_type* parameter value is found, then the *PartitionedIdentifier* will contain empty strings and the object reference will be nil.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

[Topology::NotManaged]
Raised if the entity specified is not managed by Topology.

4.3 TopologyData.idl File

```
#if !defined (TOPOLOGYDATA_IDL)
#define TOPOLOGYDATA_IDL

#include <ErrorService.idl>
#include <Topology.idl>

module TopologyData {

        interface EntityManager: CosTransactions::TransactionalObject, Entity {
                // Operations which deal with a collection of entities

                boolean manage (
                                in Topology::TopologicalEntity entity,
                                in Topology::TopologicalTypeName topo_type)
                        raises (TypeChanged,
                                Topology::SchemaViolation,
                                Topology::InvalidTypeName,
                                Topology::NoSuchType,
                                Topology::TransactionWillRollback,
                                Topology::ServerNotContactable);

                boolean unmanage (
                                in Topology::PartitionedIdentifier entity)
                        raises (Topology::SchemaViolation,
                                Topology::ServerNotContactable,
                                Topology::TransactionWillRollback);

                Topology::PartitionedIdentifier
                create_empty_entity (
                                in Topology::TopologicalTypeName topo_type)
                        raises (Topology::NoSuchType,
                                Topology::InvalidTypeName,
                                Topology::SchemaViolation,
                                Topology::TransactionWillRollback,
                                Topology::ServerNotContactable);

                boolean associate (
                                in Topology::Association assoc),
                        raises (Topology::SchemaViolation,
                                Topology::NotManaged,
                                Topology::InvalidRoleName,
                                Topology::RoleNotSupported,
                                Topology::TransactionWillRollback,
                                Topology::ServerNotContactable);

                boolean disassociate (
                                in Topology::Association assoc)
                        raises (Topology::NotManaged,
                                Topology::InvalidRoleName,
                                Topology::SchemaViolation,
                                Topology::TransactionWillRollback,
                                Topology::ServerNotContactable);

                boolean tie_entities (
                                in AggregationTie tie)
                        raises (Topology::NotManaged,
                                Topology::AggregationConstraintViolation,
```

```
                    Topology::TransactionWillRollback,
                    Topology::ServerNotContactable);

        boolean untie_entities (
                    in AggregationTie tie)
                raises (Topology::NotManaged,
                    Topology::TransactionWillRollback,
                    Topology::ServerNotContactable);

        boolean in_same_aggregate (
                    in Topology::IDList entity_list)
                raises (Topology::ServerNotContactable,
                    Topology::NotManaged);

        // Operations for which the EntityManager acts as a proxy
        // for an individual entity:

        boolean is_managed (
                    in Topology::PartitionedIdentifier entity)
                raises (Topology::ServerNotContactable);

        void state_has_changed (
                    in Topology::PartitionedIdentifier entity)
                raises (Topology::ServerNotContactable,
                    Topology::NotManaged);

        Topology::AssociationList
        get_associations (
                    in Topology::PartitionedIdentifier entity)
                raises (Topology::ServerNotContactable,
                    Topology::NotManaged);

        Topology::TopologicalTypeName
        get_topological_type (
                    in Topology::PartitionedIdentifier entity)
                raises (Topology::ServerNotContactable,
                    Topology::NotManaged);

        AggregationTieList
        get_ties (
                    in Topology::PartitionedIdentifier entity)
                raises ( Topology::ServerNotContactable,
                    Topology::NotManaged);
};

interface AEManager: CosTransactions::TransactionalObject {


        // Operations for which the EntityManager acts as a proxy
        // for an individual aggregate-entity

        Topology::TopologicalEntityList
        get_entities (
                    in Topology::PartitionedIdentifier ae)
                raises (Topology::ServerNotContactable,
                    Topology::NotManaged);

        Topology::TopologicalTypeNameList
        get_topological_types (
```

```
                                in Topology::PartitionedIdentifier ae)
                        raises (Topology::ServerNotContactable,
                                Topology::NotManaged);

                Topology::TopologicalEntity
                get_entity_of_type (
                                in Topology::PartitionedIdentifier ae,
                                in Topology::TopologicalTypeName topo_type)
                        raises (Topology::ServerNotContactable,
                                Topology::NotManaged);
        };
};
#endif  //TOPOLOGYDATA_IDL
```

Chapter 5

Managing Topological Types

This chapter describes how to manage a collection of Topological Types.

All interfaces are expressed in CORBA IDL. They are grouped under TopologyMetaData.idl.

5.1 TopologicalTypeManager

This interface provides access to the data that the Topology Service maintains about topological types. These topological types are never accessed directly by users of the Topology Service. Instead a TopologicalTypeManager acts as a *proxy proxy* to implement the operations that are conceptually performed on the individual topological types themselves.

The topological type on which the operation is to be performed is identified by its topological type name. By convention, this name is passed as a first parameter to the *proxy* operation on the TopologicalTypeManager.

Although a topological type is not an entity, this interface provides an operation that returns an entity identifier for a given topological type name. This allows the optional use of an event service to generate notifications whose source is a topological type.

NAME — DESCRIPTION

Create — This operation is used to create a new topological type.

SIGNATURE

```
boolean create (
                in Topology::TopologicalTypeName type_name,
                in Topology::TopologicalTypeNameList  parent_types);
        raises (Toplogy::InvalidTypeName,
                Toplogy::NoSuchType,
                ToplogyMetaData::TypeRedefinition,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The name of the topological type to be created.

parent_types
The name of the parent topological types.

Output/Return
A boolean which indicates whether or not a topological type was actually created as a result of this operation.

SEMANTICS

This operation creates a topological type with a topological type name as specified by the value of the *type_name* parameter and the parent topological types as specified by the value of the *parent_types* parameter.

Topological types should be identified by the interface type of the corresponding entity interface type of the entities that will be managed by Topology (in the IDL form **module::interface**).

If the topological type specified by the *topo_type* parameter value already exists, and has the same parent topological types as those specified by the *parent_types* parameter value, then no error is returned. In this case the operation returns a FALSE to indicate that the Topology metadata store was not altered, else it returns a TRUE.

EXCEPTIONS RAISED

[Toplogy::InvalidTypeName]
Returned if any of the given type names in the *type_name* and *parent_types* parameter values are not valid topological type names.

[Toplogy::NoSuchType]
Returned if the topological type names specified by the *parent_types* parameter value are not existing topological types.

[ToplogyMetaData::TypeRedefinition]
Returned if passed the name of an existing topological type as the value of the *type_name* parameter and which has different parents from those specified in the *parent_types* parameter value.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Dismiss — This operation is used to notify the Topology Service that an application no longer wishes to support a topological type.

SIGNATURE

```
boolean dismiss (
        in Topology::TopologicalTypeName type_name)
    raises (Topology::InvalidTypeName,
        Topology::TransactionWillRollback,
        Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The name of the topological type to be deleted.

Output/Return

A boolean which indicates whether or not the topological type was actually removed.

SEMANTICS

This operation notifies the Topology Service that a client is no longer interested in a topological type. The Topology Service will remove the topological type provided:

- there are no entities that are managed as instances of the topological type
- the topological type is not a parent of any other topological types.

If the topological type specified by the *type_name* parameter value does not exist, no error is returned. In this case, the operation returns FALSE, else it returns a TRUE.

EXCEPTIONS RAISED

[Toplogy::InvalidTypeName]
Returned if the given type name is not a valid topological type name.

[Toplogy::TypeInUse]
Returned if the given type has instances or other types which inherit from it.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Exists — This operation is used to check whether a topological type exists.

SIGNATURE

```
boolean exists (
                in Topology::TopologicalTypeName type_name)
        raises (Topology::InvalidTypeName,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name

The name of the topological type to be checked.

Output/Return

A boolean which indicates whether or not a topological type with this name exists.

SEMANTICS

This operation returns a TRUE if a topological type exists with the same name specified by the *type_name* parameter value, otherwise it returns FALSE.

EXCEPTIONS RAISED

[Toplogy::InvalidTypeName]

Returned if any of the given type name is not a valid topological type name.

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_All_Types — This operation is used to get all of the topological types which exist in the installation.

SIGNATURE

```
Topology::TopologicalTypeNameList
get_all_types()
        raises (Topology::ServerNotContactable);
```

PARAMETERS

Input

There are no input parameters for this operation.

Output/Return

A list of topological type names.

SEMANTICS

This operation gets a list of all topological types which exist in the installation.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_Parents — This operation is used to get the names of the parents of a topological type.

SIGNATURE

```
Topology::TopologicalTypeNameList
get_parents (
                in Topology::TopologicalTypeName type_name)
        raises (Topology::InvalidTypeName,
                Topology::NoSuchType,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The name of the topological type to be checked.

Output/Return

A list of names of the parent topological types.

SEMANTICS

This operation gets the names of the parents for the specified topological type, *type_name* parameter value. By parents, it is meant just the immediate parents of the topological type rather than all of the topological types from which it inherits.

EXCEPTIONS RAISED

[Toplogy::InvalidTypeName]
Returned if the given type name is not a valid topological type name.

[Topology::NoSuchType]
Returned if the given topological type name does not exist.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Define_Rules — This operation is used to define or redefine the association rules for a topological type.

SIGNATURE

```
void define_rules (
                in Topology::TopologicalTypeName type_name,
                in AssociationRuleList rules)
        raises (TopologyMetaData::InvalidTypeModification,
                Topology::SchemaViolation,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The topological type whose rules are to be defined.

rules
The association rules for the given topological type.

Output/Return
There are no output or return parameters for this operation.

SEMANTICS

This operation is used to define the initial association rules for the topological type specified by the *type_name* parameter value, or if association rules already exist for the topological type, this operation replaces them with the newly defined rules.

For each association rule in the input list the structure **AssocationRule** allows you to specify:

- the role of the topological type
- the name of the topological type with which instances of this topological type can be associated
- the minimum number of associations allowed with an instance of the associated topological type
- the maximum number of associations allowed with an instance of the associated topological type.

A type is considered ‘‘in use’’ if there are in existence instances of that type, or if another type inherits from it:

- rules can be added to a type
- a rule which is defined in a parent type can never be changed/redefined
- if the type is ‘‘in use’’ then rules cannot be removed or modified.

If this operation is invoked within a transaction and the Topological Type is ‘‘in use’’ and the effect would be to remove or modify existing rules, this operation raises no exception. However, at the end of the transaction, the Topology Data Service:

- rolls back the transaction
- writes the message associated within the exception
  [TopologyMetaData::InvalidTypeModification]
  to the error log.

If this operation is invoked outside of a transaction, the Topology Data Service itself starts a transaction, performs the operation, and closes the transaction. In this case, if the Topological Type is ‘‘in use’’ and the effect of this operation would be to remove or modify existing rules, this operation fails and raises the exception [TopologyMetaData::InvalidTypeModification].

EXCEPTIONS RAISED

[TopologyMetaData::InvalidTypeModification]
Returned if the effect of this operation would be to remove or modify existing rules.

[Topology::InvalidTypeName]
Returned if the type specified is not a valid topological type name.

[Topology::NoSuchType]
Returned if there is no type defined in the Topology store that matches the given type name.

[Topology::SchemaViolation]
Returned if there is a conflict between the Topology Schema definition and the persistent storage of the topology data, known as Topology data store.

[Topology::ServerNotContacted]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Define_Enforcers — This operation is used to define or redefine the enforcers for a topological type.

SIGNATURE

```
void define_enforcers (
                in Topology::TopologicalTypeName type_name,
                in EnforcerRegistrationList enforcers)
        raises (TopologyMetaData::InvalidRegisteredObject,
                Topology::InvalidTypeName,
                Topology::NoSuchType,
                Topology::SchemaViolation,
                Topology::TransactionWillRollback,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The topological type whose enforcers are to be defined.

enforcers
The list of enforcers for the given topological type.

Output/Return
There are no output or return parameters for this operation.

SEMANTICS

This operation defines the enforcers for the topological type specified by the *type_name* parameter value, or if enforcers already exist for the topological type, this replaces them with the newly defined enforcers.

A type is considered ‘‘in use’’ if there are instances of that type, or if another type inherits from it. If the type is ‘‘in use’’ then rules cannot be added, removed or modified.

There is no return for this operation.

An enforcer is informed by the Topology Service whenever an entity of its associated type is affected in a way that satisfies one or more of its associated invocation conditions. The enforcer is given a veto over the change that produced the effect.

A Topology enforcer must be an object that supports the TopologyEnforcer interface. During the prepare to commit phase of a transaction, Topology will invoke the **validate**( ) operation of the enforcer, passing it the entity which is involved in the change. **Validate**( ) then checks whatever it deems necessary and returns a *Vote* as to whether the transaction should proceed or not.

There are two types of enforcers defined.

- A COMMON_ENFORCER is one in which a single object is registered as an enforcer. Thus, all Topology servers will be using this single object to validate modifications involving the specified topological type. This introduces a single point of failure if this object is unreachable.

- A NAMED_ENFORCER is one in which Topology is given the name of an object that it can locate in a naming service and will then narrow to the TopologyEnforcer interface. This provides for the use of a Unified Service (where multiple objects are providing the same functionality) or if there is only a single object it can, if necessary, easily be moved to a different host and reregistered with the naming service.

EXCEPTIONS RAISED

[TopologyMetaData::InvalidRegisteredObject]
Returned if an object specified as an enforcer cannot be narrowed to the TopoloygEnforcer interface.

[Topology::InvalidTypeName]
Returned if the type specified is not a valid topological type name..

[Topology::NoSuchType]
Returned if there is no type defined in the Topology store that matches the given type name.

[Topology::SchemaViolation]
Returned if there is a conflict between the Topology Schema definition and the persistent storage of the topology data, known as Topology data store.

[Topology::ServerNotContacted]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_Association_Rules — This operation is used to get a list of association rules for a topological type.

SIGNATURE

```
AssociationRuleList
get_association_rules (
                in Topology::TopologicalTypeName type_name)
        raises (Topology::InvalidTypeName,
                Topology::NoSuchType,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The name of the topological type whose rules are required.

Output/Return

Returns a list of the topology rules for the topological type.

SEMANTICS

The operation returns a list of association rules that represent all of the association rules defined for the topological type as specified in the *type_name* parameter value. The association rules that only apply to the topological type by inheritance are not returned.

EXCEPTIONS RAISED

[Toplogy::InvalidTypeName]
Returned if the given type name is not a valid topological type name.

[Topology::NoSuchType]
Returned if the given topological type name does not exist.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_Enforcers — This operation is used to get a list of enforcers for a topological type.

SIGNATURE

```
EnforcerRegistrationList
get_enforcers (
                in Topology::TopologicalTypeName type_name)
        raises (Topology::InvalidTypeName,
                Topology::NoSuchType,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

type_name
The name of the topological type.

Output/Return

Returns a list of the enforcers for the topological type specified.

SEMANTICS

This operation returns a list of enforcers that represents all of the enforcers defined for the topological type specified by the type_name parameter value. The enforcers that only apply to the topological type by inheritance are not returned.

EXCEPTIONS RAISED

[Toplogy::Toplogy::InvalidTypeName]
Returned if the given type name is not a valid topological type name.

[Topology::NoSuchType]
Returned if the given topological type name does not exist.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

5.2 TopologyMetaData.idl File

```
#if !defined (TOPOLOGYMETADATA_IDL)
#define TOPOLOGYMETADATA_IDL

#include <TopologyData.idl>
#include <ErrorService.idl>

module TopologyMetaData {
        interface TopologyEnforcer;

        enum Invoke Condition {
                ENTITY_ASSOCIATED,
                ENTITY_DISASSOCIATED,
                ENTITY_STATE_CHANGED,
                AGGREGATE_ENTITY_ASSOCIATED,
                AGGREGATE_ENTITY_DISASSOCIATED,
                AGGREGATE_ENTITY_STATE_CHANGED,
                ASSOCIATED_ENTITY_ASSOCIATED,
                ASSOCIATED_ENTITY_DISASSOCIATED,
                ASSOCIATED_ENTITY_STATE_CHANGED,
                ENTITY_MANAGED,
                ENTITY_UNMANAGED,
                ALL
        };

        typedef sequence<InvokeCondition> InvokeConditionList;

        enum EnforcerType {COMMON_ENFORCER, NAMED_ENFORCER};

        typedef string EnforcerName;
        struct EnforcerRegistration {
                InvokeConditionList invoke_conditions;
                union EnforcerObjectReference switch(EnforcerType) {
                        case NAMED_ENFORCER:
                                EnforcerName enforcer_name;
                        case COMMON_ENFORCER:
                        default:
                                TopologyEnforcer enforcer_ek;
                }enforcer;
        };

        typedef sequence <EnforcerRegistration> EnforcerRegistrationList;

        const unsigned long NO_LIMIT = ~0;

        struct AssociationRule {
                string role;
                long min_cardinality;
                long max_cardinality;
                Topology::TopologicalTypeName associate_type;
        };

        typedef sequence<AssociationRule> AssociationRuleList;
        exception TypeRedefinition{ };
        exception InvalidCardinality{ };
        exception TypeInUse{ };
        exception InvalidRole{ };
        exception NoSuchEnforcerImplementation{ };
```

```
interface TopologicalTypeManager: CosTransactions::TransactionalObject {

        // Life-cycle operations

        boolean create (
                        in Topology::TopologicalTypeName type_name,
                        in Topology::TopologicalTypeNameList parent_types)
                raises (Toplogy::InvalidTypeName,
                        Toplogy::NoSuchType,
                        ToplogyMetaData::TypeRedefinition,
                        Topology::TransactionWillRollback,
                        Topology::ServerNotContactable);

        boolean dismiss (
                        in Topology::TopologicalTypeName type_name)
                raises (Topology::InvalidTypeName,
                        Topology::TransactionWillRollback,
                        Topology::ServerNotContactable);

        // Query operations

        boolean exists (
                        in Topology::TopologicalTypeName type_name)
                raises (Topology::InvalidTypeName,
                        Topology::ServerNotContactable);

        Topology::TopologicalTypeNameList
        get_all_types( )
                raises (Topology::ServerNotContactable);

        Topology::TopologicalTypeNameList
        get_parents (
                        in Topology::TopologicalTypeName type_name)
                raises (Topology::InvalidTypeName,
                        Topology::NoSuchType,
                        Topology::ServerNotContactable);

        AssociationRuleList
        get_association_rules (
                        in Topology::TopologicalTypeName type_name)
                raises (Topology::InvalidTypeName,
                        Topology::NoSuchType,
                        Topology::ServerNotContactable);

        EnforcerRegistrationList
        get_enforcers (
                        in Topology::TopologicalTypeName type_name)
                raises (Topology::InvalidTypeName,
                        Topology::NoSuchType,
                        Topology::ServerNotContactable);

        // Set operations

        void define_rules (
                        in Topology::TopologicalTypeName type_name,
                        in AssociationRuleList rules)
                raises (TopologyMetaData::InvalidTypeModification,
                        Topology::SchemaViolation,
                        Topology::TransactionWillRollback,
```

```
                                        Topology::ServerNotContactable);

                    void define_enforcers (
                                        in Topology::TopologicalTypeName type_name,
                                        in EnforcerRegistrationList enforcers)
                                raises (TopologyMetaData::InvalidRegisteredObject,
                                        Topology::InvalidTypeName,
                                        Topology::NoSuchType,
                                        Topology::SchemaViolation,
                                        Topology::TransactionWillRollback,
                                        Topology::ServerNotContactable);
            };

            interface TopologyEnforcer : CosTransactions::TransactionalObject {

                    // Operations
                    CosTransactions::Vote
                    validate (
                            in Topology::TopologicalEntity affected_entity);
            };
};
#endif   // TOPOLOGYMETADATA_IDL
```

Chapter 6

Manipulating Queries in Topology

6.1 QueryExecution

This interface provides for interaction with the execution of a query.

The CORBA object *QueryExecution* models the evaluation of a query. These objects are created by a *QueryExecutionFactory* CORBA object (see Appendix A).

This interface supports operations to get the status of a query, stop the execution of the query, retrieve the query string, and retrieve the results from the execution of the query.

The lifecycle of a query consists of its creation via the QueryExecutionFactory at which point the query evaluation begins. The client may then inquire as to the current status of the query, may prematurely stop the query, or may begin requesting query results via the **get_next**( ) or **get_all**( ) operations. When the query execution has been completed and the last query result has been requested — either via **get_next**( ) or **get_all**( ) — then the QueryExecution object destroys itself.

All interfaces are expressed in CORBA IDL. They are grouped under TopologyQuery.idl.

NAME — DESCRIPTION

Stop — This operation is used to stop (abort) the execution of the query and destroys this *QueryExecution*.

SIGNATURE

```
void stop( )
        raises (Topology::ServerNotContactable);
```

SEMANTICS

This operation stops the evaluation of a query and destroys this *QueryExecution*.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_Status — This operation is used to get the status of a QueryExecution.

SIGNATURE

```
QueryExecutionStatus get_status( )
        raises (Topology::ServerNotContactable);
```

PARAMETERS

Input/Output

There are no input or output parameters for this operation.

Return

This operation returns the status of the evaluation of a query.

SEMANTICS

This operation returns the status of the evaluation of a query. The meaning of each of the fields in the *QueryExecutionStatus* is as follows:

truth_of_proposition

indicates whether the query is a true logical proposition on topology

partial

indicates whether the results for this query are all the result that may exist (for example, a possible result may be missing due to one of the topology servers being down in an installation)

finished

indicates whether the evaluation of the query is complete

num_result

indicates the number of individual results that have been obtained so far for this query.

A query represents a logical proposition on topology. This operation sets the *truth_of_proposition* field of the *QueryExecutionStatus* to be the result of that logical proposition. The *truth_of_proposition* flag is the only mechanism through which a client can obtain the result of a query executed with the *QueryResultOption* of QRO_TRUTH. A query executed with this option will never return any topological data as part of the results, and so it is not possible to determine the result of the logical proposition of a query by looking at the list of results. If *truth_of_proposition* is FALSE then any result data returned from **get_next**( ) or **get_all**( ) will be undefined.

If it is possible that there may be more results to this query than those that can be obtained, then the *partial* field of the *QueryExecutionStatus* will be set to TRUE.

When a query is evaluated, it is possible that some of the topology servers in the installation may not be contactable. This may force the evaluation to disregard possible results as one of these topology servers may be needed to obtain a complete result. If this happens then the results of the evaluation will be flagged as *partial*. The *partial* flag may also be true when the query was evaluated with a result option of QRO_TRUTH or QRO_ANY.

A query has the following status until a result has been found for the query (that is, before the notification's *interimResult* has been sent):

truth_of_proposition = FALSE

partial = TRUE

finished = FALSE

num_results = 0

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_Query — This operation is used to return the query string represented by this QueryExecution.

SIGNATURE

```
string get_query( )
        raises (Topology::ServerNotContactable);
```

PARAMETERS

Input/Output

There are no input or output parameters for this operation.

Return

The TQL query in its expanded form.

SEMANTICS

When a TQL string is given to a *QueryExecutionFactory*, the string is manipulated before being used to create a *QueryExecution*. This manipulation includes resolving references to other queries, instantiation of parameters, and optimization. This operation allows a client to obtain the TQL string that results from this manipulation process, and therefore to have access to the ‘‘real’’ query that will be executed by this *QueryExeuction*.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]

Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_Next — This operation is used to get the next individual result for this query and destroys the QueryExeuction if this is the last result.

SIGNATURE

```
void get_next (
                out QueryExecutionStatus status,
                out QueryResult result_data)
        raises (Topology::ServerNotContactable);
```

PARAMETERS

Output

status
The status of the executed query.

result_data
The individual result of the executed query.

SEMANTICS

This operation returns the next individual result that is available for the executed query. Once a result has been retrieved through this operation, the result is removed from the cache and is no longer available. The status is returned as described for *get_status*. If the *num_results* field of the *status* is zero, and the *finished* field of *status* is TRUE, then the contents of *result_data* are undefined. If the execution of the query is complete and this call to *get_next* returns the last result, then the *QueryExecution* is destroyed at the completion of this operation.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Get_All — This operation is used to get all of the remaining results for the executed query and then destroys the *QueryExecution*.

SIGNATURE

```
void get_all (
                in QueryResultFormat result_format,
                out QueryExecutionStatus status,
                out QueryResults result_data)
        raises (Topology::ServerNotContactable);
```

PARAMETERS

Input

result_format
Indicates the format in which the result should be returned. Alternatives are as a combination of all of the individual results for the query (COMBINED_RESULT) or as a list of individual results for the query (SEPARATE_RESULTS).

Output

status
The status of the executed query.

result_data
The results of the executed query.

SEMANTICS

This operation returns all of the remaining results for the query (that is, all of the results for the query less those already returned via the **get_next**( ) operation). The *status* is returned as described for **get_status**( ). If the *num_results* field of *status* is zero, and the *finished* field of *status* is TRUE, then the contents of *result_data* are undefined.

The QueryExecution is destroyed after this operation completes.

EXCEPTIONS RAISED

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

6.2 QueryExecutionFactory

This interface is provides the factory service for *QueryExecution* objects. This interface supports operations to create instances of *QueryExecution* objects, causing the evaluation of the query to begin.

NAME — DESCRIPTION

Create — This operation is used to create a *QueryExecution*.

SIGNATURE

```
QueryExecution create (
                in string tql_string,
                in ParameterBindings parameters,
                in QueryResultOption result_option)
        raises (CompileError,
                OVTopology::ServerNotContactable);
```

PARAMETERS

Input

tql_string
The query to be evaluated.

parameters
The actual values for arguments to the query.

result_option
The way that the results will be generated.

Return
A new QueryExecution object.

SEMANTICS

This operation creates a *QueryExecution* object and returns an object reference it.

If the I result_option is QRO_ALL, the the query is executed and the execution will attempt to find all valid answers to this query. If the *result_option* is QRO_ANY, then the query is executed and the execution will attempt to find no more than one answer to this query. If the *result_option* is QRO_TRUTH, then the query is executed and the execution will only attempt to answer the logical proposition formed by this query.

EXCEPTIONS RAISED

[CompileError]
Returned if the query is not syntactically valid. The details of the error which occurred during the compile of the query are included in the error message.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

NAME — DESCRIPTION

Create_With_Filter — This operation allows the client to filter on specific types of entities or specific tags, to reduce the amount of data returned from a query.

SIGNATURE

```
QueryExecution create_with_filter (
                in string tql_string,
                in ParameterBindings parameters,
                in QueryResultOption result_option,
                in QueryResultFilterList result_filter_list)
        raises (
                CompileError,
                Topology::ServerNotContactable);
```

PARAMETERS

Input

tql_string
The query to be evaluated.

parameters
The actual values for arguments to the query.

result_option
The way that the results will be generated.

result_filter_list
The filters to be applied to the returned result.

Return
A new QueryExecution object.

SEMANTICS

This operation behaves similarly to the **create**( ) operation with the additional semantics that as the query is executed and the result is built the specified filters are applied.

The *result_filter_list* parameter specifies a number of filters to be applied to the results returned to the client. The *filter_type* field of a particular filter specifies whether the filter applies to topological types (TYPE_FILTER) or to tags (TAG_FILTER). In the case of a topological type filter the *select_list* specifies 1 or more topological types, in the case of a tag filter the *select_list* specifies 1 or more tags or the special string "*" which is used to indicate any tag.

If the *result_filter_list* is an empty sequence then no result filtering occurs, otherwise each item in the sequence specifies a filter which restricts the entities and aggregate entities which appear in the query result.

In order to appear in the result an entity must satisfy all of the type filters in the list. An entity in the result satisfies a filter of type TYPE_FILTER if its topological type matches (taking inheritance of topological types into account) at least one of the topological types in the *select_list*.

An aggregate entity in the result satisfies a filter of type TAG_FILTER if the aggregate entity matched an AE pattern in the query which was tagged and either the *select_list* contains the string ‘‘*’’, or the tag for that aggregate entity matches at least one of the tags in the *select_list*.

EXCEPTIONS RAISED

[CompileError]
Returned if the query is not syntactically valid. The details of the error which occurred during the compile of the query are included in the error message.

[Topology::ServerNotContactable]
Returned if the topology server required to complete this operation is not contactable.

6.3 TopologyQuery.idl File

```
#if !defined (TOPOLOGYQUERY_IDL)
#define TOPOLOGYQUERY_IDL

#include <Topology.idl>
#include <TopologyData.idl>

module TopologyQuery {

        typedef sequence<string> StringList;

        typedef sequence<unsigned long> IndexList;

        enum QueryResultFilterType {TYPE_FILTER, TAG_FILTER};

        struct QueryResultFilter {
                QueryResultFilterType filter_type;
                StringList select_list;
        };

        typedef sequence <QueryResultFilter> QueryResultFilterList;
        struct QueryExecutionStatus {
                boolean truth_of_proposition;
                boolean partial;
                boolean finished;
                short num_results;
        };

        struct QueryResultAssociation {
                unsigned long pred_aggregate;
                unsigned long pred_entity;
                unsigned long pred_role;
                unsigned long succ_aggregate;
                unsigned long succ_entity;
                unsigned long succ_role;
        };

        typedef sequence <QueryResultAssociation> QueryResultAssociationList;
        struct QueryResultEntity {
                Topology::TopologicalEntity entity;
                unsigned long aggregate;
                unsigned long type;
        };

        typedef sequence <QueryResultEntity> QueryResultEntityList;

        struct QueryResultAggregate {
                IndexList entities;
                IndexList tags;
        };

        typedef sequence <QueryResultAggregate> QueryResultAggregateList;

        struct QueryResult {
                QueryResultEntityList entities;
                QueryResultAggregateList aggregates;
                QueryResultAssociationList associations;
                StringList types;
```

```
        StringList roles;
        StringList tags;
};

typedef sequence <QueryResult> QueryResults;

enum QueryResultFormat {COMBINED_RESULT, SEPARATE_RESULTS};

typedef string RegisteredQueryName;

typedef sequence <RegisteredQueryName> RegisteredQueryNameList;

enum QueryResultOption {
        QRO_ALL,
        QRO_ANY,
        QRO_TRUTH
};

struct ParameterBinding {
        string parameter_name;
        string parameter_value;
};

typedef sequence <ParameterBinding> ParameterBindings;

exception CompileError  {string reason;};

exception InvalidFilter    { };

interface QueryExecution {

        // Life-cycle operations

        void stop ( )
                raises (Topology::ServerNotContactable);

        // Status operations

        QueryExecutionStatus
        get_status ( )
                raises (Topology::ServerNotContactable);

        string get_query ( )
                raises (Topology::ServerNotContactable);

        // Result retrieval operations

        void get_next (
                        out QueryExecutionStatus status,
                        out QueryResult result_data)
                raises (Topology::ServerNotContactable);

        void get_all (
                        in QueryResultFormat result_format,
                        out QueryExecutionStatus status,
                        out QueryResults result_data)
                raises (Topology::ServerNotContactable);
};
```

```
        interface QueryExecutionFactory: CosTransactions::TransactionalObject {

                // Life-cycle operations

                QueryExecution create (
                                in string tql_string,
                                in ParameterBindings parameters,
                                in QueryResultOption result_option)
                        raises (CompileError,
                                OVTopology::ServerNotContactable);

                QueryExecution create_with_filter (
                                in string tql_string,
                                in ParameterBindings parameters,
                                in QueryResultOption result_option,
                                in QueryResultFilterList result_filter_list)
                        raises (CompileError,
                                Topology::ServerNotContactable);
        };
};
#endif //TOPOLOGYQUERY_IDL
```

Appendix A

Topology Query

A.1 Introduction

Queries are expressed textually using a Topology Query Language (TQL).

Topology supports generally topology queries on the set of all defined Aggregate Entities (AEs). Each query specifies one or more *meta_AE*s which are matched against the known AEs. A *meta-AE* is essentially a template for matching AEs.

Queries also specify *navigation paths* through Topology, which dictate which associations are to be followed between AEs in evaluating a query. Navigation paths are expressed in terms of *meta-associations*, which are templates for matching associations in Topology.

The result of a topology query consists of two parts. The first part is a boolean value that indicates whether the propositional expression posed by the query evaluates to TRUE or FALSE. This part of the result is always returned. The second part of the result is a sub-graph, or sub-graphs, that match the navigation paths specified by the query. The construction and existence of this part of the result depends on the type of result requested by the client.

The process is illustrated in Figure A-1 on page 86.

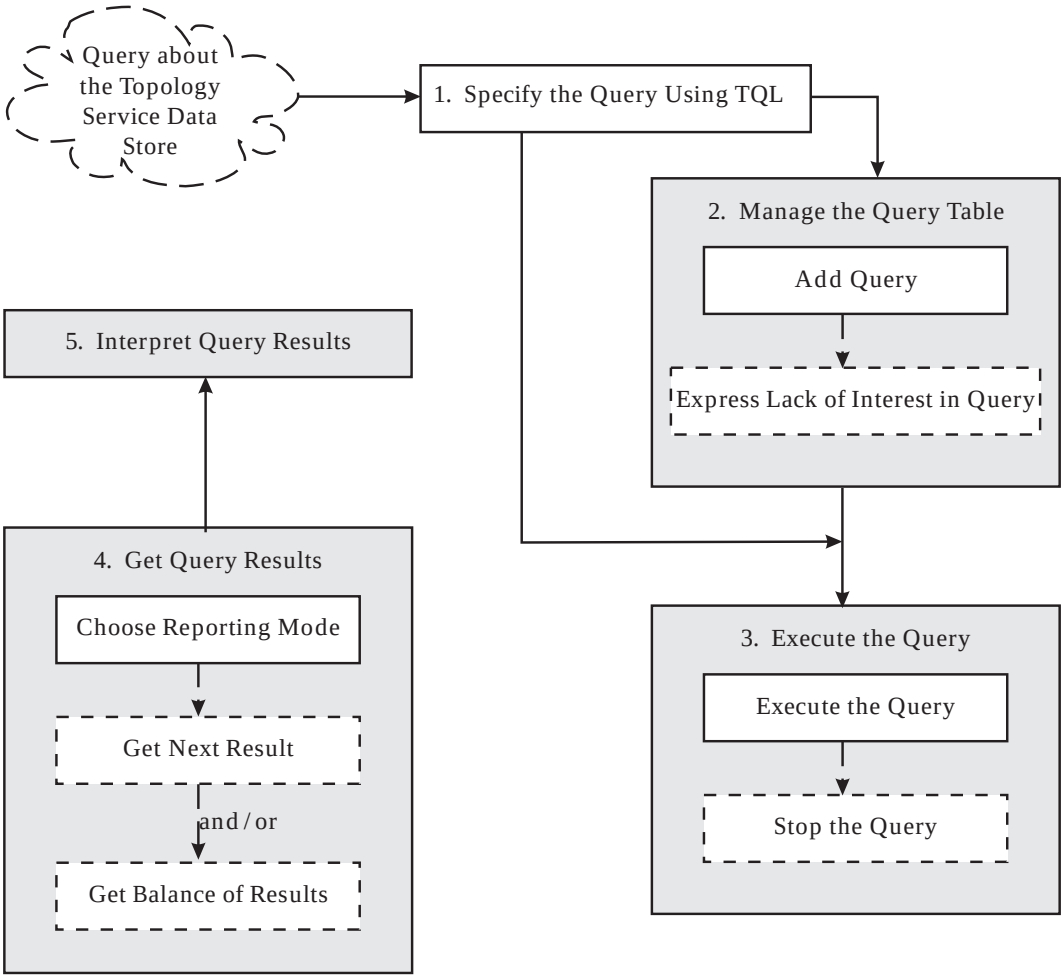


Figure A-1 A Process for Performing Navigation Queries

A.2 Topology Data Store

The Topology data store can be viewed as a graph of associated AEs. A topology graph is a set of nodes and a set of edges connecting the nodes. Nodes are AEs, and edges simply indicate the existence of an association between the two AEs.

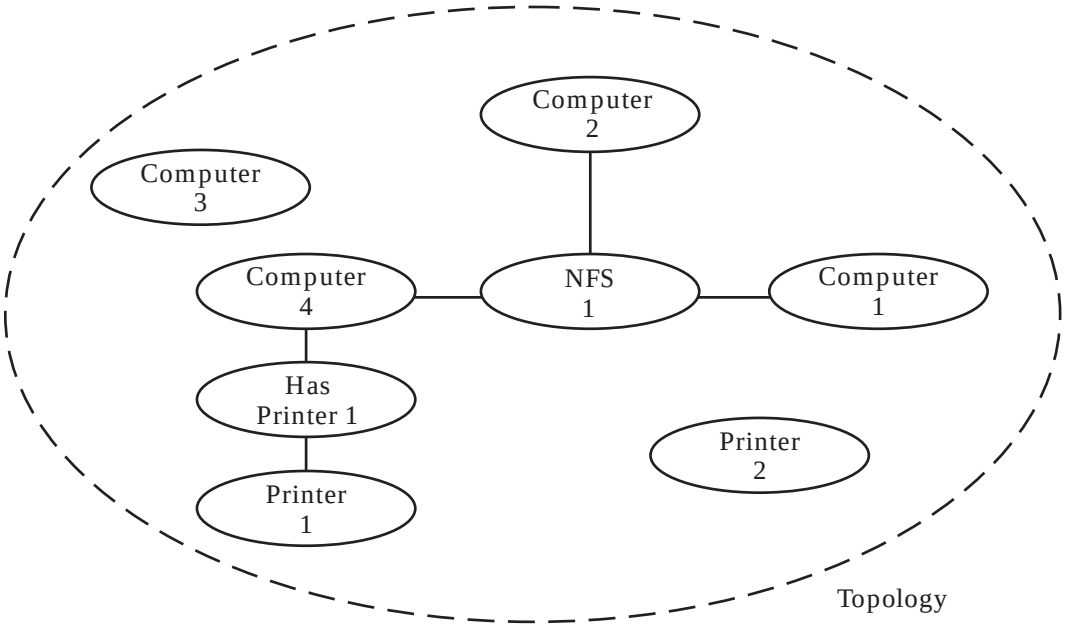


Figure A-2 Topology Graph Structure

A.2.1 Type Matching

Topological types are organized into inheritance type hierarchies.

An entity is matched against a topological type if it has the same topological type as that specified, or it has a topological type which is a specialization of that specified.

For example, in Figure A-3, if a meta-AE specified topological type RT-3, all AEs with topological type RT-3, RT-6, RT-7 and RT-9 will match the meta-AE.

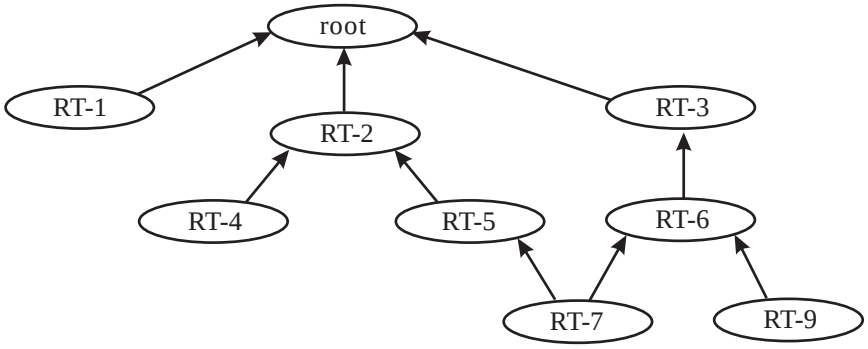


Figure A-3 Example of a Topological Type Inheritance Hierarchy

A.2.2 Meta-AE (Meta-Aggregate Entity)

A meta-AE is a template for matching AEs in topology. A meta-AE can be:

- a specific AE in topology
- *any* AE in topology
- a topological type
- a propositional expression of topological types. The propositional expression can be formed using the logical connectives AND, OR, and NOT. For example, a meta-AE may specify (Type-A AND Type-B) AND NOT Type-C.

A.2.3 Simple Navigation Paths

A simple navigation path contains one or more meta-AEs connected by meta-associations.

The presence of a meta-association in evaluating a query means that an association in the Topology data store must be traversed.

A meta-association can optionally specify a role for each of the meta-AEs that it connects. If specified, these roles limit the matching AEs to those that take the specified role in the corresponding association. The specification for a role may include an index, if cardinality is greater than 1.

A simple navigation path can be viewed as a proposition on topology that will return TRUE or FALSE, but not both. Each topology query must consist of at least one simple navigation path.

A simple navigation path is a *recipe* for matching paths through the Topology data store. A path in topology will match a simple navigation path if each AE in the topology path matches the corresponding meta-AE in the navigation path and adjacent AEs in the topology path are associated (in the correct roles) with each other; that is, the associations match the meta-associations. AE matching starts with the root meta-AE of the path and progresses sequentially

to the leaf meta-AE of the navigation path. Several paths through topology may match a simple navigation path.

Some examples of simple topology queries using a simple navigation path (see Section A.4 on page 98 for syntax specifications):

- Ex1:
  Is AE E1, associated via AE R1 with AE E2, where AE E2 is also associated via R2 with AE E3?
- Ex 2:
  Is there any AE matching ET1, associated via an AE that matches RT1, with an AE of that which matches ET2?

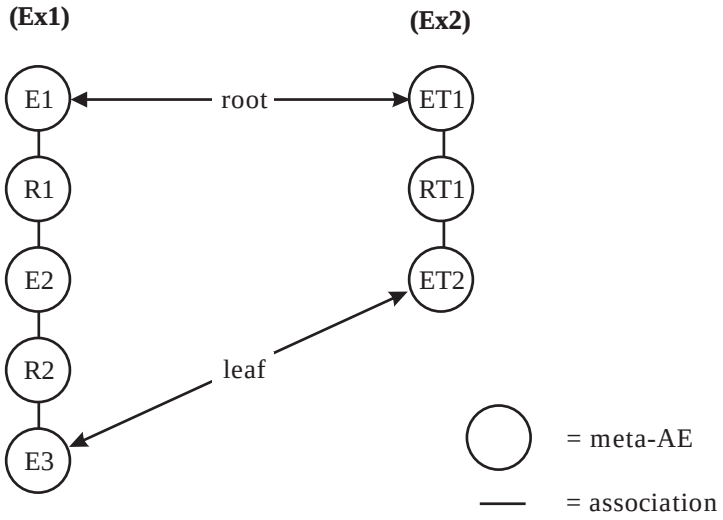


Figure A-4 Graphical Representation of the Queries

Both example queries start AE matching at the root meta-AE of their simple navigation paths. Ex 1 will only find, at most, one matching path in topology as it specifies a specific AE at every meta-AE. Ex2 may find several paths in topology that match the root meta-AE as it specifies topological types at every meta-AE. A query will return TRUE if it can find at least one matching path in topology.

A.2.4 Query Trees

Simple navigation paths can be combined, using logical connectives, to pose more advanced queries on the topology graph structure. The logical AND, OR, and NOT are used to produce compound propositional expressions on topology. They are expressed as query trees. A simple navigation path is the simplest form of query tree. Evaluation of the connectives is in accordance with the rules of logic. These are:

- Logical NOT (see next section)
- Logical AND: the result is TRUE if both query sub-trees being combined have a result of TRUE
- Logical OR: the result is TRUE if at least one query sub-tree being combined has a result of TRUE.

Nodes in query trees are either meta-AEs or logical connectives. Branches only occur at the logical AND and OR connectives which each have two branches. Complex navigation paths are paths within query trees.

If a null sub-query tree is a child of an AND or OR logical connective, that null query sub-tree is assumed to return the result TRUE. In other words, the tree is equivalent to one where the AND or OR node (with the null sub-query tree) is removed; that is, the node that was its child becomes the child of the parent of the AND or OR.

In query trees that are simple navigation paths, the edges always join nodes that are meta-AEs and always represent meta-associations. In contrast, in more general query trees, edges join nodes that may be meta-AEs or logical connectives. Edges in the tree are therefore not always meta-associations. Meta-associations exist between each pair of meta-AEs in the query tree where one meta-AE in the pair is the closest ancestor node of the other that is also a meta-AE.

A.2.5 Example Queries

Some example topology queries using the logical connectives are:

- Ex1:
  Is AE-E1 associated, via AE-R1, with AE-E2, where E2 is associated via AE-R2, with AE-E3 *AND* where E2 is also associated via AE-R3 with AE-E4?
- Ex2:
  Is AE-E1 associated, via AE-R1, with AE-E2, whree E2 is associated with AE-R2, with AE-E3 *OR* where E2 is associated, via AE-R3, with AE-E4?
- Ex3:
  Is AE-E1 associated, via AE-R1, with AE-E2, where E2 is *NOT* associated, via any AE with AE-E3?
- Ex4:
  Is AE-E1 associated, via AE-R1, with AE-E2, where E2 is associated via AE-R2, with AE-E3 *AND* where E2 is also associated, via AE-R3, with AE-E4 and where AE-E3 is associated via AE-R4 with AE-E5, *OR* where AE-E3 is associated via AE-R5 with AE-R6.

These examples show that the queries can be posed to Topology can be complex (and difficult to unambiguously describe in natural language).

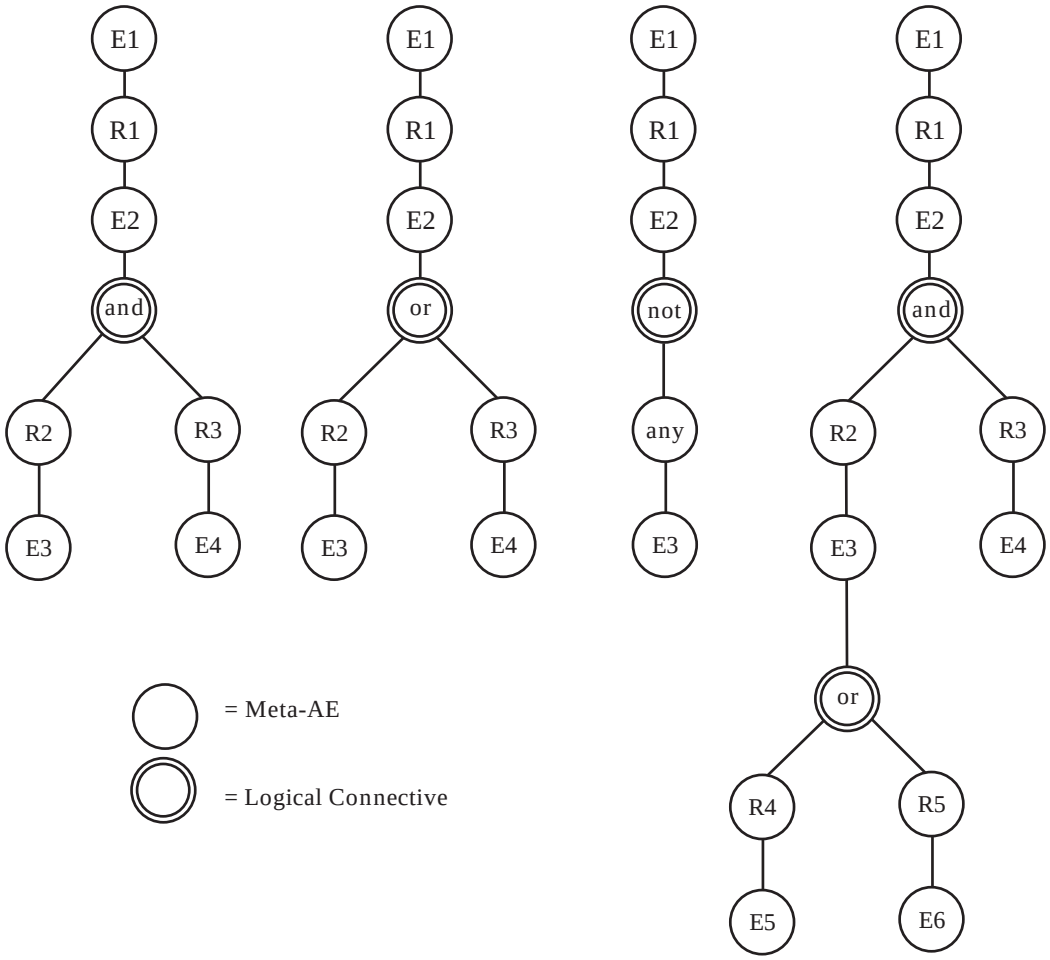


Figure A-5 Graphical Representation of Example Queries

All four queries start AE matching at the root meta-AE (E1). Queries Ex1 and Ex2 contains two complex navigation paths rooted at E1, whereas query Ex3 contains a single complex navigation path rooted at E1. Query Ex4 shows an example of nested logical connectives, with a total of three complex navigation paths that are rooted at E1.

- Ex1 will only return TRUE if there is at least one path through topology from E1 to E2 (via R1), and from E2 to both E3 (via R2) *AND* E4 (via R3).
- Ex2 will only return TRUE if there is at least one path through topology from E1 to E2 (via R1), and from E2 to either E3 (via R2) *OR* E4 (via R3).
- Ex3 will only return TRUE if there is at least one path through topology from E1 to E2 (via R1), and there is no path from E2 to E3 (via any single AE).
- Ex4 will only return TRUE if there is at least one path through topology from E1 to E2 (via R1), and from E2 to both E3 (via R2) *AND* E4 (via R3), and from E3 to either E5 (via R4) *OR* E6 (via R5).

The logical connectives AND and OR can also be used as root nodes in query trees. Examples of these types of queries are:

- Ex1:
  Is AE-E1 associated, via AE-R1 to AE-E2, AND is AE-E3 associated via AE-R2 to AE-E4?
- Ex2:
  Is AE-E1 associated, via AE-R1 to AE-E2, OR (is AE-E3 associated, via AE-R2, to AE-E4 AND is AE-E5 asssociated via any AE to AE-E6)? Paranthesis have been used to indicate evaluation precedence.

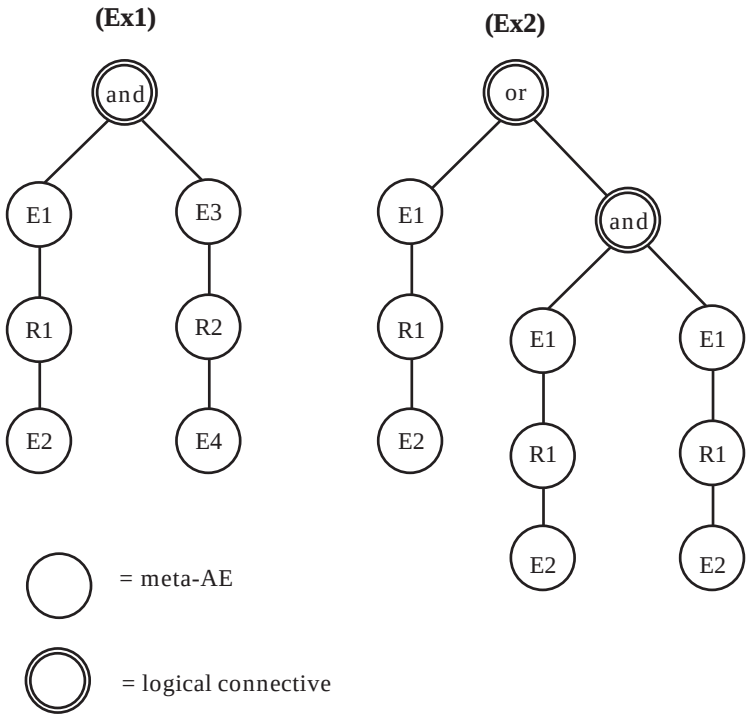


Figure A-6 Graphical Representation of Queries: Logical Connectives

A.2.6 Repetition

A simple navigation path, can be repeated a number of times as part of the query. A repetition statement is bounded by a lower limit. An optional upper limit may also be specified. These limits indicate the minimum and maximum number of times the repetition can occur in a topology path.

Here is an example of specifying repetition in a query:

- Is AE-E2 contained within AE-E1, where we understand containment to be transitive; that is, if E1 contains X and X contains Y, then E1 contains Y. Figure A-7 provides a graphical representation of the query. This example also shows how roles, specified for each meta-AE per association, are used. In the example, a topological type *containment* is used. This type supports two roles, *container* and *contained*. The other AEs also support these roles.

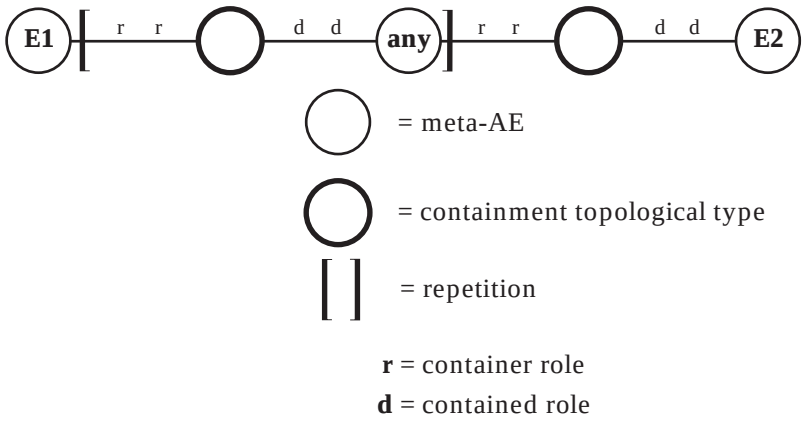


Figure A-7 Graphical Representation of Query Repetition

Queries containing repetition constructs where the upper bound is specified can always be expressed by an equivalent to a query without repetition. Figure A-8 shows the equivalent query for a query shown in Figure A-7’ where an upper bound of 2 is specified.

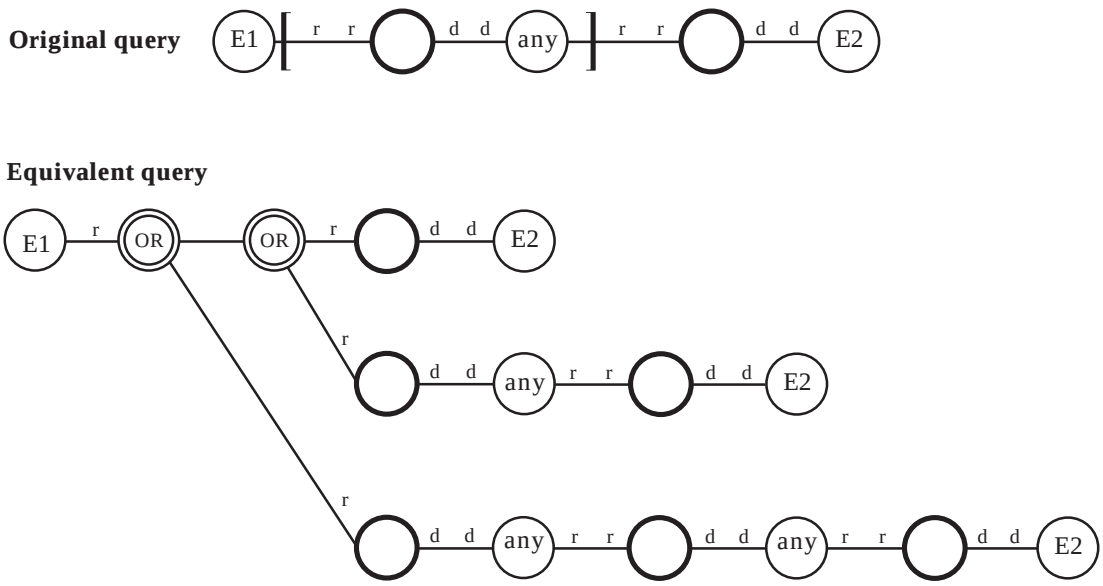


Figure A-8 Equivalent Query without Repetition

For the purposes of specifying the requirements for matching queries it is not necessary to consider the repetition construct given that equivalent queries can be constructed without them (for these purposes, queries containing repetition constructs where the upper bound is not specified can be considered as having a very large upper bound).

A.2.7 References to Registered Queries

At any point in a query where a meta-AE can be specified, it is also possible to specify a reference to a registered query. A registered query is a query that has been named. When a query that contains references to registered queries is evaluated, Topology replaces the references with the query tree that represents the registered query.

A.2.8 AE Matching Constraint

There are two things that are matched in a query. AEs match meta-AEs and associations match meta-associations.

The Topology Service supports the ability for the client to define constraints between pairs of meta-AEs that either require or forbit the same AE from matching both meta-AEs.

These constraints are specified for any pair of meta-AEs as:

- Equals: an AE that matches a specified meta-AE must also match the other meta-AE
- Not Equals: an AE that matches one specified meta-AE must not match the other meta-AE.

If any pair of meta-AEs does not have a constraint specified, then the same AE may or may not match both meta-AEs; that is, ‘‘don’t care’’ is the default.

There is similar support for expressing constraints for meta-associations.

A.2.9 Query Result

There are two parts to a query result:

- A boolean result which will return TRUE or FALSE, but not both. This part of the result is always returned and is an evaluation of the propositional expression posed on Topology.
- Sub-graphs of Topology that match the query. This part of the result is optional and is dependent on the combination of result construction rules and show indicator rules.

A.2.10 Result Construction Rules

The result construction rules specify the number of sub-graphs to be returned from a query. A query can only return sub-graphs if the boolean result of the query was true. There are three result construction rules:

- None:
  the query will not return any sub-graphs in the result; the boolean value is the only result returned
- Any:
  returns one sub-graph; the query will return the result of one successful match made on Topology
- All:
  returns one or many sub-graphs; the query will return the results of all successful matches made on Topology.

A query can only specify one result construction rule at a time.

A.2.11 User Tags

The Topology query service allows user *tags* to be specified on the meta-AEs of the query. While user tags are semantically irrelevant to the query evaluation process, the provide a means for the client to identify which topology AEs matched which meta-AE(s) in the query.

Each sub-graph returned in the result of a query will copy all user tags from the meta-AEs to their corresponding matched topology AEs. If the result construction rule of the query specifies the *graph* option, and the topology AE in the query result represents more than one meta-AE, the AE will copy all user tags from each meta-AE it matched.

The client can then extract AEs from the query result by specifying the tags of interest.

A.3 Topology Query System

The Topology Query System consists of the components as shown in Figure A-9.

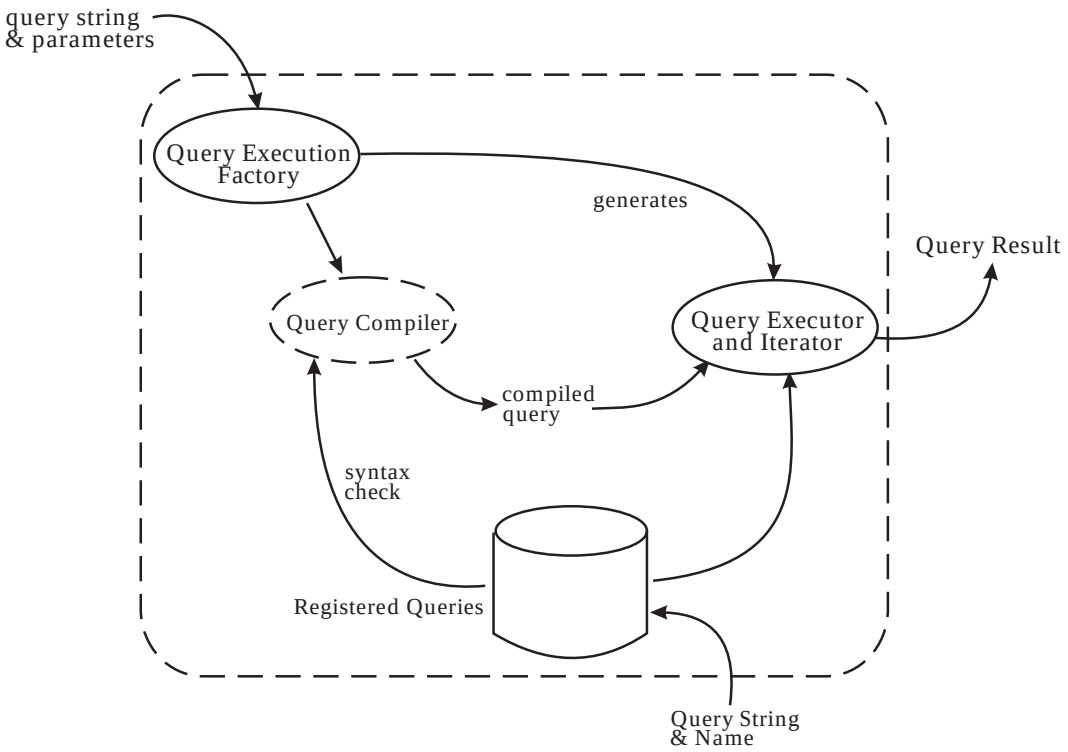


Figure A-9 Topology Query System

A.3.1 Query Language Definition

The query language definition consists of the language grammar (syntax) with corresponding semantics (see sections that follow on TQL: Topology Query Language). The language definition is a precise specification from which users can construct query programs for subsequent compilation and execution:

query program
: A query program is a textual expression of a query.

compiled query
: A compiled query is produced by the QueryExecutionFactory, which calls the TQL compiler, when a valid query program is processed.

parameterized query
: A parameterized query is a query where named parameters are used as placeholders that are substituted when the query is evaluated. There are three kinds of values that can be substituted for a parameter: a tag, a specific AE or a compiled sub-query.

query language compiler
: The TQL compiler, called by the QueryExeuctionFactory, takes as input a textual program (string) and performs syntactic and semantic checking on the input. If no errors are detected by this checking the query can be executed.

sub-query

A TQL query string may contain references to other queries which are defined as a separate string with a particular name. These references to other queries are called sub-queries. They could have been directly written into the query but because of the semantics defined for the relationships they find it makes the query more understandable to write them as a short reusable query.

For example, a query to find grandparents of an entity can most easily be written as a query to find the parents of the parents of the entity. Thus, a query to find parents is a very suitable candidate to write as a sub-query.

A potential implementation could allow these sub-queries to be registered and stored for retrieval and sharing between various applications.

A.4 TQL Overview

A.4.1 TQL Language

TQL is a textual language designed to facilitate the use of the Topology Service navigation query functions, by allowing navigation queries to be expressed textually.

The language maps directly onto the query trees that are supported by Topology. No additional functionality is supplied by the language and none is denied.

The scope of the language is restricted to providing a syntax for the expression of any allowable query tree and the conversion of the expression into the physical tree data structure. Minimal semantic checking is done by the compiler; this is deferred untiqa():-l query execution.

A.4.2 TQL Compiler

The TQL compiler methods are defined by the **QueryCompiler** interface. Every Topology server instantiates one object that supports the Query Compiler interface. There is, therefore, a TQL compiler on every host on which the Topology Server is running.

Clients access the compiler by initializing a Query object with a TQL string.

A.4.3 TQL Grammar

The YACC specification of the TQL grammar is as follows. A discussion of the TQL grammar also follows:

```
query: query_head '\{'  clause_expr '\}' ;

query_head: IDENTIFIER
    | IDENTIFIER '(' ')'
    | IDENTIFIER '(' argument_list ')' ;

clause_expr: sequence_clause
    | clause_member ;

sequence_clause: sequence_clause clause_member
    | clause_member clause_member ;

clause_member: clause
    | repetition_clause
    | meta_association
    | ABSENT_TOKEN clause_member
    | clause_member AND_TOKEN clause_member
    | clause_member OR_TOKEN clause_member
    | '('clause_expr')' ;

meta_association: labeled_meta_association
    | role labeled_meta_association
    | labeled_meta_association role
    | role labeled_meta_association role ;

labeled_meta_association: '-'
    | '-' label '-' ;

clause: clause_item
    | clause_item clause_qualifier ;

clause_item: topology_type_expression
```

```
    | query_reference
    | ARGUMENT
    | AENAME_TOKEN xfn ;    /* where xfn is an agregate entity name */

clause_qualifier: HEAD_TOKEN
    | TAIL_TOKEN
    | label
    | tag
    | clause_qualifier HEAD_TOKEN
    | clause_qualifier TAIL_TOKEN
    | clause_qualifier label
    | clause_qualifier tag ;

repetition_clause: '[' range clause_expr ']' ;

topology_type_expression: topology_type
    | topology_type_expression TOPOLOGY_TYPE_AND topology_type_expression
    | topology_type_expression TOPOLOGY_TYPE_OR topology_type_expression
    | TOPOLOGY_TYPE_NOT topology_type_expression
    | '(' topology_type_expression ')' ;

topology_type: xfn
    | ANY_TOKEN ;

tag: TAGGED_TOKEN STRING ;
  /*
   * tags are used for marking parts of the query so that when the
   * results of the query are returned you can tell what actual data
   * matched which parts of the query
   */

label: LABELED_TOKEN IDENTIFIER ;
  /* labels are used for setting constraints */

role: '<' IDENTIFIER '>'
    | '<' IDENTIFIER INTEGER '>' ;

range:  /* nothing */
    | INTEGER  RANGE_TOKEN
    | RANGE_TOKEN INTEGER
    | INTEGER RANGE_TOKEN INTEGER ;

query_reference: SUBQUERY_TOKEN xfn '(' ')'
    | SUBQUERY_TOKEN xfn '('query_actual_list')' ;

query_actual_list: clause_item
    | query_actual_list ',' clause_item  ;

argument_list: IDENTIFIER
    | argument_list ',' IDENTIFIER  ;

xfn: IDENTIFIER
    | STRING ;

ABSENT_TOKEN: 'ABSENT'
AENAME_TOKEN: 'AENAME'
AND_TOKEN: 'AND'
ANY_TOKEN: 'ANY'
HEAD_TOKEN: 'HEAD'
```

```
LABELED_TOKEN: 'LABELED'
OR_TOKEN: 'OR'
RANGE_TOKEN: '..'
SUBQUERY_TOKEN: 'SUBQUERY'
TAGGED_TOKEN: 'TAGGED'
TAIL_TOKEN: 'TAIL'
TOPOLOGY_TYPE_AND: '&'
TOPOLOGY_TYPE_NOT: '!'
TOPOLOGY_TYPE_OR: '|'
```

A.4.4 TQL Semantics

This section describes the grammar and semantics of TQL.

A.4.4.1 Query Structure

A TQL query has the following structure:

```
name(parameters) \{body\}
```

Name and Parameters

A TQL query has a name and optionally a list of parameters. This has the form:

```
name(parameter, parameter,parameter)
```

The name uniquely identifies the query. Parameters define placeholders for external queries to be used in the query body. A placeholder defines where in the query body the value for the parameter is used; for example:

```
q(p)  \{TypeA-p-TypeC\}
```

When the query is called, a value must be supplied for each parameter and this value is bound into the query where the corresponding placeholder is used.

Query Body

The query body specifies the declarative logic of the query. When reference is made to a ''query'', what it usually means is the query body.

The query body is an expression of clauses.

Clause Expressions

The body of the query is a *clause* expression. A clause expression consists of clause expressions (recursive definition) separated by the binary operators **AND**, **OR**, the unary operator **ABSENT** (equivalent to ''NOT''), and the binary **meta-association** operator (DASH, or ''-''). A clause expression can be a *clause* or a *repetition* expression.

Clause expressions are evaluated as per normal expression syntax, that is, a left associative infix expression of a binary tree with brackets for explicit grouping. Clause operators in order of precedence are: ABSENT , AND , OR , DASH.

For instance, the expression:

```
q() \{ clause OR clause AND clause or ABSENT clause \}
```

which is equivalent to the following, with brackets explicitly shown:

```
q() \{ ((clause OR (clause AND clause)) OR (ABSENT clause)) \}
```

An example of explicit precedence follows:

```
q() \{ (clause OR clause) AND (clause OR ABSENT clause) \}
```

A clause expression can be a list of clause expressions separated by *meta-associations* denoted by dashes.

```
q() \{ clause-clause-clause \}
```

We can then have the following expression:

```
q() \{ clause-clause-clause AND clause-clause \}
```

Brackets may be used for grouping clause epxressions so that they can be associated:

```
q() \{ (clause)-(clause)\}
q() \{ (clause AND clause)-(clause OR clause)-clause \}
```

Clauses

A clause can be a *meta-AE*, a *query reference* or a *placeholder*.

Meta-AEs

A meta-AE expression is a template for matching AEs in topology. There are three forms:

- topological type
- topological type expression template
- *any* topological type.

A topological type consists of a valid name for a topological type, for example:

```
query() \{ Computer\}
```

This defines a query that will match all AEs of type *Computer*.

A meta-AE may be an expression of topological type identifiers. A topological type identifier is the name of a valid topological type. The binary operators ‘‘&’’ (and) ‘‘|’’ (or) and the unary operator ‘‘!’’ (not) are used as type operators, with brackets for explicit grouping (as with clause expressions). The precedence of the operators from highest to lowest is ! & |. All of the type operators have a higher precedence than the clause expression operators, so they will bind more tightly to type identifiers, avoiding ambiguities.

For instance:

```
(Computer & Workstation) | MiniComputer & !Mainframe
```

This specifies a single AE matching this expression for typological types.

The following example shows that the type expression operators have a higher precedence. The expression:

```
Type1 AND Type2 & Type3
```

is evaluated as:

```
(Type1 AND (Type2 & Type3))
```

Note that meta-associations are not applied to topological type identifiers in the type expression but to the entire expression (meta-AE).

A wildcard match can be defined using the keyword *any*. This specifies that an AE of any type may match the meta-AE.

Constraints, *tags*, *roles* and *binding points* may be specified on meta-AEs.

A *constraint* is simply a label associated with a meta-AE. However, for a given query, the labels constrain the query result in that any two meta-AEs that share the same label must result in the same matching AEs in every query result produced. Correspondingly, non-matching labels must result in non-matching AEs. Constraints can also be assigned to meta-associations to produce matching or non-matching associations in the query results. Constraint labels are identifiers specified with the Keyword **LABELED**.

Tags are explained in Section A.2.11 on page 95.

Tags are indicated in the keyword **TAGGED**.

Roles are explained in Section 2.2.3 on page 14, and are specified at the start and end of the meta-AE expression to indicate the roles the AE plays in associations coming into it (the role specification at the start) or going out of it (the role specification at the end). A role specification is indicated by braces:

```
q() \{‘‘AE-pattern’’ <role> ‘‘AE-pattern’’\}
```

Binding points are used to process query references. There are binding points in each query referred to as *head* and *tail* binding points. Binding points are assigned to AEs to define the way that references to that query by other queries are resolved. When a referenced query is bound to its reference, a new bound query is produced by taking the referencing query and replacing the reference node with a referenced query. Replacement here means the head node of the referenced query will replace the reference node, and the subtree rooted at the reference node will be duplicated and attached to each tail of the referenced query.

A meta-AE can be labeled as both a head and a tail.

The following diagram depicts this:

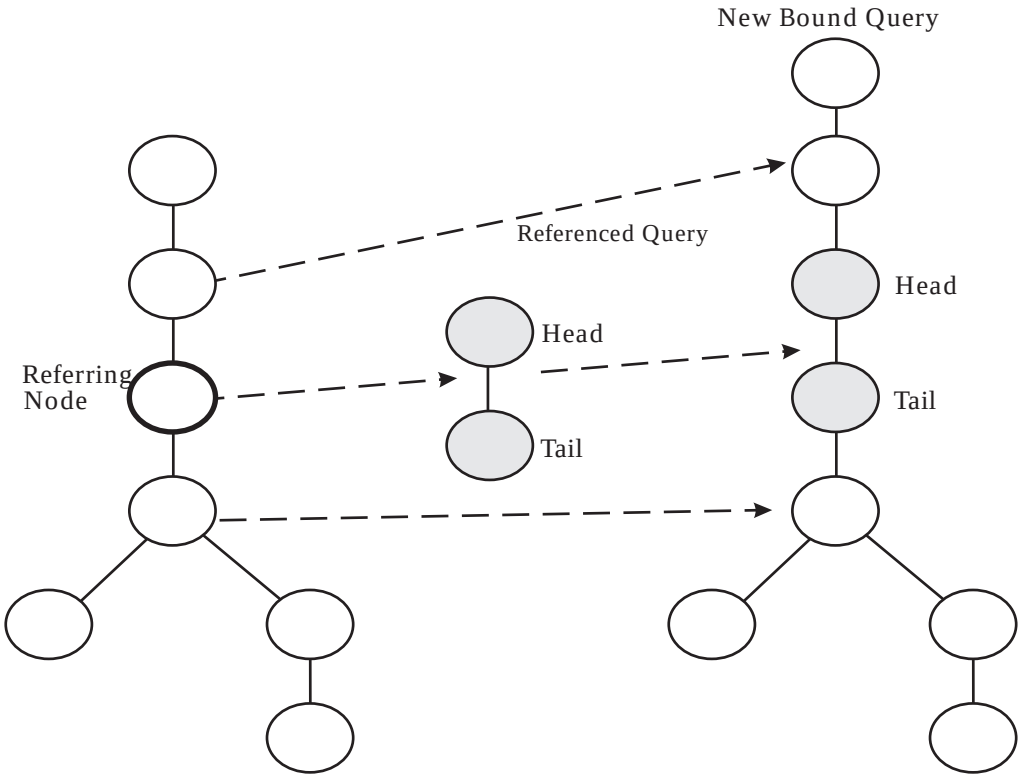


Figure A-10 Binding Points assigned to AEs

An example of a fully featured meta-AE expression is:

```
HEAD TAIL <client> (Computer&Workstation|PC) <server> TAGGED ‘‘box’’
```

This matches an AE that is a *Computer and a Workstation*, or a *PC*. Furthermore, it has two tags: *box* and *node*. It has a constraint label ‘‘x’’ on it and is both the head and tail binding point. It will be a client role with any associated meta-AE on the left, and a server role with an associated meta-AE on the right.

Meta-associations

A meta-association is specified using the binary ‘‘-’’ (dash) operator:

```
TypeA-TypeB-TypeC
TypeA-(TypeB AND TypeC)
(TypeA AND TypeB)-(TypeC OR TypeD)
TypeA | TypeB-TypeC & TypeD
```

A meta-association is placed between clause expressions:

```
clausexpr-clausexpr
```

The expression is evaluated from left to right. Every clause expression on the left is associated with every clause expression on the right.

In the simple case, with clause expressions that are clauses, this could look like:

```
q() \{ clause-clause-clause-clause-clause \}
```

for example, ‘‘Vehicles belonging to people’’:

```
q() \{ Person-Owns-Vehicle \}
```

Slightly more complex clause expressions could be associated:

```
q() \{ clause-(ABSENT clause)-clause-(clause) \}
```

A clause expression could be even more complex, forming a tree, such as:

```
clause AND clause OR clause
```

For example, ‘‘all coconuts or lemons, as long as there are no guavas’’:

```
q() \{ (Coconuts OR Lemons) AND ABSENT Guavas \}
```

Meta-associations can be placed between clause expressions that are grouped by brackets, giving a mechanism to associate subtrees.

```
q() \{ (clause1 AND clause2)-(clause3 OR clause4) \}
```

The example above would be equivalent to:

```
q() \{ (clause1-clause3 OR clause1clause4)
AND (clause2-clause3 OR clause2-clause4) \}
```

The subtree `(clause3 OR clause4)` is attached to both leaves of the tree `(clause1 AND clause2)`.

If the subtree (clause expression) on the left has multiple leaf nodes, the subtree associated to the right will be placed at each of these leaves with equals constraints between the corresponding elements in the right subtree.

For example, ‘‘computers with both a modem and a printer that are located in the same location’’:

```
q() \{ Computer-(Modem AND Printer)-(Located-Location) \}
```

Meta-associations may have constraint labels attached to them.

Meta-associations can be established between repetition expressions, denoted by square brackets, for example:

```
q () \{ TypeA-TypeB-[1..20,TypeC-TypeD]-[2,TypeE-]TypeF \}
```

Meta-associations can be established between placeholders. The binding points in the external query determine how it is bound into the query.

```
q(x) :- TypeA-x-TypeB \}
```

Meta-associations can be made between query references. The binding points in the referenced query determine how it is bound into the query.

```
q(x) :- parent(x)-Owns-Vehicle \}
```

Roles

A meta-AE can be in different roles for different meta-associations. Roles are specified inside ‘‘<’’ and ‘‘>’’ symbols and using the role type name.

The role is specified on the side of the meta-AE adjacent to the associated meta-AE, as a particular meta-AE can be in a different role with an associated meta-AE on the other side in the query:

```
MetaAE<roletype>
<roletype>MetaAE
<roletype>MetaAE<roletype>
```

For instance, a query, ‘‘Clients and Servers in the NetWare relationship’’:

```
q() \{ Computer<client>-NetWare-<server>Computer \}
```

Constraint Labels

Meta-AE and meta-association constraints are specified using labels. Labels are specified with **LABELED**.

A *meta-AE* constraint label is expressed after the meta-AE:

```
q() \{ TypeA LABELED x - TypeB - TypeC AND TypeD LABELED y;
```

A *meta-association* constraint label is expressed using two DASH operators with the label between them.

```
q() \{ TypeA-(LABELED x)-TypeB-(LABELED y)-TypeA;
```

There are three conditions for constraints:

- equals constraint
- not equals constraint
- don’t care (default).

Equals constraints are specified using the same label; *not equals* constraints with different labels; *don’t care* conditions are the default, with no label.

Sub-queries and Query References

A query may be defined in terms of other sub-queries. Sub-queries can be passed in as parameters and referenced in a clause with a placeholder or called using a query reference.

An example of a query passed in as a parameter is:

```
p() :- TypeA-TypeB;
q(p) :- p;
```

In the example, ‘‘p’’ is a placeholder for an external query and is equivalent to the fully expanded form:

```
q() \{ TypeA-TypeB\}
```

A query can be called via a query reference which has the form:

```
SUBQUERY queryIdentifier(parameter)
```

Parameters in a query reference can be either a placeholder for a parameter passed into the calling query or another query reference. The previous example is also equivalent to:

```
q() \{ p() \}
```

Consider the following query:

```
parent(x) \{ x<child>-Family-<parent>head,tail Person \}
grandparent(y)\{ parent(parent(x)) \}

q(x) \{ parent(x)-Located \{ loc \} Location
         AND grandparent(x)-Located- \{ loc \}Location \}
```

It queries, ‘‘Are X’s parents and grandparents located in the same location’’. It shows how parameters may be defined in terms of other query references.

Binding points are used to control what is bound for a sub-query that is called from a query. If no binding points are specified, the entire query is bound by default.

Binding Points

Binding points are used to control how a sub-query is bound into a calling query. The keywords **HEAD** and **TAIL** are used for this purpose. These are specified before the meta-AE expression. They are put before the label, if one exists, and between the role specifiers:

```
<role>HEAD TAIL (x) Typexpr TAGGED ‘‘tags’’<role>
```

One, all or none may be specified on a meta-AE.

The validity of binding points is not enforced by TQL but by the QueryExecutionFactory:

```
p()\{ HEAD TypeA-(TypeB-TypeC OR TypeD-TAIL TypeE)\}
```

The head and tail may be specified on a single meta-AE:

```
p() \{ TypeA-(TypeB-TypeC OR TypeD-(HEAD TAIL TypeE)) \}
```

Repetition

Repetition is used to specify paths that may be repeated over and over within bounds. The path that may be repeated is called a repetition unit.

Square brackets define the repetition unit:

```
q() \{ TypeA-TypeB[1..20, TypeC-TypeD-]TypeE \}
```

In a list of one or more clauses, such as:

```
clause-clause-clause-clause-clause
```

a repetition unit may be placed starting at a clause and ending at a meta-association:

```
clause-[clause-clause-]clause-clause
```

or starting at a meta-association and ending at a clause:

```
clause[-clause-clause]-clause-clause
```

The smallest repetition unit is one of the following:

```
[clause-]clausexpr
clausexpr[-clause]
```

An integer range can be placed on the repetition unit:

```
[1, TypeA-TypeB-]TypeC
```

This means repeat unit 1 to infinity times.

```
[1...20, TypeA-TypeB-]TypeC
```

This means repeat unit 1 to 20 times.

If no repetition range is specified, it defaults to ‘‘0 . . . ∞’’ (zero to infinity).

The repetition unit consists of an expression that is similar to a clause expression, except that it cannot have clause expression operators AND, OR, ABSENT and cannot have a repetition statement (repetitions cannot be nested).

Tags

Tags are strings that are attached to sets of entities that match the meta-AE template in the query. This allows individual result sets to be extracted from the query result later on.

Tags are specified as a list of strings immediately after the meta-AE type expression:

```
q() \{ TypeA TAGGED’’tag1’’ \}
q() \{ TypeA TAGGED ’’tag1’’,TAGGED ’’tag2’’,TAGGED ’’tag3’’ \}
```

Glossary

aggregate entity (AE)
A collection (aggregate) of topological entities which are recognized as forming a part of a larger logical or physical resource in a computer system environment.

CORBA
Common Object Request Broker Architecture.

COS
CORBA Object Service

IDL
Interface Definition Language.

interoperability
The ability of software and hardware on multiple machines and from multiple vendors to communicate effectively.

Topology
A set of valid relationships (associations) between resources in a defined computing environment.

Topological Entity
An individual logical or physical resource in a computer system environment.

Topology Service
A service which provides storage of topological data, maintains semantic integrity of relationships (associations), and provides query support by responding to clients interested in retrieving topological information.

OMG
Object Management Group.

XCMF
Common Management Facilities for a CORBA environment.