Utility Management System (UMS) Data Access Facility

Version 1.0
June 2001

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by several hundred members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG documentation is organized as follows:

OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency*, *Event*, *Externalization*, *Naming*, *Licensing*, *Life Cycle*, *Notification*, *Persistent Object*, *Property*, *Query*, *Relationship*, *Security*, *Time*, *Trader*, and *Transaction*.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility.*

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

The OMG collects information for each specification by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

Obtaining OMG Documents

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Alstom ESCA
- COMPAQ
- Langdale Consultants
- Oracle
- Power Technologies Inc.
- Psycor International
- Siemens
- SISCO Systems

Overview 1

Contents

This chapter contains the following topics.

1.1 Introduction

Utilities operate their water, gas, or power assets through control systems. The scope of control may include production facilities, bulk transmission networks, distribution networks, and supply points.

The most basic control system provides Supervisory Control and Data Acquisition (SCADA) functions. More sophisticated systems provide simulation and analysis applications that help the operators optimize performance, quality, and security of supply.

We will use the umbrella term Utility Management System (UMS) to refer to this class of system. It covers Water Quality and Energy Management Systems (WQEMS) in the water sector, and Energy Management Systems (EMS) or Distribution Management Systems (DMS) in the power sector.

Figure 1-1 illustrates a UMS. Two interfaces are shown as block arrows: SCADA interface (which will be covered in other RFPs) and the analysis data interface (the subject of this specification).

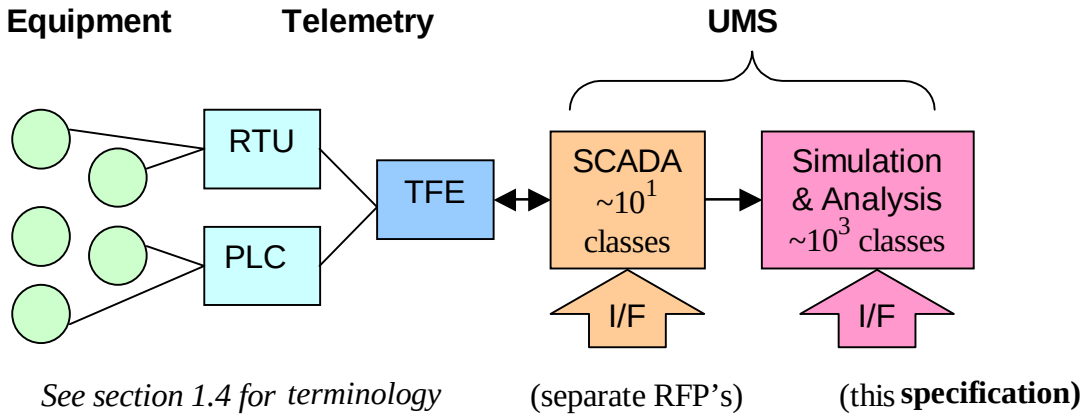

Figure 1-1 Utility Management System

The applications in a UMS employ extensive physical models representing networks, production facilities, and demand behavior among other things. These models distinguish a UMS from other types of control system.

For example, a power system model in an EMS may contain several hundred classes representing both physical and circuit theoretical concepts. Parts of this model must be understood by any external application that hopes to interpret the simulation and analysis results.

However, in all extant systems, this model is implemented in one or another proprietary database management system. There are no standard query languages or APIs for today's EMS, DMS, or WQEMS resident data.

This sets the UMS data access problem apart from conventional database access on the one hand, and from SCADA data access on the other where just a few classes are involved representing generic concepts such as measure and device.

Notwithstanding the difficulties, the need for inter-operation between the UMS and other applications or systems is evident. The UMS automates key utility activities that touch on many other parts of the business. Moreover, UMS functions are being redefined as the utility business environment is reshaped. Consequently, new inter-operation requirements are steadily emerging.

The goal of the Utility Management System Data Access Facility is to improve the interoperability of these UMS applications with other applications and systems.

1.2 Problems Being Addressed

This specification addresses the problem of obtaining the analysis data (both inputs and results) from a UMS on a read-only basis. This includes information describing a real or simulated state of the system together with the system's physical model data. The specification should be sufficient for integrating many applications and systems that operate "downstream" of a UMS in a near-real-time or non-real-time mode. It will apply to water, gas and electric power systems. This facility is also intended to work in concert with future facilities to handle more extensive integration cases.

Data Access

The specification provides interfaces to query the data that occur in a UMS including analysis inputs, analysis results, and physical model data.

Notification

Interfaces are provided for coarse-grained notification of changes in this data.

Concurrency Control

A mechanism is provided to allow a self-consistent group of data to be read from the UMS.

Data Semantics

For electric power systems, the specification establishes the semantics and structure of the data by reference to the Electric Power Research Institute Common Information Model (EPRI CIM).

1.3 Problems Not Being Addressed

The scope of the DAF was limited to exclude functionality that is likely to be system-specific, or which might increase the complexity of implementations. Accordingly, this specification does not address the following areas:

- Interfaces to update data in the UMS.
- Interfaces to define data schema, that is, classes, relationships, and attributes.
- Higher level functionality such as programmable data derivations, generalized query processing, or programmable event filtering. Nor does this specification address the following areas that are anticipated to be the subject of other RFPs:
  - Interfaces for generic SCADA functionality.
  - Interfaces for schedule and trading functionality.

1.4 Design Rationale

Simple Implementations

The DAF is distinguished from some other database APIs because it can be applied to very simple systems, which may not contain a full-blown database management system. We want people to create quick and simple implementations of the DAF to solve small-scale integration problems, without prejudice to more elaborate implementations.

Accordingly, the DAF defines very few interfaces, and does not require implementations to manage large, dynamic populations of CORBA objects. Most activity centers on the interface **ResourceQueryService**, which defines a small but sufficient set of queries as methods. The queries defined by the DAF are simple enough to be implemented in any UMS database and many related systems and applications.

High Performance Implementations

A UMS is a real-time system in the sense that it is used to make and execute operational decisions within strictly limited time boundaries. The performance requirements mean that a typical UMS does not use a typical database management system, which again leads to the need for the DAF.

To be effective in operational as well as off-line roles, the DAF must not introduce performance bottlenecks of its own. This has influenced the design in several ways, listed below. The first two emerged from performance testing and optimization in the prototype:

- ***Query Results Granularity*** - While simple, DAF queries can return a large amount of data at once in the form of a **ResourceDescriptionSequence**. On the server side, this allows implementations to optimize data retrieval without the need for read-ahead schemes. On the client, it minimizes network latency without the need for caching schemes.

  Data access patterns for UMS analysis software are well known, and are not friendly to caches. A typical analysis module reads a large amount of input data exactly once at the beginning of each analysis cycle. This input data is generally out-of-date by the next analysis cycle and therefore not amenable to caching.

- ***Data Value Representation*** - The basic unit of data, from which query results are composed, is a union type: **SimpleValue**. **SimpleValue** exploits our knowledge of the basic data types needed and eliminates CORBA *any* from the highest bandwidth part of the interface. This can make a significant impact on performance when accumulated across large amounts of data.

- ***Support for Pre-joined Views*** - Join optimizations are particularly applicable to relational databases, whether real-time or general-purpose. Here views are often defined to flatten the schema, effectively pre-joining tables along the lines of anticipated queries. The DAF includes a query, **get_descendent_values()**, which gives implementations the opportunity to optimize this type of query.

Partial Schema

The EPRI CIM is relatively large schema and a given DAF data provider (in the electric power domain) may not support all of it. In an effort to enable partial implementations of the EPRI CIM, the EPRI CCAPI task force is in the process of defining conformance blocks for specific application areas. A need for partial schema support is envisaged in the water and gas domains as well. The DAF provides a method, **get_resource_ids()**, which clients can use to determine whether a given class or attribute is supported by a given implementation. The same method will be used to determine whether a conformance block as a whole is supported, once those blocks are defined.

- ***Attribute-level Decomposition*** - A partial schema may omit whole classes of data, but just as often it will omit some attributes of a class and support others. This reflects the observation that different groups of attributes correspond to different functional areas. Members of the EPRI CCAPI and OMG Utility Domain Task Forces have frequently referred to these functionally related groups of attributes as *aspects*.

  In a traditional object oriented design, aspects would be classes in their own right. However, the EPRI CIM defines no such classes and it would be very difficult to keep such definitions up-to-date as new applications are discovered.

  Accordingly, the DAF allows implementations to support or omit data at the level of individual attributes on individual objects. This influences the design of the query results structures and, more fundamentally, the model of data that underlies the DAF.

- ***Federation of Data Providers*** - This specification anticipates that a number of data providers, each supporting part of an overall schema, could be combined to create a complete system. This would permit independent developers to extend a UMS with data providers as well as clients. However, this implies the need for system and configuration dependent logic that must direct queries to the appropriate data providers and merge the results obtained. The DAF provides for a proxy data provider to hide these details from both the clients and the ultimate data providers, ensuring that these components can be developed independently of any given system.

Schema Access

The DAF is not required to support access to metadata (i.e., schema information); however, it is not possible to formulate queries without reference to at least some schema elements: classes, attributes, and relationships. Furthermore, many generic applications not only reference schema information but also query it in detail. To take an example, an XML export facility might determine what to export and how by querying what classes exist, what attributes belong to each, and what the attribute types are.

- ***Identifying Schema Elements*** - This specification deals with schema at two levels. For simple clients and servers it is sufficient to identify the schema elements. The DAF employs Universal Resource Identifier references (URIs) for this purpose. A simple client is expected to know the URIs for the classes and attributes it wants to access. A mapping is provided between the EPRI CIM defined in UML and the URIs used in the DAF.

- ***Querying Schema Information*** - More sophisticated clients may want to query the schema information in greater detail. This is accommodated without requiring every client and data provider to deal with the complexities. Experience with developing data providers indicates that the schema query capabilities add significant cost and complexity and may even dominate the overall implementation cost. The design rationale employed here is that implementations only pay for what they use.

  When available, schema information is provided by reflection through the same query interface that provides population data. A standard meta-model has been adopted as the basis of schema queries. As with cases of partial schema support described above, the specification allows for a proxy data provider to match clients to the ultimate data providers. The proxy may add or merge schema information in those cases where it is required by the client but is not available from the ultimate data provider.

- ***Schema Versions*** - The DAF provides a way for clients and data providers to negotiate schema versions. All schema elements are designated by URI references that may include version identification. Mismatches between clients and data providers are exposed when the data provider does not recognize a given schema URI. Alternatively, a data provider can support multiple schema versions at once, an approach that is especially useful when the differences are slight.

- ***Schema and Meta-Model Extensions*** - The DAF provides a way for an existing or standard schema to be extended in a system-specific or proprietary way. In particular, because schema elements are named by URI references, additional elements can be introduced without name conflicts. Similarly, it is possible to extend the meta-model to provide richer schema information for those generic applications that need it.

Consistency with XML/CIM

The DAF shares both a model of data and a schema with the XML/CIM language. Thus the same interpretations of the CIM model are made in both standards. Moreover, this common basis should insure that these standards remain compatible in the future.

1.5 Module Dependencies

The following dependency diagram shows the modules that make up the DAF specification and their dependencies.

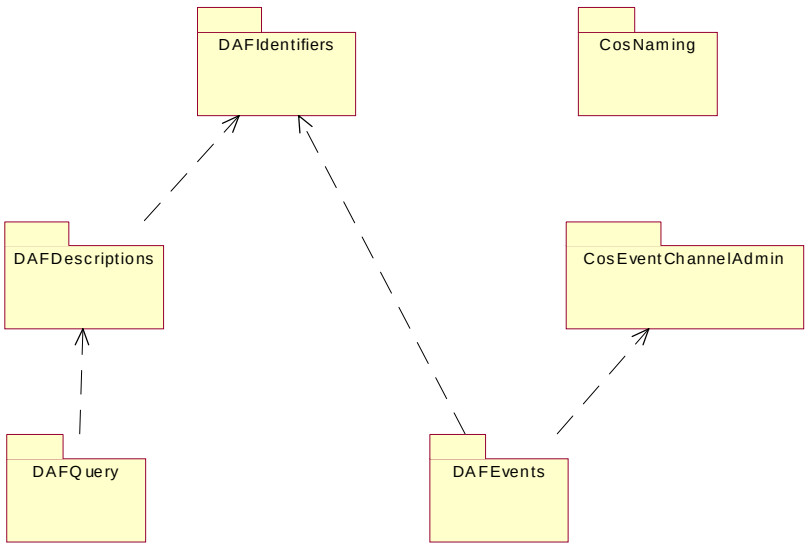

Figure 1-2 Module Dependencies

1.6 Conformance to the DAF

The DAF has two conformance points:

- The DAF interfaces.
- The EPRI CIM schema.

1.6.1 The DAF Interfaces

The DAF interfaces defined in Chapters 3 and 4, their semantics and the underlying data model are the first conformance point. The consolidated IDL is given in Appendix B. This is a mandatory conformance point

1.6.2 EPRI CIM Schema

The EPRI CIM schema for power system data, as defined in [8], is the second conformance point. This is an optional conformance point.

Resource Description Framework 2

Contents

This chapter contains the following topics.

2.1 Motivation

The Resource Description Framework (RDF), standardized by a W3C recommendation [1], provides the DAF with a formal model of data and a consistent terminology. The RDF model is similar in scope to the relational model of data but it is easier to adapt to various object oriented and hierarchical data repositories, while remaining compatible with the relational model. It also links the DAF to Web technologies and to the XML CIM language. Like the DAF, this language is based on a combination of RDF and the EPRI CIM.

Adopting an explicit model of data in the DAF is expected to help implementers adapt or wrap data repositories. Without it, they would need to infer an implicit structure behind the DAF interfaces. Implementers will find it useful to refer to the W3C recommendation, however this specification can be understood without it.

2.1.1 Fundamentals

The DAF formulates queries and their results in terms of *resources*, *properties*, and *values*. For example, the simplest query asks for the values of several properties of a given resource. The results are collectively known as a *resource description*. Other queries request information for all the resources belonging to a given *class*. These raw ingredients of the DAF are defined below.

2.1.1.1 Resources

A resource is anything with a distinct identity including, but not limited to, utility assets such as *switches*, *pumps*, and *generators*. In general, a Uniform Resource Identifier (URI) with an optional fragment identifier, as defined in [3], can identify any resource. However, a more compact proxy for the URI, the **ResourceID**, is used in DAF queries and query results. The Events chapter elaborates on how and when URIs and **ResourceID**s are used.

2.1.1.2 Properties

A property is some aspect of a resource that can be described. For example, *location* and *operation-count* could be properties of a switch. When properties appear in queries they are represented by the type **PropertyID**. However, a property is itself a resource and **PropertyID** is defined as a **ResourceID**.

A property has a *domain*, which is the set of resources to which it applies and a *range*, which is the set of values it can take. Associations between resources are created by properties whose range, like the domain, is a set of resources. For example, the *location* property might associate a *switch* with the *substation* in which it is located.

2.1.1.3 Property Values

A value in the Resource Description Framework is an elementary unit of data, which can be a *literal* such as a string or integer, or a reference to a resource.

A value represented in the DAF by the type **SimpleValue**, which is essentially a union of several fundamental CORBA types and the **ResourceID**.

2.1.1.4 Resource Descriptions

The purpose of the DAF is to supply information about resources, or resource descriptions. A resource description gives a value for each of several properties of a given resource. The simplest resource description describes a single property and is made up of a resource identifier, a property identifier, and a simple value. This elemental resource description is sometimes called a *statement*, or a *triple*. In the DAF a resource description is represented by the type **ResourceDescription**.

2.1.1.5 Classes

A class is a set of resources to which a given set of properties applies. For example, the set of all switches is a class that appears in a power system schema. Classes are referenced in queries by the type **ClassID**. As with properties, a class is itself a kind of resource and a **ClassID** is defined as a **ResourceID**.

2.2 Relationship To Other Models

The foregoing concepts are fundamental enough that they should find equivalents in any data repository. Some, perhaps approximate, equivalents are given in the following table as a guide.

Table 2-1 Relationship to Other Models

RDF	**DAF**	**Relational Model**	**UML**
Resource	Resource, ResourceID	Tuple (i.e., row)	Object
Property	Property, PropertyID	Attribute (i.e., column) or foreign key	Attribute or association
Class	Class, ClassID	Relation (i.e., table)	Class
Resource Description	ResourceDescription	Tuple value	-
URI	URI, ResourceID	Key value	-
Value	SimpleValue	Field value	-
Incomplete Information	Absence of a requested Property-Value in a Resource-Description	NULL value	-

2.2.1 Enumerations

It is often necessary to represent an enumeration, that is, a finite set of labeled values. Data models do not always provide a specific technique for this. For example, in the relational model an enumeration could be represented by a string or integer, with or without an associated reference table.

Similarly, the RDF model does not prescribe a specific way to represent an enumeration. However, the EPRI CIM RDF schema adopts the following convention:

- An enumeration type is represented by a Class.
- An enumerated value is represented by a resource whose type is that Class.

This convention does not support ordered enumerations. These are not used in the EPRI CIM schema.

Data Access Interfaces 3

Contents

This chapter contains the following topics.

This chapter defines the structures used to convey data, the resource descriptions, and the operations used to obtain data, the queries. Queries are formulated in terms of identifiers, which are dealt with at the end of the chapter.

3.1 Simple Values

The basic types, from which resource descriptions are constructed, are defined below.

```
module DAFDescriptions
{
// imported from identifiers module.
    typedef DAFIdentifiers::ResourceID ResourceID;
    typedef DAFIdentifiers::URI URI;

    // absolute time stamps in 100 nanosecond units
    // base time is 15 October 1582 00:00 UTC
    // as per Time Service specification
    typedef TimeBase::TimeT DateTime;
```

```
// a complex number
struct Complex
{
      double real;
      double imaginary;
};

// SimpleValue's can take on the following types.
typedef short SimpleValueType;
const SimpleValueType RESOURCE_TYPE = 1;
const SimpleValueType URI_TYPE = 2;
const SimpleValueType STRING_TYPE = 3;
const SimpleValueType BOOLEAN_TYPE = 4;
const SimpleValueType INT_TYPE = 5;
const SimpleValueType UNSIGNED_TYPE = 6;
const SimpleValueType DOUBLE_TYPE = 7;
const SimpleValueType COMPLEX_TYPE = 8;
const SimpleValueType DATE_TIME_TYPE = 9;
const SimpleValueType ULONG_LONG_TYPE = 10;

// a SimpleValue is the object of a resource description.
union SimpleValue switch( SimpleValueType )
{
      case RESOURCE_TYPE      : ResourceID resource_value;
      case URI_TYPE           : URI uri_value;
      case STRING_TYPE        : string string_value;
      case BOOLEAN_TYPE       : boolean boolean_value;
      case INT_TYPE           : long int_value;
      case UNSIGNED_TYPE      : unsigned long unsigned_value;
      case DOUBLE_TYPE        : double double_value;
      case COMPLEX_TYPE       : Complex complex_value;
      case DATE_TIME_TYPE     : DateTime date_time_value;
      case ULONG_LONG_TYPE    : unsigned long long ulong_long_value;
};
```

ResourceID and URI

These types are imported from the **DAFIdentifiers** module. Property values can refer to other resources through **ResourceID**s.

DateTime

Time is expressed by a 64 bit unsigned number (a **ULongLong**), which counts the number of 100 nanosecond units passed since 15 October 1582 00:00 UTC (the date of Gregorian reform to the Christian calendar). This convention is adopted from the CORBA Time Service, which in turn takes it from the *X/Open DCE Time Servic*e.

Complex

Complex numbers are used in a range of scientific applications and especially in power system analysis. In the DAF, complex numbers are expressed in Cartesian form.

SimpleValue, SimpleValueType

SimpleValues are the object of properties; their role in the resource description framework was outlined in the previous section. The DAF defines a **SimpleValue** as a discriminated union, where **SimpleValueType** is the discriminant. Fundamentally, a **SimpleValue** can be a resource identifier or a literal. The basic type of literal is a string to which the DAF adds explicit representation of the types defined above and (the largest variant of) each CORBA intrinsic type.

3.2 Resource Descriptions

All DAF queries return results in the form of resource descriptions.

```
module DAFDescriptions
{
    // ... see the previous section for first part of the module

    // properties are represented by their resource identifiers
    typedef ResourceID PropertyID;

    // predicate and object for a resource description
    struct PropertyValue
    {
        PropertyID property;
        SimpleValue value;
    };
    typedef sequence<PropertyValue> PropertyValueSequence;

    // resource description with one subject, multiple predicates
    struct ResourceDescription
    {
        ResourceID id;
        PropertyValueSequence values;
    };
    typedef sequence<ResourceDescription> ResourceDescriptionSequence;

    // iterator for handling large numbers of resource descriptions
    interface ResourceDescriptionIterator
    {
        unsigned long max_left();
        boolean next_n(
            in unsigned long n,
            out ResourceDescriptionSequence descriptions );
        void destroy();
    };
};
```

PropertyValue, PropertyValueSequence

A property value specifies a property by its resource identifier and its associated value for a given resource. The types **ResourceID** and **SimpleValue** are defined in the **DAFBasic** module.

ResourceDescription, ResourceDescriptionSequence

Each resource description identifies a single resource, by its resource identifier, and zero or more property values for that resource. The properties must be single valued, no provision is made for returning more than one value per property within a resource description. The class of the resource can be included in the resource description as the value of the *type* property (refer to the Schema chapter for details).

ResourceDescriptionIterator

Queries that return information about more than one resource return an iterator. The resource description iterator[1] allows a client to access a large query result sequentially, several resources at a time. This is necessary where the ORB limits message sizes. It also enables implementations to overlap the client and server processing of query results, if necessary.

The client and the data provider should cooperate to manage the lifetime of the iterator and the resources it consumes. The **destroy()** and **next_n()** methods allow the client and data provider respectively to indicate that the iterator may be destroyed.

In addition, the data provider may autonomously destroy the iterator at any time (for resource management or other reasons). If a client detects that an iterator has been destroyed, it will not interpret this condition in itself as either an indication that the end of the iteration has been reached, or as a permanent failure of the data provider.

next_n( )

This operation returns possibly 0 and at most n resource descriptions in the form of a resource description sequence. In all cases the state of the iteration is indicated by the Boolean return value:

- True means that there may be more resource descriptions beyond those returned so far.
- False means all the resource descriptions have now been returned. No further calls are expected for this iterator and the data provider may destroy the iterator at any time after the call returns.

destroy()

This operation is used to terminate iteration before all the resource descriptions have been returned. After **destroy()** is invoked, no further calls are expected for this iterator. The data provider may destroy the iterator at any time after the call returns.

---

1. The use of an iterator and the details of its design in this specification are an attempt to follow the pattern established in other CORBA standards. However, there are slight differences between specifications in the way that the lifetime of the iterator is managed and the behavior of the next_n method.

max_left( )

This operation returns an estimate of the number of resource descriptions remaining in the iteration. The result is intended to provide feedback in user interfaces or to select query strategies, but it cannot be used to detect the end of the iteration. Clients should allow that this operation might be expensive.

3.3 Resource Query Service

Resource descriptions are obtained from operations on the resource query service. The interface provides a family of three operations intended to be easy to use: **get_values()**, **get_extent_values()**, and **get_related_values()**. A fourth operation, **get_descendent_values()**, is a generalization of the other three and is capable of greater optimization.

The resource query service is defined as follows:

```
module DAFQuery
{
    // properties and classes are represented by resource identifiers
    // imported from the identifiers module.
    typedef DAFIdentifiers::ResourceID ResourceID;
    typedef DAFIdentifiers::ResourceID ClassID;
    typedef DAFIdentifiers::ResourceID PropertyID;
    typedef DAFIdentifiers::ResourceIDSequence PropertySequence;

    // results are resource descriptions from the descriptions module
    typedef DAFDescriptions::ResourceDescription ResourceDescription;
    typedef DAFDescriptions::ResourceDescriptionIterator
            ResourceDescriptionIterator;

    // queries that perform navigation use the association concept
    struct Association
    {
        PropertyID property;
        ClassID    type;
        boolean    inverse;
    };
    typedef sequence<Association> AssociationSequence;

    // exceptions generated by queries
    exception UnknownAssociation { string reason; };
    exception UnknownResource { string reason; };
    exception QueryError { string reason; };

    // the query service
    interface ResourceQueryService
    {
        ResourceDescription get_values(
            in ResourceID resource,
            in PropertySequence properties)
            raises( UnknownResource, QueryError );

        ResourceDescriptionIterator get_extent_values(
```

```
            in PropertySequence properties,
            in ClassID class_id )
            raises (UnknownResource, QueryError);

        ResourceDescriptionIterator get_related_values(
            in PropertySequence properties,
            in Association assoc,
            in ResourceID source )
            raises ( UnknownResource, UnknownAssociation, QueryError );

        ResourceDescriptionIterator get_descendent_values(
            in PropertySequence properties,
            in AssociationSequence path,
            in ResourceIDSequence sources,
            out AssociationSequence tail )
            raises ( UnknownResource, QueryError );
    };
};
```

ResourceQueryService

Each operation on this interface performs a single query. From a client's perspective there is always exactly one resource query service in a given context. (The section on proxies describes how multiple data providers are handled. The section on service location defines what is meant by a *context*.)

Each resource description returned by a query contains values for a subset of the properties requested. The property values appear in the same order as the properties that were passed to the query, although some may be omitted. A property value is omitted when it is not available from the data provider for the particular resource, or when the property identifier is unrecognized. This behavior makes it possible to federate multiple query services where each answers part of the query.

On the other hand, if the property is recognized but the data provider detects that it is not a member of the resource's class, the QueryError exception is raised. Similarly, QueryError is raised if the data provider determines that a property is many-valued (a resource description cannot represent multiple values for a property).

get_values( )

This query requests a resource description for a single resource given by its resource identifier. If the resource identifier is unknown to the data provider, the UnknownResource exception is raised.

get_extent_values( )

This query requests a description for each resource of a given class, that is, for each member of the class extent set. The class is given by its **ClassID**, which is a resource identifier.

- If the resource identifier is unknown to the data provider, the UnknownResource exception is raised.

- If it is recognized but does not represent a class, the QueryError exception is raised.

get_related_values()

This query requests a description for each resource associated with a given source resource. The source is specified by a **ResourceID**, and the association by an **Association** structure (defined below).

The data provider evaluates the association for the source resource, which yields zero or more result resources. For each result resource the data provider evaluates the given properties and generates a resource description, which is returned through the iterator.

- If the source resource identifier is unknown to the data provider, the UnknownResource exception is raised.
- If the data provider does not recognize the property specified in the association, the UnknownAssociation exception is raised.

Since data providers sometimes have only partial information, it is possible that the association is recognized but its value is not available for the given source resource. In that case, the UnknownAssociation exception is also raised. This distinguishes the case where no information is available from the case where the association value is empty.

If the data provider detects an error in the association or determines that it is not type-compatible with the source resource (as defined below), the QueryError exception is raised.

get_descendent_values()

This query is a generalization of the foregoing queries and is designed for clients that form queries in a generic manner. It also provides the most opportunity for optimization on the part of the data provider.

Comparison to other operations

The inputs to the query are the properties, path, and sources sequences.

- When the **path** has length 0 and the **sources** sequence has length 1, this operation is equivalent to the **get_values()** operation.
- When the **path** and **sources** are both of length 1, this operation is equivalent to the **get_related_values()** operation. However, where **get_related_values()** would raise UnknownAssociation this operation returns a non-empty tail sequence.
- When **path** and **sources** are of length 1 and the path specifies the inverse of the **rdf:type** property, this operation is equivalent to the **get_extent_values()** operation.

Normal Cases

The **get_descendent_values()** operation requests a description for each resource in a result set. The result set is formed from the given set of source resources, by following a chain of associations. The source set is specified by a **ResourceIDSequence**,

sources, and the chain of associations by an **AssociationSequence path**. If the length of the **path** argument is 0, the source set becomes the result set. Otherwise, the data provider evaluates the first association in the path for each source resource. This yields zero or more intermediate-result resources. For each intermediate-result the data provider evaluates the second association to yield another generation of intermediate-results. This process continues until the end of the association sequence is reached or an association is encountered that is not recognized or not available. In this way the query traverses a tree of resources.

If all of the associations are recognized and available, the result set is the set of resources generated from the last association in the path. The data provider returns descriptions for the result set. For each, the data provider evaluates the given properties and generates a resource description, which is returned through the iterator. The query also returns an empty association sequence through its tail argument.

Error Cases

If an unknown or unavailable association is reached before the end of the path the query returns partial results. The result set is the set of resources generated from the last association in the path that was recognized and available. If the n'th association is unrecognized or unavailable for a particular resource, the result set is obtained from the n-1'th association. If the first association fails, the result set is the source set. A resource description containing no property values is generated for each resource in the result set and is returned through the iterator.

An association sequence representing the failed part of the chain is returned through the tail argument. This tail sequence begins with the failed association and continues with the associations that follow it in the chain. The purpose of the tail is to enable proxy data providers (see Chapter 6) to complete the query by decomposing it and retrying the parts, through other data providers.

Exceptions are raised in limited circumstances. If any of the source resource identifiers is unknown to the data provider, the UnknownResource exception is raised. If the data provider detects an error in the association sequence (as defined below) or detects that the first association is not type-compatible with the source resource, the QueryError exception is raised.

Association

An association is a mapping from source set of resources to a destination set of resources. It contains a property and a class, each represented by their resource identifier. The interpretation of the property depends on the inverse member, a boolean. In any case, the property range must be a resource, not a literal type. It may be single-valued or many-valued. An association may be evaluated in the context of a source resource to yield a set of zero or more destination resources. There are two cases:

Inverse is false

If inverse is false, the value or values of the property for the source resource are obtained. If the association class is not a null resource identifier, then the results are further restricted to be members of that class.

The association is said to be type-compatible with the source resource if the type of the source resource is the same or a sub-class of the property's domain class.

Inverse is true

If inverse is true, the inverse of the property is evaluated. All resources are obtained for which the property value is the source resource. If the association class is not a null resource identifier, then the results are further restricted to be members of that class.

The association is said to be type-compatible with the source resource if the type of the source resource is the same or a sub-class of the property's range class.

The association class can be used to deal with a common pattern in schema which employ inheritance, such as the EPRI CIM, where the property range is a general class but the query addresses one of its more specific derived classes. If an EPRI CIM application wants all *GeneratingUnits* belonging to a given *Substation*, it must query the *Substation.MemberOf* property. But the range of this property includes all kinds of *PowerSystemResource*, so the association class can be used to limit the result to *GeneratingUnits*.

AssociationSequence

An association sequence defines a mapping from a source set of resources via zero or more intermediate sets, to a destination set of resources.

An association sequence is evaluated by repeated application of the procedure defined for a single association. Beginning with the source resource, the first association in the sequence is evaluated yielding a set of intermediate resources. For each of these the second association is navigated and the results concatenated forming a multi-set. Duplicates may or may not be eliminated from this intermediate result reducing it to a proper set. The **get_descendent_values()** query does not eliminate duplicates. The next association is then evaluated and the procedure is repeated for each subsequent association.

If any step in the sequence can yield resources that are not type-compatible with the next association in the sequence, then the association sequence is in error.

3.4 Resource Identifiers

Resource identifiers are compact tokens that designate resources in queries, query results, and events. The **ResourceIDService** provides methods to translate **ResourceID**s to and from a standard textual form, the Uniform Resource Identifier (URI) [3].

3.4.1 Purpose of Resource Identifiers

Queries use resource identifiers in two main ways:

- To designate properties and classes in queries. Class and property resource identifiers would be typically obtained from the resource identification service.

- To cascade queries, so that a resource identifier returned by one query is input to another.

3.4.2 Purpose of URIs

Unlike Resource Identifiers, URIs provide a *universal* naming scheme and thus a vehicle for interoperation between independent software and systems.

The main uses for URIs in the DAF relate to schema elements:

- ***Bootstrapping*** - No query can be issued without at least some Resource Identifiers (to designate the class and properties to be queried). Initial Resource Identifiers are obtained from URIs via the Resource Identifier Service.
- ***Standard schema*** - URIs are used as agreed names for the classes and properties of the EPRI Common Information Model (CIM) and the Resource Description Framework Schema (RDFS).
- ***Custom schema*** - URIs are used to designate system-specific classes and properties. URIs avoid name conflicts, especially among properties of the same class.
- ***Version control*** - URIs may contain version information, which enables clients and data providers to distinguish different versions of a class or property.

The role of URIs in the DAF is further elaborated in the Schema chapter.

3.4.3 Resource Identifier Service

Resource identifiers and the resource identification service are defined by the following IDL.

```
module DAFIdentifiers
{
    // the uniform resource identifier from IETF RFC 2396
    typedef string URI;
    typedef sequence< URI > URISequence;

    // the resource identifier
    struct ResourceID
    {
        unsigned long long container;
        unsigned long long fragment;
    };
    typedef sequence < ResourceID > ResourceIDSequence;

    // service for translating and managing resource identifiers
    exception LookupError {      string reason; };

    interface ResourceIDService
    {
        ResourceIDSequence get_resource_ids(
            in URISequence uris )
            raises( LookupError );
```

```
        URISequence get_uris(
            in ResourceIDSequence ids )
            raises( LookupError );
    };
};
```

URI, URISequence

A URI is a Uniform Resource Identifier with an optional fragment identifier as defined in [3]. This combination is more properly called a *URI-reference*. In order to clarify the following discussion, the syntax of a URI-reference is:

<scheme-name> ':' <opaque-part> '#' <fragment-identifier>

The **opaque-part** represents the main body of the URI and (depending on the scheme) it may include a domain name and a pathname.

In principle, anything that can be named can be given a unique URI. Moreover, any existing naming scheme can be subsumed into this universal system by, for example, assigning it a scheme-name. In the DAF it is assumed that all objects in a UMS, both concrete and abstract, population and schema, have unique URIs.

A DAF implementation is not required to use the protocol implied by the scheme-name of the URIs it employs. For example, if the URI for the class of Transformers was http://iec.ch/TC57/2000/CIM-schema-cimu09a#Transformer, it would not imply that a DAF implementation needs to use the http protocol, nor contact a host in the IEC domain (iec.ch). In this example, the URI is merely the agreed designation of a commonly understood concept.

However, the DAF does require URIs to be in absolute, canonical form so that they may be consistently compared and translated into DAF resource identifiers. In addition, the DAF treats the fragment separately from the rest of the URI-reference, as explained below.

ResourceID, ResourceID

A **ResourceID** is a compact, fixed length substitute for a URI. It is introduced to enable high performance implementations of the DAF where repeated URI parsing, comparison, and lookups would be too expensive. The **ResourceID** may also simplify some DAF implementations.

A **ResourceID** is divided into two 64-bit fields to further improve the performance of implementations when dealing with large groups of related resources and multiple data providers. The **container** and **fragment** fields correspond to equivalent fields in a URI-reference and support a similar division of responsibility for allocation and resolution of the identifier (see below).

ResourceIDService

This interface enables translation of resource identifiers to and from URI-references. From a client's perspective there is always exactly one resource identifier service in a given context. (The section on proxies describes how multiple data providers are handled. The section on service location defines what is meant by a *context*.)

Each data provider implements this interface and most must be prepared to use this interface as a client of another data provider. The latter is required of data providers that use shared resources, such as the EPRI CIM schema.

get_resource_ids()

This operation accepts a sequence of URI-references and returns the corresponding resource identifiers. The returned sequence is the same length as the input sequence and in the same order. If the data provider does not recognize a particular URI or is not responsible for its translation, a null resource identifier is returned for that position.

If the data provider needs a resource identifier for a resource it does not control, it invokes this operation as a client of the responsible data provider or the proxy data provider. For example, a data provider might manage a population of UMS data but not the schema for that population. It would then need to obtain schema resource identifiers from another data provider.

get_uris()

This operation accepts a sequence of resource identifiers and returns the corresponding URI-references. The returned sequence is the same length as the input sequence and in the same order. If the data provider does not recognize a particular resource identifier or is not responsible for its translation, an empty string is returned for that position.

As with the **get_resource_ids()** method, a data provider may need to invoke **get_uris()** acting as a client of another data provider.

3.4.4 Containers and Fragments

ResourceIDs are grouped by **container**. The following holds true for two **ResourceID**s with the same **container** value and different **fragment** values in a system with multiple data providers:

- The same data provider allocates the two **ResourceID**s and is able to translate them to and from URIs via its **ResourceIDService**.
- When translated to URIs the two **ResourceID**s yield the same **scheme-name** and **opaque-part**, respectively.

In addition:

- An implementation should attempt to ensure that the Property values of two resources with the same **container** value are *more likely* to be available from the same data provider than otherwise.

In effect, the **container** field corresponds to the main part of the URI-reference and the **fragment** field corresponds to the **fragment-identifier**.

3.4.5 Resource Identifier Allocation

In a DAF implementation, resources and **ResourceID**s correspond on a one-to-one basis. In other words, a resource has one and only one **ResourceID**.

To ensure uniqueness, a DAF implementation must coordinate allocation of container values among all participating data providers. The allocation of **ResourceID container** values takes place at system initialization or configuration time and is beyond the scope of this specification.

However, each data provider is solely responsible for **fragment** allocation within some given set of containers. Thus, in a URI-reference the **fragment-identifier** is independently allocated and interpreted in the context of the containing resource. Similarly, in a **ResourceID** the **fragment** field is allocated in the context of the **container**.

3.4.6 URI to Resource Identifier Translation

URI-references and resources correspond on a many-to-one basis. A URI-reference designates a single distinct resource, but a given resource may have more than one URI-reference.

In a DAF implementation one URI-reference for each resource is distinguished. This is the URI returned by **ResourceIDService** when the **ResourceID** is translated.

Conversely, the **ResourceIDService** must be able to translate the distinguished URI-reference of a resource to its **ResourceID**. It is implementation dependent whether the **ResourceIDService** can translate any other URIs for the same resource to its **ResourceID**.

3.4.7 Standard and Implementation-Specific URIs

Each URI-reference used in a DAF implementation is either *standard* or *implementation-specific*.

A standard URI-reference is fixed at design time and represents a point of coordination between otherwise independent client and/or data provider implementations. Generally, a standard URI-reference will include the domain name of the organization that defined it and information to identify the version of the standard. This specification standardizes the URI-references of CIM classes and properties.

Implementation-specific URI-references may be used in all other cases. This type of URI can be generated by the data-provider in one of several ways:

- Based the resource's (numeric) **ResourceID**.
- From attributes of the resource.
- By mapping a proprietary naming scheme to a URI syntax.

3.4.8 Resource Identifiers for Standard URIs

Standard URIs are, by definition, known to more than one client and/or data provider. If there are multiple data providers that support a given URI, then only one of them can be responsible for translating it to a **ResourceID** (along with the other URIs with the same container).

The designated data provider must supply the translation to the other data providers, which must function as clients of its **ResourceIDService**.

A conforming DAF data provider implementation must be capable of obtaining the translations of standard URIs from another data provider if so configured.

3.4.9 Resource Identifier Persistence

The mapping between resource identifiers and resources is required to be stable for the lifetime of a system, that is, for as long as there are active clients or data providers operating in the same context. In practice this may mean that a data provider must make its resource identifiers persistent in case it is restarted during the lifetime of the system.

However, this is the minimum standard of persistence. The resource identifier design is intended to support implementations that want to permanently assign identifiers to resources.

3.4.10 Null Resource Identifiers and Null Fragments

The null resource identifier is a reserved value, which refers to no resource. The **container** and **fragment** of the null resource identifier are both zero.

A URI-reference with no fragment identifier also has a special representation. This type of URI corresponds to a non-null resource identifier with a **fragment** value of zero.

3.5 Query Sequence

The following diagram illustrates the sequence of operations involved in a query, and the interaction of the **ResourceIDService** and **ResourceQueryService**.

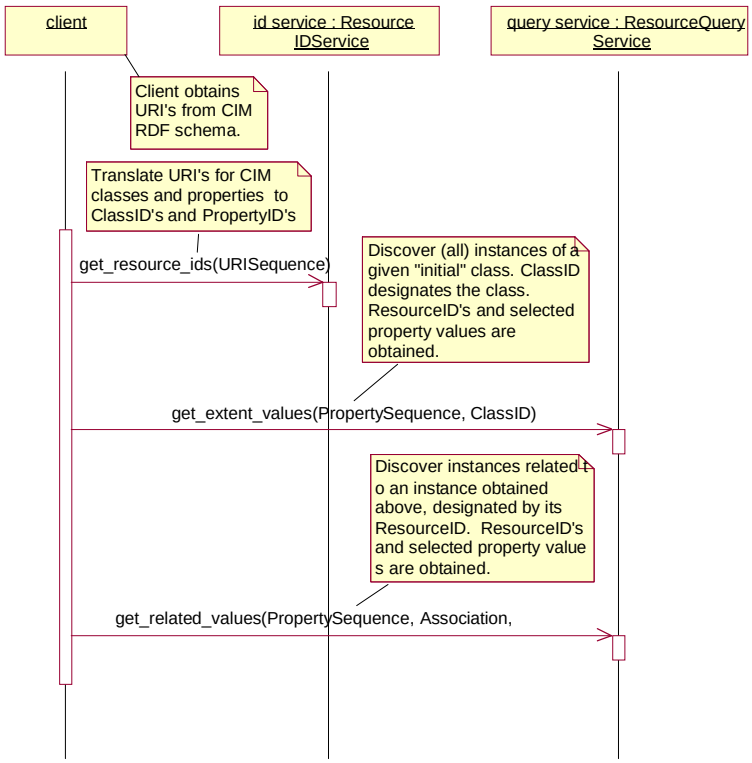

Figure 3-1 Query Sequence Diagram

Events 4

Contents

This chapter contains the following topics.

4.1 Introduction

The **DAFEvents** module provides the means to notify clients of data changes and to ensure consistent data access.

```
module DAFEvents
{
    // event emitted by data provider after data changes
    struct ResourceChangeEvent
    {
        DAFIdentifiers::ResourceIDSequence affected;
    };

    // interface for connection an event push consumer
    interface ResourceEventSource
    {
        CosEventChannelAdmin::ProxyPushSupplier obtain_push_supplier();
        unsigned long long current_version();
    };
```

};

ResourceChangeEvent

This structure is passed as an event to indicate a data change. Events of this type are delivered after a data change and indicate that the client may begin reading or rereading data.

affected

The **affected** member is a sequence of resource identifiers that indicates what data has changed. If **affected** is empty, then the client must assume that any or all of data supplied by the data provider may have changed.

Otherwise, affected contains one or more class identifiers. The client should assume that instances of each class have been created or destroyed, or their property values have changed.

ResourceEventSource

This interface allows a client to detect updates and concurrency conflicts. The interface is implemented as a singleton object.

obtain_push_supplier()

This operation is similar to the **obtain_push_supplier()** defined in the CORBA Events interface **CosEventChannelAdmin::ConsumerAdmin**. It is used by a client to connect its **CosEventComm::PushConsumer** to the data provider, through which resource change events will be delivered.

Connection is a two step process. First, **obtain_push_supplier()** is invoked to return a **ProxyPushSupplier**. Then **connect_push_consumer()** is invoked on that object, passing the client's **PushConsumer**.

Events of type **ResourceChangeEvent** will then be passed via the **PushConsumer**'s **push()** operation.

Disconnection is achieved by invoking **disconnect_push_supplier()** on the **ProxyPushSupplier** object.

current_version()

This operation is used by clients to detect concurrency conflicts. A client should query the current version before and after issuing a set of queries. If the values are equal and non-zero, then the query results are taken to be self-consistent. Section 4.3, "Current Version and Transactions," on page4-3 elaborates on this protocol and its motivation.

4.2 Event Sequence

The following sequence diagram illustrates the lifecycle of an event connection between a client and a data provider.

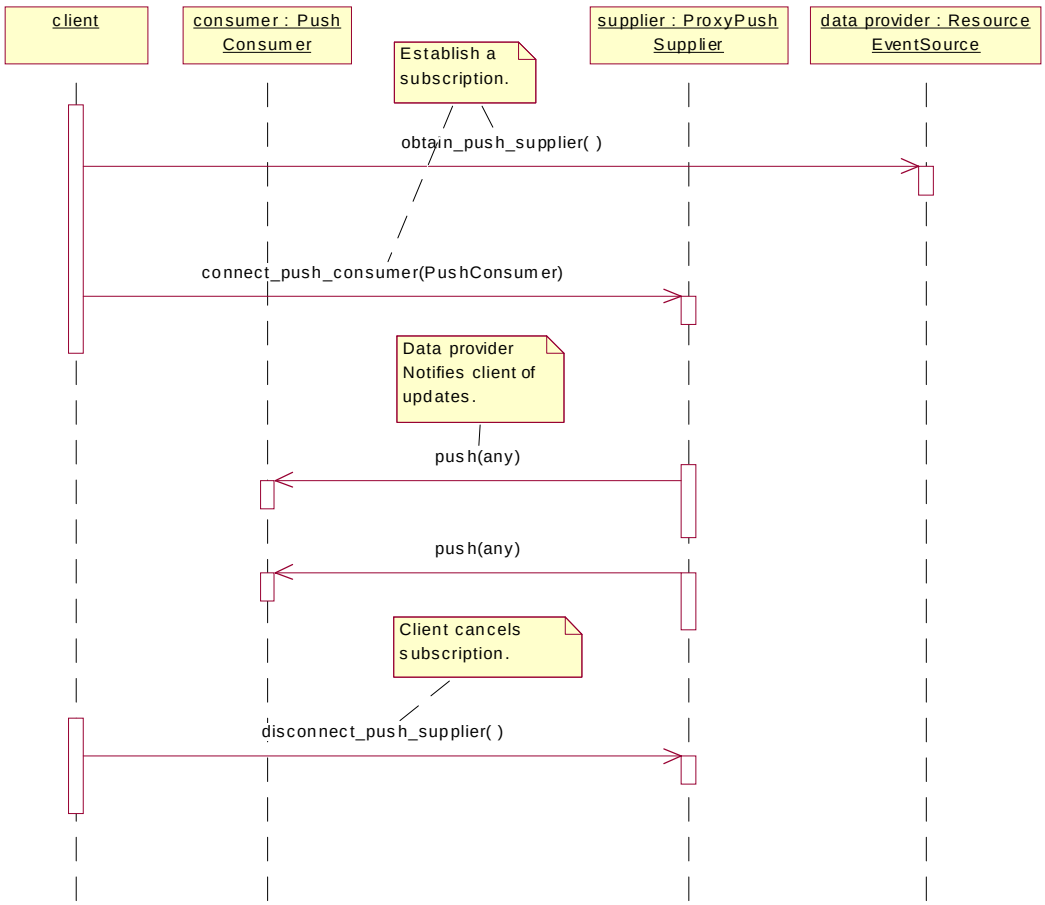

Figure 4-1 Life Cycle Diagram

4.3 *Current Version and Transactions*

The DAF interfaces are intended to be useful for a wide variety of data providers with and without transaction support. The DAF interfaces may be used in conjunction with CORBA Transactions where the data provider supports transactions.

On the other hand, DAF interfaces can be implemented by realtime, legacy, or other systems where transaction support is not available.

The following protocol must be implemented by all data providers and may be implemented by any client.

Client	Data Provider
Invokes **current_version()**	Returns X
Invokes **ResourceQueryService** operations.	Returns query results.
Invokes **current_version()**	Returns Y such that X = Y != 0 only if query results are consistent.

The definition of consistency is determined by the implementation of the data provider and is beyond the scope of this specification.

The response of the client to potentially inconsistent query results (when X != Y or Y = 0) is implementation-specific. A client may retry the same query sequence in an attempt to obtain consistent results.

A data provider that never produces inconsistent results may implement **current_version()** to return the same, non-zero value on every invocation. Assuming the value 1 was chosen, X = Y = 1 in every query scenario. As a special case, this applies to data providers that produce constant results for queries.

A data provider that implements CORBA transactions is not required to report potentially inconsistent results via **current_version()**. It may implement **current_version()** to return the same, non-zero value on every invocation. It is assumed that clients will bracket queries with a transaction if consistent results are required.

Other data providers must implement a strategy for generating **current_version()** values so that X = Y !=0 implies the query results gathered between X and Y are consistent. Examples are provided in the next section.

The protocol is illustrated in the following interaction diagram:

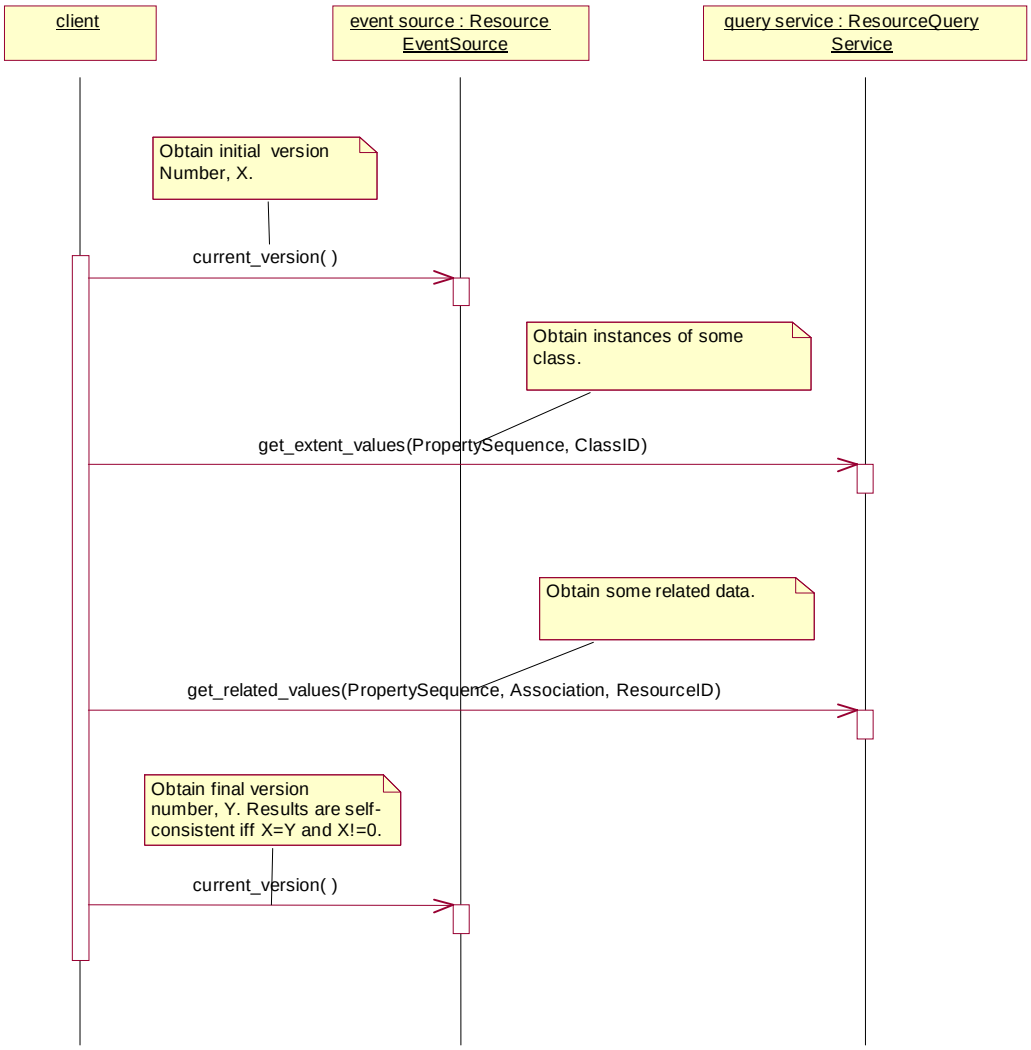

Figure 4-2 current_version() Diagram

4.3.1 Implementation Examples

This section is non-normative and other implementations are permitted. A data provider that does not implement CORBA Transactions could implement the **current_version()** operation using one of the techniques below.

4.3.1.1 Version Numbering Example

Define an internal variable, V, as follows.

i. On initialization set V to 1.

ii. On entering a new, consistent state increment V.

iii. When **current_version()** is invoked return V if the current state is consistent, otherwise return 0.

4.3.1.2 *Timestamp Example*

Define a persistent datum variable T, managed as part of the model.

i. On model creation set T to the current time.

ii. During model update suspend query service.

iii. On completion of model update set T to the current time.

iv. When **current_version()** is invoked return T.

4.4 *Event Compression*

An implementation is not required to issue an event for every update as it occurs. A data provider may be capable of grouping a series of transactions or simple updates into a larger unit. This series may constitute a logical grouping or a temporal grouping of changes. A single resource change event may be issued after the last of change of the series, to cover all of the changes. This is referred to as event compression.

4.5 *Event Handling Guidelines*

A resource change event can be issued at either of two levels of precision. A general update event contains an empty **affected** sequence, while a specific event identifies the changed data via a non-empty **affected** member. The precision used depends on both the capability of the data provider and the nature of the update. A given data provider may use either technique depending on circumstances.

To ensure interoperability, the following guidelines for events should be observed:

- If **affected** is not empty then it should identify every class of data changed since the preceding event.
- The data provider should emit exactly one event for an update or series of updates. It should not issue a general event and a specific event for the same update.
- Any client that responds to events should respond to both general and specific update events.

If a client is not capable of interpreting the **affected** information, it should treat specific update events the same way as general update events.

Service Location 5

Contents

This chapter contains the following topics.

5.1 Introduction

The goal of the service location protocol is to ensure that independently developed clients and data providers can be combined to form a coherent system, isolated from other systems that may be operating in the same host or network. Such a system is referred to as a *context*. Three context configurations are considered by this specification:

1. The simplest context consists of one client and one data provider. It is only necessary for the client to locate the data provider (and not some other data provider belonging to a different context).
2. The next most elaborate context consists of a number of clients and one data provider. Here, all of the clients must locate (the same) data provider.
3. The most general context contemplated by this specification consists of multiple clients, multiple data providers, and a single proxy data provider (described in the Proxies chapter).

Clients and data providers must be capable of operating in any of these contexts without modification.

5.2 Client View

From a client's perspective it is only necessary to locate three persistent CORBA singletons implementing **ResourceQueryService**, **ResourceIDService**, and **ResourceEventSource** respectively. Together, these services constitute a data provider.

The client locates the three services in a single **CosNaming::NamingContext** within the CORBA Naming service. The location of this **NamingContext** must be configurable for each instance of the client implementation.

The client *resolves* the three services using the following standard names.

Service	Name	Kind
ResourceQueryService	"resource_query_service"	""
ResourceIDService	"resource_id_service"	""
ResourceEventSource	"resource_event_source"	""

5.3 Data Provider View

A data provider exports the three services above. In addition, the data provider may need to locate a **ResourceIDService** to translate standard URI's to **ResourceID**s (see the Data Access Interfaces chapter).

The data provider exports its three services by *binding* the respective objects in a **CosNaming::NamingContext**, which it implements directly or creates within the CORBA Naming service implementation. It uses the same names as a client, given in the table above.

The data provider obtains the **ResourceIDService** to translate standard URIs from this **NamingContext**. It does so by resolving the following standard name.

Service	Name	Kind
ResourceIDService	"standard_id_service"	""

The location of the **NamingContext** used must be configurable for each data provider. If the context contains multiple data providers, then a different **NamingContext** will be used by each and the clients will use the **NamingContext** of a proxy data provider (see the Proxies chapter).

Proxies 6

Contents

This chapter contains the following topics.

6.1 Introduction

A *proxy* data provider is intended to enable multiple, independently developed data providers to be combined in one context. In the simpler DAF configurations no proxy is required. Multiple clients can share a single, data provider, which has a comprehensive schema and population. For example, the data provider might be a wrapper on an existing UMS.

However, a data provider that handles only part of the overall data needed by clients may be combined with others to form a complete system. This leads to a configuration in which multiple clients share multiple data providers. It would arise when the DAF interfaces are used to wrap individual data providers in a UMS rather than the UMS as a whole. This permits independent developers to extend a UMS with data providers as well as clients.

6.2 Rules for Proxy Data Providers

A proxy data provider presents a simple facade to the clients, on the one hand, and

combines the services of multiple data providers on the other. At one extreme, very simple proxies could be envisaged. At the other, the proxy may actually be a wrapper for a more extensive application integration environment, an “integration bus” or a “data federation service.”

In any case, the proxy data provider must provide the following capabilities:

- The proxy must hide the existence of multiple data providers from the clients so that they can operate unchanged in both single-data provider contexts and multiple-data provider contexts.
- The proxy must encapsulate configuration information about which data providers are present and what data they are responsible for supplying. Data providers and clients must be insulated from the configuration details.
- A proxy must not require additional interfaces to be exposed either to clients or the ultimate data providers. It acts as both a server and a client for the **ResourceQueryService**, **ResourceIDService**, and **ResourceEventSource** interfaces defined in this specification.
- A proxy must present a unified view of all the data in the context to the clients, combining information provided by the ultimate data providers. A query must return all the requested information that is available, even though it may be obtained from multiple data providers.
- A proxy may implement various policies and optimizations to delegate queries, but these must be transparent to clients and the ultimate data providers.

6.3 Proxy Resource Query Service

A proxy data provider must implement a **ResourceQueryService**. Each query invoked by a client will be directed to this service, which will delegate to one or more of the ultimate data providers. The results of the delegated queries (if there were more than one) will then be merged to form the reply to the client.

A proxy **ResourceQueryService** would therefore contain logic and configuration information for selecting the ultimate data providers, partitioning queries among them, combining the results and resolving conflicts between results.

The query interfaces are designed so that each data provider can return partial query results (without raising an exception).

6.3.1 Proxy Query Sequence

The following diagram illustrates the delegation of queries through a proxy **ResourceQueryService**.

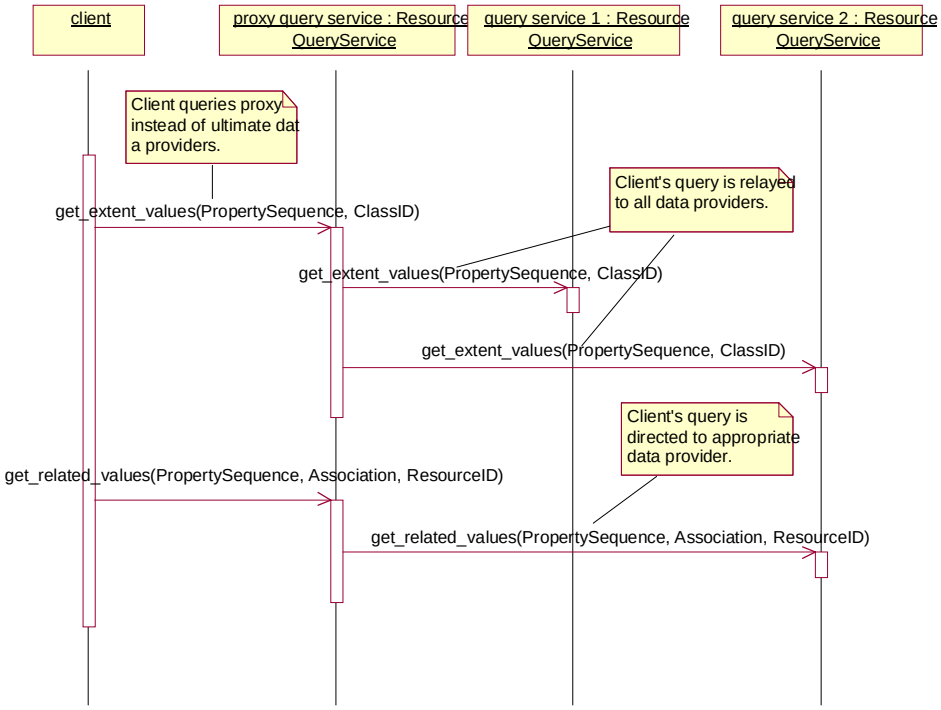

Figure 6-1 Proxy Query Sequence Diagram

6.4 Proxy Resource Identification Service

A proxy data provider must implement a **ResourceIDService**. Each translation request will be delegated to one or more of the ultimate data providers and the results combined. The translation methods are designed so that each data provider can return results for those resources known to it without raising an exception for unknown resources.

6.5 Proxy Resource Event Source

A proxy data provider must implement a **ResourceEventSource**. This involves synthesizing a **current_version()** operation and acting as an event channel for **ResourceChangeEvent**s.

The proxy must report:

- A different current version whenever the current version reported by one of the ultimate data providers changes.
- The value zero when any of the ultimate data providers reports zero.

In order to combine event sources, the proxy must connect to all of the ultimate data providers and receive their **ResourceChangeEvent**s. It must then provide an event

service to the clients. Each event from each data provider is passed on to the clients. An implementation could employ a standard CORBA Events service in this role.

Schema 7

Contents

This chapter contains the following topics.

7.1 Introduction

The information made available through DAF interfaces can be described by one or more *schema*. At least some agreement on schema must exist between a data provider and a client before effective communication is possible.

In particular, every operation on **ResourceQueryService** requires the **ResourceID**s of some properties. Moreover, the results of a query are meaningful only to the extent that the client and data provider agree on the definition of these properties.

Therefore, this specification provides:

- A standard schema for EMS applications, generated from the EPRI CIM [3].
- Provision for additional industry and application-specific schema, and revisions of the EPRI CIM.

- A means to extend standard schema or merge custom and standard schema without conflicts.
- A means to manage multiple versions of a schema.
- A means for the data provider to support a subset of a schema, and for the client to discover which parts are supported.
- A standard schema, taken from [2], for querying other schema.

7.2 Industry and Application-Specific Schema

The standard schema to be used with the DAF in EMS applications is described in the EPRI Common Information Model chapter. It is fully defined in RDF syntax in **[8]**.

Additional industry and application-specific schema can be defined in a similar way. Alternatives to RDF syntax may be used provided that the schema unambiguously specifies:

- A unique URI for the schema as a whole, which includes version identification. (Uniqueness may require the inclusion of the Internet domain name of the issuer.)
- Unique URI-references for any conformance blocks and a list of the classes and properties belonging to each conformance block. A conformance block URI-reference may be constructed from the schema URI appended with a fragment identifier. (Note that EPRI CIM version 9a does not define conformance blocks.)
- A unique URI-reference for each class and property that also includes version identification. Again, the URI-reference may be constructed from the schema URI appended with a fragment identifier.
- The domain and range of each property. (Where domain and range indicate the property *from* and *to* classes respectively, as per RDF schema.)
- The multiplicity of each property. (Whether it is single or many-valued.)
- The inverse property of each property (if present).
- Any sub-class/super-class relationships.

Clients can use an industry or application-specific schema in scenarios similar to that shown in Section 3.5, “Query Sequence,” on page3-14. Since the URI-reference of each class and property is specified, the client simply translates them to **ResourceID**s that are used in queries.

7.3 Schema Extension

A data provider may implement extensions to a standard schema, such as the EPRI CIM or the RDF schema (described below). Standard schema may be extended provided compatibility with standard clients is maintained. Extensions may be made as follows:

- The extension may add classes and/or properties.
- Classes may be added which are sub-classes or super-classes of standard classes.

- Properties may be added whose domain and/or range is a standard class.
- The URI of each added class or property must be unique. (Uniqueness may require the inclusion of the Internet domain name of the extender.)
- The definition of any standard property may not be changed.
- No sub-class/super-class relationships between standard classes may be altered.

Unlike standard schema, an extension is not necessarily agreed by more than one independent software developer. Therefore schema extensions are not necessarily published according to the rules in Section 7.2, “Industry and Application-Specific Schema,” on page7-2.

An implementation that extends a nominally standard schema in an incompatible way (violating one or more of the foregoing rules) must rename the resulting schema or the affected elements so as not to conflict with a standard implementation.

7.4 Schema Support Testing

An implementation may support all, part, or none of a given standard schema. In addition, an implementation may support one or more versions of a standard schema. The scope of support is indicated to the client by the **get_resource_ids()** operation.

A client may test a collection of schema features with a **get_resource_ids()** operation at any time, possibly preceding any other interactions with the data provider. The result of the operation will indicate which of the listed features are supported and which are not. The client might use this information to decide whether it can proceed normally, whether it can proceed with limited capabilities, or to choose an internal strategy that will be compatible with the data provider.

The schema features that can be tested in this way are as follows.

7.4.1 Schema and Version Test

If some or all of a schema of a particular version is supported, then the schema URI for that version will be translated to a non-null resource identifier. For example the schema URI for EPRI CIM 9a is http://iec.ch/TC57/2000/CIM-schema-cimu09a and this URI can be tested to determine if general support of that CIM version is available.

7.4.2 Conformance Block Test

If an entire conformance block of a particular schema and version is supported, then the URI-reference for that conformance block will be translated to a non-null resource identifier. This differs from schema discovery in that complete conformance is tested.

7.4.3 Class and Property Test

If a class is supported (with all, some, or none of its properties), then the URI-reference for that class will be translated to a non-null resource identifier. Similarly, if a property is

supported, then its URI-reference will be translated to a non-null resource identifier.

7.4.4 Schema Query Test

Support for schema queries (described in the next section) can also be tested. Schema query is available if the URI http://www.w3.org/2000/01/rdf-schema translates to a non-null resource identifier.

7.5 Schema Query

Some clients may want to query the schema information in greater detail than is possible with the feature tests defined in the previous section. This specification aims to accommodate these clients without requiring every data provider to implement a full schema query facility[1].

If a data provider supports schema queries, they are performed via its **ResourceQueryService** interface in the same way as general population queries. The meta-model supporting schema query is derived from the RDF schema candidate recommendation [2].

A data provider that supports schema query must, at a minimum, translate the following URI-references to **ResourceID**s and implement the designated classes and properties in its **ResourceQueryService**.

URI	Meaning
http://www.w3.org/2000/01/rdf-schema#Class	The class of all classes.
http://www.w3.org/2000/01/rdf-schema#Property	The class of all properties.
http://www.w3.org/2000/01/rdf-schema#label	A property of **Class** and **Property** giving a short name.
http://www.w3.org/2000/01/rdf-schema#comment	A property of **Class** and **Property** giving a human-readable description.
http://www.w3.org/2000/01/rdf-schema#domain	The domain property of **Property** (see Section 2.1.1.2, “Properties,” on page 2-2).

1. Experience with developing data providers indicates that the schema query capabilities add significant cost and complexity and may even dominate the overall implementation cost. The design rationale adopted here is that implementations should only pay for what they use.

http://www.w3.org/2000/01/rdf-schema#range	The range property of **Property** (see Section 2.1.1.2, “Properties,” on page 2-2).
http://iec.ch/TC57/1999/rdf-schema-extensions-19990926#Multiplicity	The multiplicity property of **Property**.
http://iec.ch/TC57/1999/rdf-schema-extensions-19990926#M:0..n	The value of multiplicity which indicates a many-valued property.

A proxy data provider is required when clients require schema query features but the ultimate data providers do not support the foregoing metadata. The proxy must supply the missing metadata, which may be available from a third data provider.

7.6 Schema Query Sequence

The following sequence diagram illustrates the use of the **ResourceQueryService** to query schema information. This sequence should be compared with that of Secti on3.5, “Query Sequence,” on page3-14. The client begins by obtaining resource identifiers for **Class**, **Property domain**, **range**, and **label** metadata. These resource identifiers may then be used in schema queries. For example, to ask for the names of all classes the **Class** and **label** resource identifiers are passed to the **get_extent_values()** query.

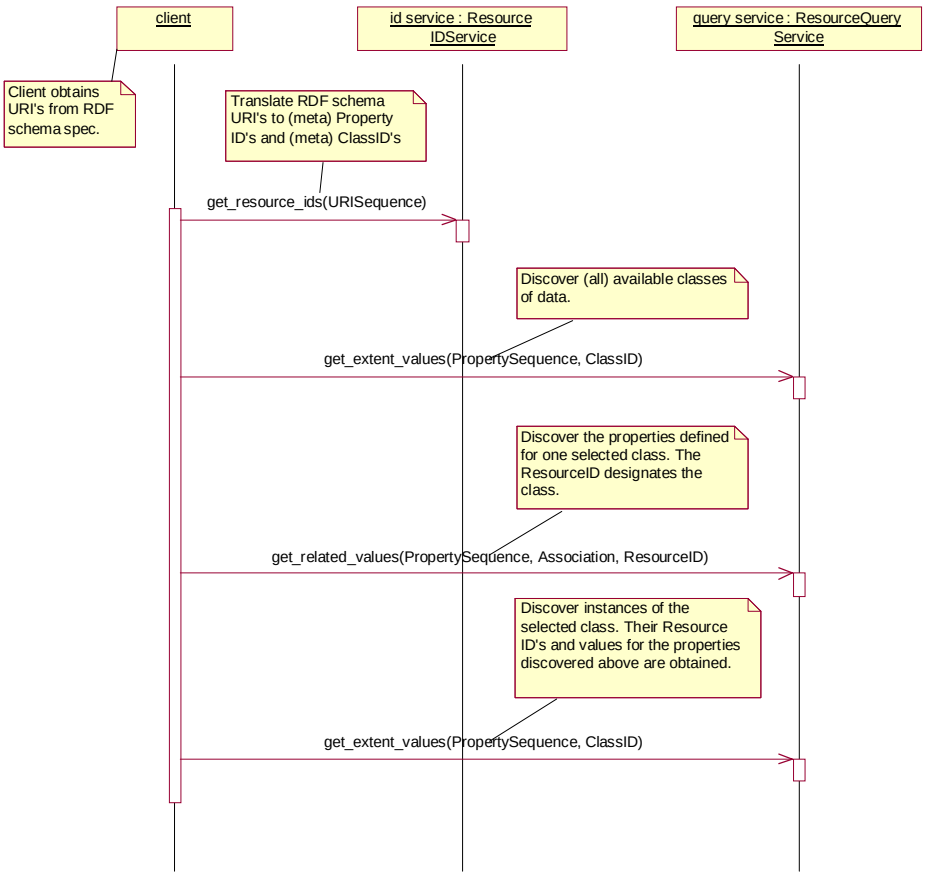

Figure 7-1 Schema Query Sequence Diagram

EPRI Common Information Model 8

Contents

This chapter contains the following topics.

8.1 Schema

When the DAF is used to access power system model data, a schema derived from the EPRI CIM will be employed. This schema is undergoing standardization in the IEC under Technical Committee TC57. From time to time, the IEC may standardize newer versions of this schema, which the OMG may designate as the DAF standard. When this happens a new OMG document number will be issued.

There are two URIs designating the EPRI CIM schema:

1. The EPRI CIM document URI gives the location of the schema document on the public internet. This URI points to the OMG document dtc/2000-11-10 [8] and is subject to change if the OMG rearranges its web site. The document URI plays no part in the operation of the interface. Neither clients nor data providers need to recognize the document URI.

2. The EPRI CIM namespace URI identifies a schema, including its version. This URI is used purely as a name and it does not necessarily identify any document on the public internet. It is used in the interface to uniquely and universally identify CIM classes, properties, and enumerated values. Implementations complying with the EPRI CIM conformance point must recognize this URI in the **ResourceIDService::get_resource_ids()** operation.

The EPRI CIM namespace URI will normally take the following form:

http://iec.ch/TC57/2000/CIM-schema-<version>

The current version of this schema has the following namespace URI:

http://iec.ch/TC57/2000/CIM-schema-cimu09a

8.2 Mapping

As the EPRI CIM is updated it will be necessary to create new versions of the schema used with the DAF. The EPRI CIM is published in UML and the following guidelines are to be used in mapping the UML form to RDF schema. In the following mapping table the namespaces are:

rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
rdfs="http://www.w3.org/2000/01/rdf-schema#"

UML	RDF Schema Mapping	
	Schema Element Property	**Schema Element Value**
Class	Type	rdfs:Class
	ID	UML class name
Attribute	Type	rdf:Property
	ID	C.a where C is the class name and a is the UML attribute name with first letter lower case and subsequent words capitalized.
	Domain	The RDF class corresponding to the attribute's class.
	Range	The RDF class corresponding to the attribute's type.
Association role	Type	rdf:Property
	ID	C.A where C is the class name and A is the UML attribute name with first letter upper case and subsequent words capitalized.
	Domain	The RDF class corresponding to the attribute's class.
	Range	The RDF class corresponding to the attribute's type.

References A

A.1 List of References

1. *Resource Description Framework (RDF) Model and Syntax Specification*, W3C Recommendation 22 February 1999 http://www.w3.org/TR/REC-rdf-syntax, Ora Lassila, Ralph R. Swick.
2. *Resource Description Framework (RDF) Schema Specification*, W3C Candidate Recommendation 27 March 2000 http://www.w3.org/TR/2000/CR-rdf-schema-20000327, Dan Brickley, R.V. Guha, Netscape
3. *Uniform Resource Identifiers (URI): Generic Syntax*; Berners-Lee, Fielding, Masinter, Internet Draft Standard August, 1998; RFC2396.
4. *Namespaces in XML*; Bray, Hollander, Layman eds, W3C Recommendation; http://www.w3.org/TR/1999/REC-xml-names-19990114
5. *Naming Service Specification*, OMG Document formal/97-12-10
6. *Event Service Specification*, OMG Document formal/97-12-11
7. *Guidelines for Control Center Application Program Interfaces*, EPRI Technical Report TR-106324, Project 3654-01, Final Report, June 1996.
8. *CIM RDF Schema Exported from Ccapi.mdl Version: CIMU09a*, Leila Schneberger. OMG TC Document dtc/2000-11-10.
9. *Additional Metadata Definitions for CIM*, Leila Schneberger. OMG TC Document dtc/2000-11-11.
10. *IDL for UMS DAF Specification*. OMG TC Document dtc/2000-11-12.
11. *UML Model for UMS DAF*, Arnold deVos. OMG TC Document dtc/2000-11-13.

A

Complete IDL B

B.1 DAFIdentifiers Module

```
//File: DAFIdentifiers.idl
#ifndef _DAF_IDENTIFIERS_IDL_
#define _DAF_IDENTIFIERS_IDL_

#pragma prefix "omg.org"
module DAFIdentifiers
{
    // the uniform resource identifier from IETF RFC 2396
    typedef string URI;
    typedef sequence< URI > URISequence;

    // the resource identifier
    struct ResourceID
    {
        unsigned long long container;
        unsigned long long fragment;
    };
    typedef sequence < ResourceID > ResourceIDSequence;

    // service for translating and managing resource identifiers
    exception LookupError {      string reason; };

    interface ResourceIDService
    {
        ResourceIDSequence get_resource_ids(
            in URISequence uris )
            raises( LookupError );

        URISequence get_uris(
            in ResourceIDSequence ids )
            raises( LookupError );
    };
};
```

```
#endif // _DAF_IDENTIFIERS_IDL_
```

B.2 DAFDescriptions Module

```
//File: DAFDescriptions.idl
#ifndef _DAF_DESCRIPTIONS_IDL_
#define _DAF_DESCRIPTIONS_IDL_

#include <DAFIdentifiers.idl>
#include <TimeBase.idl>

#pragma prefix "omg.org"
module DAFDescriptions
{
    //++
    // Simple Types used as property values.
    //--

    // imported from identifiers module.
    typedef DAFIdentifiers::ResourceID ResourceID;
    typedef DAFIdentifiers::URI URI;

    // absolute time stamps in 100 nanosecond units
    // base time is 15 October 1582 00:00 UTC
    // as per Time Service specification
    typedef TimeBase::TimeT DateTime;

    // a complex number
    struct Complex
    {
        double real;
        double imaginary;
    };

    // SimpleValue's can take on the following types.
    typedef short SimpleValueType;
    const SimpleValueType RESOURCE_TYPE = 1;
    const SimpleValueType URI_TYPE = 2;
    const SimpleValueType STRING_TYPE = 3;
    const SimpleValueType BOOLEAN_TYPE = 4;
    const SimpleValueType INT_TYPE = 5;
    const SimpleValueType UNSIGNED_TYPE = 6;
    const SimpleValueType DOUBLE_TYPE = 7;
    const SimpleValueType COMPLEX_TYPE = 8;
    const SimpleValueType DATE_TIME_TYPE = 9;
    const SimpleValueType ULONG_LONG_TYPE = 10;

    // a SimpleValue is the object of a resource description.
    union SimpleValue switch( SimpleValueType )
    {
        case RESOURCE_TYPE  : ResourceID resource_value;
        case URI_TYPE       : URI uri_value;
        case STRING_TYPE    : string string_value;
        case BOOLEAN_TYPE   : boolean boolean_value;
```

```
        case INT_TYPE      : long int_value;
        case UNSIGNED_TYPE  : unsigned long unsigned_value;
        case DOUBLE_TYPE    : double double_value;
        case COMPLEX_TYPE   : Complex complex_value;
        case DATE_TIME_TYPE : DateTime date_time_value;
        case ULONG_LONG_TYPE: unsigned long long ulong_long_value;
    };

    //++
    // Resource Descriptions
    //--

    // properties are represented by their resource identifiers
    typedef ResourceID PropertyID;
    // predicate and object for a resource description
    struct PropertyValue
    {
        PropertyID property;
        SimpleValue value;
    };
    typedef sequence<PropertyValue> PropertyValueSequence;

    // resource description with one subject, multiple predicates
    struct ResourceDescription
    {
        ResourceID id;
        PropertyValueSequence values;
    };
    typedef sequence<ResourceDescription> ResourceDescriptionSequence;

    // iterator for handling large numbers of resource descriptions
    interface ResourceDescriptionIterator
    {
        unsigned long max_left();
        boolean next_n(
            in unsigned long n,
            out ResourceDescriptionSequence descriptions );
        void destroy();
    };
};

#endif // _DAF_DESCRIPTIONS_IDL_
```

B.3 DAFQuery Module

```
//File: DAFQuery.idl
#ifndef _DAF_QUERY_IDL_
#define _DAF_QUERY_IDL_

#include <DAFDescriptions.idl>

#pragma prefix "omg.org"
module DAFQuery
{
```

```
        // properties and classes are represented by resource identifiers
        // imported from the identifiers module.
        typedef DAFIdentifiers::ResourceID ResourceID;
        typedef DAFIdentifiers::ResourceID ClassID;
        typedef DAFIdentifiers::ResourceID PropertyID;
        typedef DAFIdentifiers::ResourceIDSequence PropertySequence;

        // results are resource descriptions from the descriptions module
        typedef DAFDescriptions::ResourceDescription ResourceDescription;
        typedef DAFDescriptions::ResourceDescriptionIterator
                ResourceDescriptionIterator;

        // queries that perform navigation use the association concept
        struct Association
        {
            PropertyID   property;
            ClassID      type;
            boolean      inverse;
};
        typedef sequence<Association> AssociationSequence;

        // exceptions generated by queries
        exception UnknownAssociation { string reason; };
        exception UnknownResource { string reason; };
        exception QueryError { string reason; };

        // the query service
        interface ResourceQueryService
        {
            ResourceDescription get_values(
                in ResourceID resource,
                in PropertySequence properties)
                raises( UnknownResource, QueryError );

            ResourceDescriptionIterator get_extent_values(
                in PropertySequence properties,
                in ClassID class_id )
                raises (UnknownResource, QueryError);

            ResourceDescriptionIterator get_related_values(
                in PropertySequence properties,
                in Association association,
                in ResourceID source )
                raises ( UnknownResource, UnknownAssociation, QueryError );

            ResourceDescriptionIterator get_descendent_values(
                in PropertySequence properties,
                in AssociationSequence path,
                in ResourceIDSequence sources,
                out AssociationSequence tail )
                raises ( UnknownResource, QueryError );
        };
};
#endif // _DAF_QUERY_IDL_
```

B.4 DAFEvents Module

```
//File: DAFEvents.idl
#ifndef _DAF_EVENTS_IDL_
#define _DAF_EVENTS_IDL_

#include <DAFIdentifiers.idl>
#include <CosEventChannelAdmin.idl>

#pragma prefix "omg.org"
module DAFEvents
{
    // event emitted by data provider after data changes
    struct ResourceChangeEvent
    {
        DAFIdentifiers::ResourceIDSequence affected;
    };

    // interface for connection an event push consumer
    interface ResourceEventSource
    {
        CosEventChannelAdmin::ProxyPushSupplier obtain_push_supplier();
        unsigned long long current_version();
    };
};
#endif // _DAF_EVENTS_IDL_
```

SQL Examples C

This is a non-normative appendix and does not constitute part of the definition of the DAF. It is provided for illustration only.

The four queries provided by the resource query service can be understood in terms of their SQL equivalents. In comparing the SQL form to the IDL, it should be noted that the IDL encompasses an API as well as a query formulation. That is, the DAF IDL combines a (vastly simplified) equivalent to the ODBC API and the SQL queries given below.

C.1 SQL Identifiers

In the following illustration, the SQL identifiers are

`property1, property2 ...`

Attributes corresponding to the properties in the property sequence in the DAF query.

`implied-class ...`

The table corresponding to the class to which the given resource belongs. Given a resource identifier, an implementation needs to be able to determine the class of the resource.

`Class`

The table corresponding to the class_id in the DAF query.

`resource, source ...`

The key value of a record that represents a resource.

`association-type, path-type ...`

The table corresponding to the Association.type in the DAF query.

`association-inverse, path-inverse`

The attribute corresponding to the inverse of the Association.property in the DAF query.

`path1, path2 ...`

The associations in the path sequence in the DAF query.

C.2 get_values()

IDL

```
ResourceDescription get_values(
    in ResourceID resource,
    in PropertySequence properties)
    raises( UnknownResource, QueryError );
```

SQL

```
select property1, property2 ... from implied-class where
implied-class.ID = resource;
```

C.3 get_extent_values()

IDL

```
ResourceDescriptionIterator get_extent_values(
    in PropertySequence properties,
    in ClassID class_id )
    raises (UnknownResource, QueryError);
```

SQL

```
select property1, property2 ... from class;
```

C.4 get_related_values()

IDL

```
ResourceDescriptionIterator get_related_values(
    in PropertySequence properties,
    in Association association,
    in ResourceID source )
    raises ( UnknownResource, UnknownAssociation, QueryError );
```

SQL

```
select property1, property2 ...
from association-type
where association-type.association-inverse = source;
```

C.5 get_descendent_values()

IDL

```
ResourceDescriptionIterator get_descendent_values(
    in PropertySequence properties,
    in AssociationSequence path,
    in ResourceIDSequence sources,
    out AssociationSequence tail )
    raises ( UnknownResource, QueryError );
```

SQL

```
select property1, property2 ... from implied-view;
```

SQL (alternative)

```
select property1, property2 ...
from path1-type, path2-type, path3-type..
where path1-type.path1-inverse in sources
   and path2-type.path2-inverse = path1-type.ID
   and path3-type.path3-inverse = path2-type.ID
   and ...;
```

Glossary

Glossary Terms

DAF	The Utility Management System Data Access Facility, the subject of this specification.
DAF Client	A program or software entity that uses the DAF interfaces to obtain information. Abbreviated to client in most of this specification.
Data Provider	An implementation of the DAF. That is, a program or software entity that supplies information via the DAF interfaces. Also referred to as a **DAF server** or just a **server**.
DMS	A **Distribution Management System**. This is a UMS for operating an electric power sub-transmission and distribution system.
EMS	An **Energy Management System**. This is a UMS for operating an electric power main transmission and/or production system.
EPRI	Electric Power Research Institute. A power industry body that is engaged in an effort to define APIs and data models for EMS systems and applications.
EPRI CIM	The EPRI Common Information Model. A data model defined in UML that can be used to describe power systems and related concepts. This model is undergoing standardization in the IEC.
Power System	The integrated facilities and resources that produce, transmit and/or distribute electric energy.

PLC	Programmed Logic Controller, a device that controls an item or items of equipment. A PLC may transmit data it gathers to a UMS and receive control commands from the UMS. In this case it fills a role similar to an RTU.
RDF	**Resource Description Framework**. A model of data that has been defined by a W3C recommendation and is used in conjunction with XML notation.
RTU	Remote Terminal Unit, a device located at a (usually) remote site that connects equipment with a central UMS. An RTU gathers data from equipment, and transmits that data back to the UMS. It also receives commands from the UMS and controls the equipment.
SCADA	Supervisory Control and Data Acquisition, a system that gives operators oversight and control of geographically dispersed facilities.
UML	Unified Modeling Language. The OMG standard modeling language, which has been used to define the EPRI Common Information Model.
UMS	Utility Management System, a control system that incorporates simulation and analysis applications used by a water, gas or electric power utility for operations or operational decision support.
WQEMS	A **Water Quality and Energy Management System**. This is a UMS for operating water supply and/or waste water systems.
XML	Extensible Markup Language. A generic syntax defined by a W3C recommendation that can be used to represent UMS data and schema, among other things.

Index

Index