Data Acquisition from Industrial Systems Specification

Draft Adopted Specification
July 2001

Contents

Preface v

1. Overview 1-1

1.1 Introduction 1-1

1.2 Problems Being Addressed 1-3

1.2.1 Data Access 1-4

1.2.2 Concurrency Control 1-5

1.2.3 Data Semantics 1-5

1.3 Problems Not Being Addressed 1-5

1.4 Design Rationale 1-5

1.4.1 Adherence to OPC 1-5

1.4.2 Simplicity and Uniformity 1-5

1.4.3 High Performance Implementations 1-6

1.5 Conformance to the DAIS 1-7

1.5.1 Conformance to the Server 1-7

1.5.2 Conformance to Data Access 1-7

1.5.3 Conformance to Alarms and Events 1-7

2. Relations to Other Standards 2-1

2.1 OLE for Process Control (OPC) 2-1

2.1.1 Objects 2-1

2.2 Data Access Facility (DAF) 2-4

2.2.1 Resources and Properties 2-4

2.2.2 Information model/schema 2-5

2.2.3 Data Types 2-6

2.3 COM and CORBA IDL Differences 2-6

2.3.1 Object Referencing 2-6
2.3.2 Interface Management 2-7
2.3.3 Error Management 2-7
2.3.4 IDL 2-7
2.4 IEC 1346-1, Structuring and Naming 2-7
2.5 IEC 61970 EMS API 2-7

3. DAIS Server 3-1
3.1 Common Declarations 3-1
3.1.1 Common IDL Overview 3-1
3.1.2 DAFIdentifiers IDL 3-2
3.1.3 DAFDescriptions IDL 3-2
3.1.4 DAISCommon IDL 3-2
3.1.5 Iterator Methods IDL 3-10
3.1.6 DAISNode IDL 3-11
3.1.7 DAISType IDL 3-15
3.1.8 DAISProperty IDL 3-18
3.1.9 DAISSession IDL 3-21
3.2 Server 3-24
3.2.1 DAISServer IDL Overview 3-24
3.2.2 DAISServer IDL 3-24

4. DAIS Data Access 4-1
4.1 Information Model 4-1
4.1.1 Nodes, Items, Types, and Properties 4-2
4.1.2 Naming 4-3
4.1.3 Item Values 4-4
4.1.4 OPC Recommended Properties 4-5
4.1.5 Utility SCADA/EMS Measurement Model . . . 4-6
4.2 API 4-8
4.2.1 Data Access IDL Overview 4-8
4.2.2 DAISDASession IDL 4-9
4.2.3 DAISDANode IDL 4-11
4.2.4 DAISItem IDL 4-14
4.2.5 DAISDAIO IDL 4-22
4.2.6 DAISGroupEntry IDL 4-31
4.2.7 DAISGroup IDL 4-37

5. Alarms & Events 5-1
5.1 Information Model 5-1
5.1.1 OPC Recommended Properties 5-4

5.2 API 5-5
5.2.1 Alarms & Events IDL Overview 5-5
5.2.2 Alarms and Events Common IDL Definitions . 5-6
5.2.3 DAISAESession IDL 5-7
5.2.4 DAISAESubscription IDL 5-10
5.2.5 DAISAEArea IDL 5-20
5.2.6 DAISAESource IDL 5-22
5.2.7 DAISConditionSpace IDL 5-25
5.2.8 DAISAESourceCondition IDL 5-31
5.2.9 DAISReason IDL 5-36
5.2.10 DAISAEIO IDL 5-40

Appendix A - References A-1

Appendix B - OMG IDL B-1

Appendix C - UML Model C-1

Appendix D - Glossary 1

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 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.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set 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*.)

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
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 parts of this specification:

- ABB Automation Systems
- Alstom ESCA
- Langdale Consultants

Supporting Organizations

The following organizations have been involved in the process of contributing to, and reviewing this specification. These companies have indicated their support for the specification. We thank them for participating and giving their valuable input.

- Hitachi Ltd.
- IBM Corporation
- NIIP Project Office
- SISCO Inc.

Overview 1

Contents

This chapter contains the following sections.

Section Title	Page
"Introduction"	1-1
"Problems Being Addressed"	1-3
"Problems Not Being Addressed"	1-5
"Design Rationale"	1-5
"Conformance to the DAIS"	1-7

1.1 Introduction

The purpose of the DAIS API is to support efficient real time transfer of large amounts of data from an industrial process to a wide range of clients. It supports discovery of parameters and update of parameter values. The DAIS is intended for on-line data transfer and can not be used to configure servers implementing the API.

Control systems used to monitor and control industrial processes consists of the following major pieces:

- Process instrumentation making sensor data and actuation capabilities available.
- Process stations or remote terminal units (RTUs) reading sensor data and controlling actuators.
- SCADA system making processed sensor data and control capabilities available to operators, applications or other systems.
- Management systems using SCADA data to make further processing and control.

SCADA and management systems can be regarded having a server part where data processing is performed and a HMI part where visualization and command dialogs are made. The SCADA and management system may have common or different HMI.

This results in a hierarchical structure shown below.

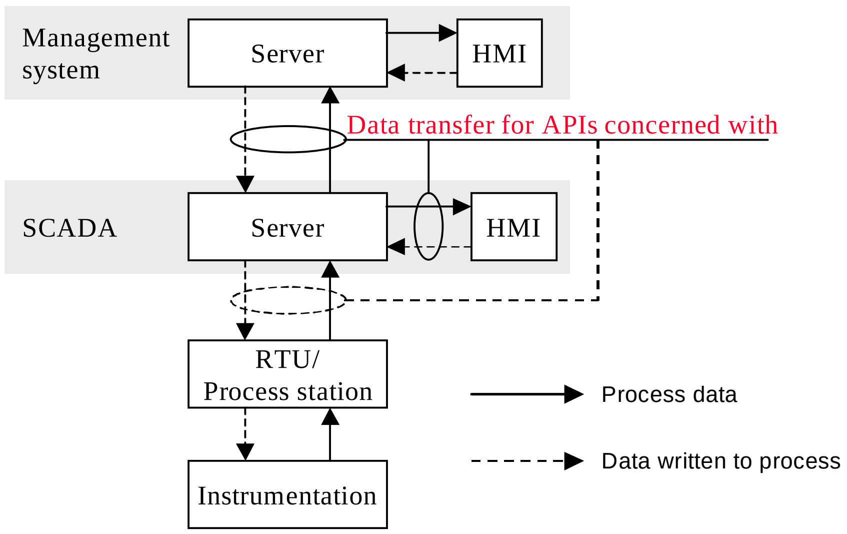


Figure 1-1 Control system structure

The solid arrows in Figure 1-1correspond transfer of data from servers to clients. The data can be state variables and parameters (for example, measured values, calculated values, limit values, dead bands, scan rates, engineering units).

The dashed arrows correspond to data written to servers by clients. The data can be control variables and parameters. The API support simple writes operations. More complex controls like select and execute (for example, breaker on/off) or raise lower commands can be implemented combining multiple write and read operations.

As indicated in Figure 1-1 the DAIS API can be used at several levels in a system. For example, a DAIS server can be an RTU/Process station communication unit, a SCADA server or even a Management system.

DAIS support both subscription and read/write operations. A subscription means server has no *a priori* knowledge of clients and it is clients that establish connection with servers. Once connection is established a server calls the clients back when data becomes available or updated. The callbacks mean the DAIS API also defines an interface that the client has to implement.

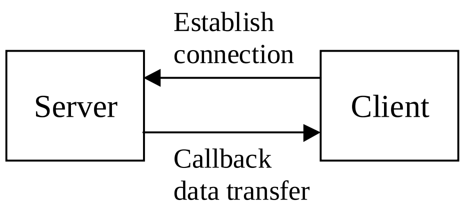


Figure 1-2 Data subscription

Figure 1-2 shows the bi-directional subscription with connection establishment and callback interfaces.

For historical reasons SCADA systems for different industrial processes has evolved along different lines. SCADA for power systems has evolved onto a UNIX base and SCADA for most other industrial processes has evolved onto a Windows NT base. For UNIX based systems APIs formulated in CORBA IDL is now emerging (for example, DAIS RFP [1] and UMS DAF [2]). For Windows NT based systems as OLE for Process Control (OPC) [3] has become the dominating standard. OPC defines three different APIs; measurement data access [4], alarms & events [5] and history [6]. OPC is based on Microsoft COM.

OPC is focused on the interfaces and does not explicitly describe the information model behind the interface. The information model is however implicit and can be derived from the OPC specifications. This specification describes both the API and information model expressed in UML.

Within the CCAPI project an information model, Common Information Model (CIM) [7] has been developed. This model is now evolving into the IEC 61970-30x draft standard [8]. The CIM contains a SCADA information model (61970-303) with its roots in power transmission and generation. The DAIS API supports the CIM.

1.2 Problems Being Addressed

The DAIS API is intended for transfer of process data on subscription basis as indicated in Figure 1-1 and Figure 1-2. Process data consists of quality tagged and time stamped scalar values. The API is intended to efficiently transfer large amounts of data simultaneously to many clients (subscribers). Clients and servers involved in data exchange can be of many kinds (for example, HMI or management systems as indicated in Figure 1-1). A client may also appear as a server (for example, aggregating data from other servers or performing calculations as indicated in Figure 1-3). This creates hierarchical structures of DAIS servers.

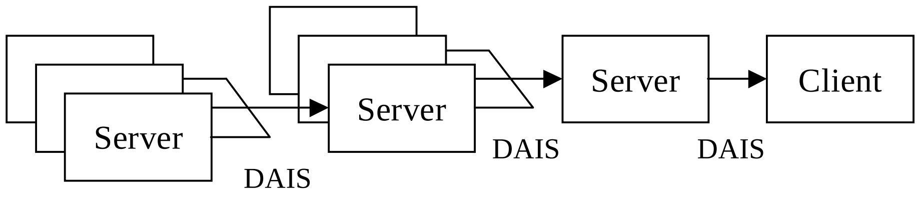


Figure 1-3 Using DIAS as interface between multiple servers

As an example the servers in the leftmost layer in Figure 1-3 might be OPC compliant RTUs (or IEDs), next right might be communication front ends and the rightmost server may provide both telemetered and estimated data.

The DAIS API is intended to be used for a wide range of industrial processes. For example:

- power transmission
- power generation
- power distribution
- water and sewage management
- oil and gas
- district heating
- pulp and paper
- food manufacturing

The kinds of data that can be reached through the DAIS API are:

- measurement data access
- alarms & events access

This data is typically available from hardware units in the process (for example, RTUs, PLCs, distributed controllers) or other control centers (for example, SCADA systems). Refined or calculated data, parameters and alarms & events might also come from applications (for example, custom calculations, state estimation, optimization) in SCADA or Management systems. These data might be provided through the DAIS API as well.

1.2.1 Data Access

This specification provide interfaces for data access including:

- Discovery of data available in a server.

- Discovery of the information model supported by a server (for example, available types and their properties).
- Synchronous and asynchronous read or write of server data.
- Creation and maintenance of subscriptions at the server.
- Client side subscription callback interfaces for event driven data transfer.

1.2.2 Concurrency Control

There are no explicit means to synchronize clients. Time stamping of data is provided so that clients can judge the age.

1.2.3 Data Semantics

Data is hierarchically organized in trees of nodes and items where the items are the leafs. The nodes in the hierarchy have a type (for example, substation, pump, breaker, or any collection of items). The type is transparent to the interface. An item is an instance of a property belonging to a type. Properties can describe any kind of state variables, control variables or parameters existing in a control system.

Data transferred by a reads or callbacks are time stamped and quality coded.

1.3 Problems Not Being Addressed

The problem of configuring a server or control system with nodes, items, areas, sources, reasons, condition spaces and source conditions is outside the scope of this specification.

The problem of supervising the control system equipment (for example, communication lines, computers, hard disks) and provide specific interfaces for this is outside the scope of this specification. It is however possible for a DAIS server to provide control system statuses as measurements.

1.4 Design Rationale

Besides meeting the requirements spelled out in the RFP there are a number of design goals that have shaped solutions.

1.4.1 Adherence to OPC

OPC has been in use for a number of years today and this specification leverage on the experience gained by OPC. There are a large number of OPC based products in the market place and cases where DAIS and OPC will be bridged are likely. Adherence to OPC is important to facilitate simple bridging and porting DAIS software to/from OPC.

1.4.2 Simplicity and Uniformity

Some design principles used when creating OPC where

- method behavior is sometimes controlled by an input parameter.
- related data is transferred in multiple parallel vectors.
- outputs are always returned in one or more output parameters.

To simplify and get a more uniform interface these principles has been replaced by the following:

- A method has one single behavior resulting in some OPC methods being replaced by more than one DAIS method.
- Related data is kept together in structs resulting in reduction of the number parameters compared to OPC.
- Outputs are returned as method return results resulting in the OPC HRESULT parameter being replaced by exceptions and reduced number of output parameters compared to OPC.

1.4.3 High Performance Implementations

A DAIS server is a real-time system required to deliver data in high rates and volumes. The performance requirements mean that a typical DAIS server does not use a relational database management system for on-line operation but some kind of real-time database. The DAIS API efficiently encapsulates such real-time databases from clients.

To be effectively deliver data DAIS must not introduce performance bottlenecks of its own. This has influenced the design in several ways, listed below.

1.4.3.1 Subscription

The subscription mechanism consists of two phases. In the first the client negotiate with the server on what data items to subscribe for and in the second the actual data transfer take place. This minimizes the amount of transferred data between the server and the client during on-line operation.

1.4.3.2 Sequences

DAIS support to use sequences of data in calls rather than having calls requiring single valued parameters. This allows clients to ask for processing of multiple data in a single call rather than making multiple calls thus reducing the number of LAN round trips.

1.4.3.3 Iterators

Large volumes of data are not efficiently transferred in one method call. For this reason many methods return an iterator that is used to transfer optimal volumes of data in each call.

1.4.3.4 *Data Value Representation*

The basic unit of data 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.

1.5 *Conformance to the DAIS*

The DAIS has three major conformance points:

1. The DAIS Server
2. The DAIS Data Access
3. The DAIS Alarms and Events.

An implementation

- shall conform to point a.
- shall conform to either point b or point c.
- may conform to both point b and point c.

1.5.1 *Conformance to the Server*

The DAIS interface as defined in Section 3.2, "Server," on page 3-24 is a mandatory conformance point for a DAIS Server with the exception of the following optional methods

- DAIS::Server::find_views()
- DAIS::Server::create_data_access_session_for_view()
- DAIS::Server::create_alarms_and_events_session_for_view()
- DAIS::Server::inspect()

1.5.2 *Conformance to Data Access*

The DAIS interface as defined in Section 3.1, "Common Declarations," on page 3-1 and Section 4.2, "API," on page 4-8 is a mandatory conformance point for Data Access.

1.5.3 *Conformance to Alarms and Events*

The DAIS interface as defined in Section 3.1, "Common Declarations," on page 3-1 and Section 5.2, "API," on page 5-5 is a mandatory conformance point for Alarms and Events.

Relations to Other Standards 2

Contents

This chapter contains the following sections.

Section Title	**Page**
"OLE for Process Control (OPC)"	2-1
"Data Access Facility (DAF)"	2-4
"COM and CORBA IDL Differences"	2-6
"IEC 1346-1, Structuring and Naming"	2-7
"IEC 61970 EMS API"	2-7

2.1 OLE for Process Control (OPC)

Differences and similarities between OPC and DAIS are described in this chapter.

2.1.1 Objects

OPC is a service API providing access to data managed by the server. The data (for example, nodes, items, areas, sources, conditions, reasons) are not instantiated as objects at the client. This means that OPC does not define any particular APIs for the data instances that a client can deal directly with. OPC has a few coarse objects like OPCGroup and OPCEventSubscription supporting the data access. DAIS has adopted this principle and is identical to OPC in this respect.

In OPC each client has its own OPC server object. In DAIS there is only one DAIS server object shared by all clients. To support individual client sessions a new session interface is defined. There is one interface for data access session objects and one for alarms & events session objects. The session objects correspond to the OPC server object.

The OPC server object has interfaces for browsing server data (for example, **IOPCBrowseServerAddressSpace** and **IOPCEventAreaBrowser**) and information model (for example, **IOPCItemProperties**). In DAIS each type of data has an own object for browsing, these objects are called “home” objects. The OPC browse interface methods are hence divided among the different home objects.

2.1.1.1 Interface, method and parameter naming

Many OPC interface, method, and parameter names have been kept but re-casted according to the CORBA style guide. But many OPC names have been replaced by new names to more clearly indicate the meaning. This is particularly the case for the browse and alarms & events APIs.

2.1.1.2 Error and status codes

In OPC it is common to return arrays of HRESULTs corresponding to arrays of data. In the case the data did not contain any errors an array with “no error occurred” codes still is returned. In DAIS such error codes will be returned as a sequence of error structs identifying the erroneous data and an error code. In case of no errors the sequence of error structs is empty.

2.1.1.3 Identifiers, handles, and blobs

OPC data accesses use both server and client side handles created based on identification texts for nodes and items. To make the translation from identification texts to handles fast and avoid repeated translation in OPC an intermediate server side identifier called the blob exists. In DAIS the server side handles and blobs are replaced by identifiers based on **ResourceID**s (both for nodes and items). For a description of **ResourceID**s refer to Section 2.2, “Data Access Facility (DAF),” on page2-4. The OPC client side handles are still kept in DAIS.

2.1.1.4 Callbacks

COM/OPC use the standard interfaces **IConnectionPoint** to set up callbacks. In DAIS callbacks are set up directly between server and client by updating an attribute holding the callback object.

2.1.1.5 Enumerators

The COM style enumerators in OPC are replaced by CORBA style iterators in DAIS.

2.1.1.6 *Parameters and structs*

In OPC multiple parallel vectors often pass data. In DAIS a single vector holding structs thus reducing the number of parameters replaces the parallel vectors.

2.1.1.7 *Method return data*

In OPC all methods return the HRESULT error value. In DAIS HRESULTs are replaced by exceptions and out parameters are returned as method results instead. Only in a few cases are there additional output parameters in DAIS.

2.1.1.8 *Items, structuring, and naming*

In OPC data is generally organized in hierarchical structures. In data access the leafs are called leaf nodes and the branches branch nodes or items. A leaf node or item is the same as an instance of property at an object. A branch node corresponds to an

- object having properties.
- arbitrary organization of other branch nodes.
- object having both properties and other branch nodes as children.

In DAIS data access the branch nodes are called nodes and the leaf nodes are called items. In DAIS alarms & events the branch nodes are called areas and the leaf nodes sources.

In OPC branch and leaf nodes has a name unique among the nodes or items that are children of the same node. A second name is formed by combining these names from each branch node in the path from the node or leaf to the root. Both names are called **ItemID**s in OPC and the name created by following the path to the root is sometimes called a fully qualified **ItemID**.

In DAIS the name unique among the child's to the same node is called label and the name including the labels from all nodes in the path to the root is called pathname. Both DAIS branch nodes and leaf nodes have both a label and a pathname.

A label is used in the same way as a name but the word label is preferred before the word name as it denotes something that is atomic; that is, it cannot be further divided.

2.1.1.9 *Server side cursors*

OPC provides a server side cursor from where browsing is made and the OPC interface provides methods to move the cursor.

DAIS does not provide server side cursors and requires the client to keep track of browse positions themselves. The reason for removing server side cursors is it makes clean up after crashed clients easier and simplifies the server design.

2.1.1.10 *Properties and types*

OPC does not support types meaning it is not possible in OPC to get information about the type of a node. OPC however supports properties which means it is possible to browse the existing properties and what properties a particular node has. DAIS has extended this to also include types; that is, it is possible to browse existing types, the properties each type has and each node has a type.

OPC defines the following property sets:

- set 1 of OPC specific properties
- set 2 of OPC recommended properties
- set 3 of vendor specific properties

In DAIS, OPC set 1 properties cannot be browsed through the property-browsing interface (Property). Instead access of set 1 properties is direct in the interfaces as parameters or struct members as is the case for OPC. This means that DAIS exposes only OPC set 2 and 3 properties through the property browse interface.

2.2 *Data Access Facility (DAF)*

The UMS DAF and DAIS are server APIs for access of object data rather than the data objects themselves.

2.2.1 *Resources and Properties*

The DAF describe a generic interface for navigating and reading data from complex data structures including relations between objects. Both DAF and DAIS support navigation in a space of hierarchically structured objects (an object is called a node in DAIS and a resource in DAF). Both support identification of objects and properties at an object. In DAIS an instance of property at an object is called an item and an **ItemID** (item identifier) identifies an item. An **ItemID** consists of a **ResourceID** (resource identifier) for the node and a **PropertyID** (property identifier) for the property. The DAIS API use **ItemID**s to access data while the DAF use **ResourceID**s and **PropertyID**s separated.

A system may implement both a DAF server and a DAIS server. In such a system it shall be assumed that the same object will have the same **ResourceID** seen through either API. This means it shall be possible to navigate to an object and retrieve its **ResourceID** through one of the APIs and use that **ResourceID** with the other API. In the same way **PropertyID**s are the same.

Textual identification of resources and properties in the DAF is by URIs (Uniform Resource Identifier). **ResourceID**s and **PropertyI**Ds have their own URIs. URIs can be translated into corresponding **ResourceID**s and **PropertyID**s and vice versa. A property name is a URI where the container part is a unique schema identifier and the fragment part is the property name.

For example:

http://www.epri.com/schema/CIM-07f.xml#Measurement.positiveFlowIn

DAIS does not support URIs directly but has a textual representation both for nodes, items, types, and properties. This textual representation may correspond to the fragment part of a URI.

2.2.2 Information model/schema

In the case where a system supports both the **DAF** and the **DAIS** interfaces it is expected that the same object will be identified by the same **ResourceID** through both interfaces and the same property by the same **PropertyID**. Sameness of objects and properties; however, depends on the information model exposed through the two interfaces and if the information models are different, then a mapping is required.

In the case where a system supports both the **DAF** and the **DAIS** interfaces it is expected that the same objects will be identified by the same **ResourceID** and URI fragment through both interfaces and the same property by the same **PropertyID** and URI fragment.

The exact mapping has to be described for each DAIS item – DAF property pair according to the underlying information models. Section 4.1.5, "Utility SCADA/EMS Measurement Model," on page 4-6 elaborates on the information model and a detailed mapping between IEC 61970-30x and the DAIS can be found in section 4.1.5.

The foregoing concepts are fundamental enough that they should find equivalents in any data repository. Some, perhaps approximate, equivalents are given in Table 2-1 as a guide.

Table 2-1 Mappings between modeling languages

RDF	**DAF**	**Relational Model**	**UML**	**DAIS**
Resource	Resource, `ResourceID`	Tuple (i.e. row)	Object	Node, `ResourceID`
Property	Property, `PropertyID`	Attribute (i.e. column) or foreign key	Attribute or association	Property, `PropertyID`
Class	Class, `ClassID`	Relation (i.e. table)	Class	Type, `TypeID`
Resource Description	ResourceDescription	Tuple value	-	Sequence <ItemState>
URI	URI, ResourceID	Key value	-	
Value	SimpleValue	Field value	-	SimpleValue
-	ResourceID and PropertyID pair	-	-	Item, `ItemID`

2.2.3 Data Types

DAIS uses the DAF SimpleValue type for data transferred over the API instead of the OPC type VARIANT. The data types used in OPC are Microsoft COM types and as DAIS is a CORBA API a mapping of the data types is needed. The translation of OPC/COM data types to DAF/CORBA data types are listed below.

Table 2-2 Mappings between OPC and DAF datatypes

OPC/COM basic types	**DAF Simple Value types**
-	ResourceID (RESOURCE_TYPE)
-	URI (URI_TYPE)
LPWSTR (VT_BSTR)	string (STRING_TYPE)
BOOL (VT_BOOL)	boolean (BOOLEAN_TYPE)
LONG (VT_R4)	long (INT_TYPE)
DWORD (VT_I4)	unsigned long (UNSIGNED_TYPE)
double (VT_R8)	double (DOUBLE_TYPE)
-	Complex (COMPLEX_TYPE)
FILETIME (VT_Date)	DateTime (DATE_TIME_TYPE)
-	ULongLong (ULONG_LONG_TYPE)
WORD (VT_I2)	unsigned long (UNSIGNED_TYPE)
FLOAT (VT_R4)	double (DOUBLE_TYPE)
BYTE (VT_UI1)	-
HRESULT (VT_ERROR)	-
VARIANT (VARTYPE)	SimpleValue (SimpleValueType)

2.3 COM and CORBA IDL Differences

Interfaces defined in COM and CORBA IDL differs in a number of ways. Important differences concerning interface definitions are

- object referencing,
- interface management,
- error management, and
- IDL.

A detailed description of the mapping can be found in the CORBA 2.3 specification, sections 17, 18, and 19.

2.3.1 Object Referencing

In CORBA each object is uniquely referenced in one step. In COM obtaining an object usually is a two step process. In the first step a state less server object is obtained. In the second step the server object is loaded with state data. In CORBA an object is a unique individual contains state from the very beginning.

2.3.2 Interface Management

A CORBA object has a single interface. This interface can be built from several other interfaces through inheritance. The resulting interface might have many methods and hence become big. A COM object usually has multiple interfaces and supports the client to detect and navigate between these interfaces at run time. As a CORBA interface is defined by inheritance it has to be fully defined at compile time. As COM allows run time detection of interfaces (the **IUnknown::QueryInterface()** method) a full match between interfaces implemented by a server and interfaces known to a client is not required.

Mapping OPC interfaces to DAIS interfaces can be done in two ways:

1. inherit a number of OPC interfaces into one CORBA object. This may result in name clashes that require renaming of methods.
2. instantiate an OPC interface as an own CORBA object referenced by a container object.

Both techniques are used in the DAIS specification.

2.3.3 Error Management

CORBA provides exceptions for error handling and COM does not. COM provides error status through a return parameter called HRESULT. The caller has to explicitly test the HRESULT to decide if the operation was successful. HRESULT return parameters are replaced by CORBA exceptions.

2.3.4 IDL

The COM and CORBA IDL have several syntactical differences and use different style guides. For details refer to the CORBA style guide [TBD].

2.4 IEC 1346-1, Structuring and Naming

DAIS (as well as OPC) structure nodes and items hierarchically. Nodes and items have a label uniquely identifying child’s located at the same node. The labels from the nodes in the path from a node or item to the root form a pathname. The pathname uniquely identifies a node or item in the tree.

The same principle for naming objects is described in the IEC 1346-1 standard. IEC 1346-1 call labels for single level designation and the pathname for multi level designation and supports multiple hierarchical structures. The DAIS supports multiple structures by allowing multiple views where each view exposes one structure.

2.5 IEC 61970 EMS API

The IEC 61970-30x [8] draft standard describes a specific organization of power system objects in a hierarchical structure. DAIS is transparent to the structure and hence supports the IEC 61970 structure.

DAIS is also transparent to what attributes are defined as long as they can be reached through the hierarchy. DAIS however defines a set of attributes that a DAIS server is expected to support. Some of these attributes can also be found in IEC 61970-30x. A detailed attribute mapping is provided later in this specification and is further elaborated in Section 4.1.5, “Utility SCADA/EMS Measurement Model,” on page 4-6.

DAIS Server 3

Contents

This chapter contains the following sections.

Section Title	**Page**
"Common Declarations"	3-1
"Server"	3-24

This section describes the DAISServer and IDL common to the server, data access, and alarms & events.

3.1 Common Declarations

3.1.1 Common IDL Overview

The DAIS relies on the DAF for basic data type declarations and some DAF declarations (for example, **ResourceID** and **SimpleValue**) are included because of this. DAIS common declarations (for example, **DAIS::Quality**) are made in **DAISCommon**. Interfaces for nodes, types, properties, and sessions are also common between data access and alarms & events. The IDL files and dependencies listed in Figure 3-1 are defined.

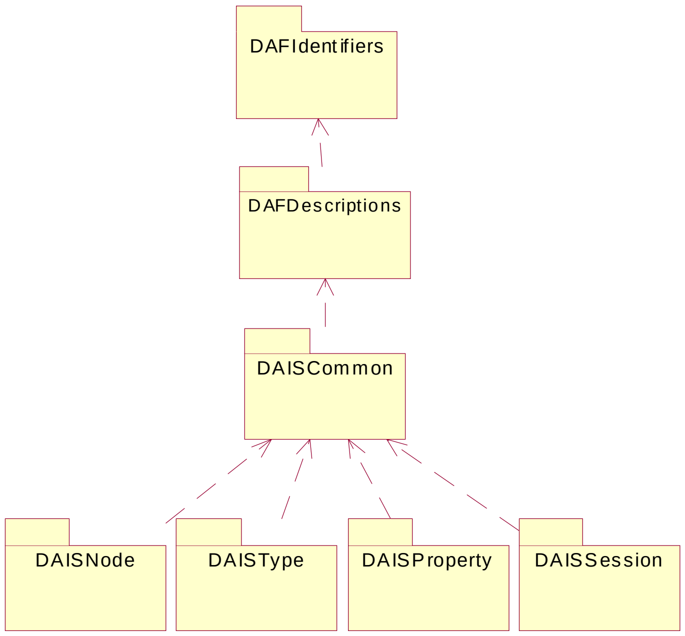


Figure 3-1 Dependencies between common IDL files

3.1.2 *DAFIdentifiers IDL*

Refer to the Data Access Facility specification (OMG document number: utility/99-11-02) .

3.1.3 *DAFDescriptions IDL*

Refer to the Data Access Facility specification (OMG document number: utility/99-11-02) .

3.1.4 *DAISCommon IDL*

3.1.4.1 *IDL*

```
module DAIS
{
typedef DAFDescriptions::ResourceID              ResourceID;
typedef DAFDescriptions::SimpleValueType         SimpleValueType;
typedef DAFDescriptions::SimpleValue             SimpleValue;
typedef DAFDescriptions::DateTime                DateTime;
typedef DAFDescriptions::PropertyID              PropertyID;
typedef DAFDescriptions::PropertyValueSequencePropertyValues;
typedef ResourceID          TypeID;

// sequences of resource ids
typedef sequence<ResourceID>   ResourceIDs;
typedef sequence<PropertyID>   PropertyIDs;
typedef sequence<TypeID>       TypeIDs;

// sequences of values
typedef sequence<string>              Strings;
typedef sequence<DateTime>            DateTimes;
typedef sequence<SimpleValueType> SimpleValueTypes;
typedef sequence<SimpleValue>         SimpleValues;

struct ItemID
{
    ResourceID                     resource;
    PropertyID                     property;
};
typedef sequence<ItemID>           ItemIDs;

typedef unsigned long              ClientHandle;

typedef unsigned short             Error;
struct ItemError
{
    Error                          err;
    ItemID                         item;
    string                         pathname;
};
typedef sequence<ItemError>        ItemErrors;

// error codes
const Error    ERROR_SUCCESS                = 0;
const Error    ERROR_BAD_RIGHTS             = 1;
const Error    ERROR_UNKNOWN_ITEMID         = 2;
const Error    ERROR_CLAMPED                = 3;
const Error    ERROR_OUT_OF_RANGE           = 4;
const Error    ERROR_UNKNOWN_PATHNAME       = 5;
const Error    ERROR_BAD_TYPE               = 6;
const Error    ERROR_UNKNOWN_ACCESS_PATH    = 7;

enum AccessRights
{
    READABLE,
    WRITEABLE,
    READ_AND_WRITABLE
};

typedef unsigned short         OPCQuality;

typedef unsigned short         ExtendedQuality;
```

```
struct Quality {
    OPCQuality                  opc_quality;
    ExtendedQuality             dais_ext_quality;
};

// Masks for extracting quality subfields
// (note 'status' mask also includes 'Quality' bits)

const   OPCQuality OPC_QUALITY_MASK         = 0xC0;
const   OPCQuality OPC_STATUS_MASK          = 0xFC;
const   OPCQuality OPC_LIMIT_MASK           = 0x03;

// Values for QUALITY_MASK bit field

const   OPCQualityOPC_QUALITY_BAD           = 0x00;
const   OPCQualityOPC_QUALITY_UNCERTAIN     = 0x40;
const   OPCQualityOPC_QUALITY_GOOD          = 0xC0;

// STATUS_MASK Values for Quality = BAD

const   OPCQualityOPC_QUALITY_CONFIG_ERROR      = 0x04;
const   OPCQualityOPC_QUALITY_NOT_CONNECTED     = 0x08;
const   OPCQualityOPC_QUALITY_DEVICE_FAILURE    = 0x0C;
const   OPCQualityOPC_QUALITY_SENSOR_FAILURE    = 0x10;
const   OPCQualityOPC_QUALITY_LAST_KNOWN        = 0x14;
const   OPCQualityOPC_QUALITY_COMM_FAILURE      = 0x18;
const   OPCQualityOPC_QUALITY_OUT_OF_SERVICE    = 0x1C;

// STATUS_MASK Values for Quality = UNCERTAIN

const   OPCQualityOPC_QUALITY_LAST_USABLE       = 0x44;
const   OPCQualityOPC_QUALITY_SENSOR_CAL        = 0x50;
const   OPCQualityOPC_QUALITY_EGU_EXCEEDED      = 0x54;
const   OPCQualityOPC_QUALITY_SUB_NORMAL        = 0x58;

// STATUS_MASK Values for Quality = GOOD

const   OPCQuality     OPC_QUALITY_LOCAL_OVERRIDE= 0xD8;

// Values for Limit Bitfield

const   OPCQuality     OPC_LIMIT_OK        = 0x00;
const   OPCQuality     OPC_LIMIT_LOW       = 0x01;
const   OPCQuality     OPC_LIMIT_HIGH      = 0x02;
const   OPCQuality     OPC_LIMIT_CONST     = 0x03;

// DAIS extended quality
// Masks

const   ExtendedQuality EXQ_REMOTE_SOURCE_MASK= 0x0003;
const   ExtendedQuality EXQ_LOCAL_SOURCE_MASK   = 0x000C;
const   ExtendedQuality EXQ_MASK                = 0x00F0;

// Values

const   ExtendedQuality EXQ_REMOTE_PROCESS      = 0x0000;
const   ExtendedQuality EXQ_REMOTE_SUBSTITUTED  = 0x0001;
const   ExtendedQuality EXQ_REMOTE_DEFAULTED    = 0x0002;

const   ExtendedQuality EXQ_LOCAL_NONE          = 0x0000;
const   ExtendedQuality EXQ_LOCAL_SUBSTITUTED   = 0x0004;
const   ExtendedQuality EXQ_LOCAL_SE_REPLACED   = 0x0008;

const   ExtendedQuality EXQ_OCILLATORY          = 0x0010;
```

};DAF declarations

These declarations (for example, **ResourceID**) import DAF declarations to DAIS.

TypeID

A **ResourceID** identifying type. A node has a type and the type defines what properties a node has.

ItemID

A pair of a **ResourceID** and a **PropertyID**. It uniquely identifies an item; that is, a property at a node.

Member	**Description**
resource	The ResourceID.
property	The PropertyID.

ClientHandle

A client created numeric handle used by the client to efficiently identify data coming from the server in callbacks.

Error

Numeric error codes that are returned by **ItemError**.

EnumValue	**Description**
ERROR_BAD_RIGHTS	The Item’s AccessRights do not allow the operation.

ERROR_UNKNOWN_ITEMID	The resource or property in the ItemID is unknown.
ERROR_CLAMPED	A value passed to WRITE was accepted but the output was clamped.
ERROR_OUT_OF_RANGE	The value was out of range.
ERROR_UNKNOWN_PATHNAME	The pathname was not recognized.
ERROR_BAD_TYPE	The server cannot convert the data between the specified format/ requested data type and the canonical data type.
ERROR_UNKNOWN_ACCESS_PATH	The item's access path is unknown.

ItemError

A struct for reporting of item related errors.

Member	Description
error	An error code as described below.
item	The **ItemID** identifying the item.
pathname	The pathname for display or presentation purposes.

AccessRights

Numeric access rights supported per item.

EnumValue	Description
READABLE	Read only data.
WRITEABLE	Write only data.
READ_AND_WRITABLE	Both read and write data.

OPCQuality

A flag word giving the OPC quality. Each flag has a specific meaning as described below. Three groups of flags exist

1. main quality telling if a value is good, bad, or suspected.
2. detailed quality.
3. limits telling if the value is stuck.

Bit masks are defined to extract these flags.

Quality, status and limit bit masks

Mask	Description
OPC_QUALITY_MASK	Bit mask for main quality.
OPC_STATUS_MASK	Bit mask for detailed quality.
OPC_LIMIT_MASK	Bit mask for the limits.

Main quality enumeration numbers

Enum	Description
OPC_QUALITY_BAD	The number for bad quality.
OPC_QUALITY_UNCERTAIN	The number for uncertain quality.
OPC_QUALITY_GOOD	The number for good quality.

After application of the **OPC_QUALITY_MASK** the quality shall be compared directly to the enumeration numbers to decide the quality.

Detailed quality flags for bad quality

Flag	Description
OPC_QUALITY_CONFIG_ERROR	There is a server configuration error concerning this value.
OPC_QUALITY_NOT_CONNECTED	The source of the value is not connected.
OPC_QUALITY_DEVICE_FAILURE	A device failure has been detected.
OPC_QUALITY_SENSOR_FAILURE	A sensor failure has been detected.
OPC_QUALITY_LAST_KNOWN	The updating has stopped but there is an old value available.
OPC_QUALITY_COMM_FAILURE	Communication has failed and no value available
OPC_QUALITY_OUT_OF_SERVICE	The updating of the value is manually blocked for update (the item is not active).

Detailed quality flags for uncertain quality

Flag	Description
OPC_QUALITY_LAST_USABLE	The value is old. The time stamp gives the age.

OPC_QUALITY_EGU_EXCEEDED	The value is beyond the predefined range.
OPC_QUALITY_EGU_EXCEEDED	The value is beyond the capability of representation (for example, counter overflow).
OPC_QUALITY_SENSOR_CAL	The sensor calibration is bad.
OPC_QUALITY_SUB_NORMAL	Value is derived from multiple sources where the majority has less than required good quality.

Detailed quality flags for good quality

Flag	Description
OPC_QUALITY_LOCAL_OVERRIDE	The value is substituted (usually manually).

Definition of limit flags

Flag	Description
OPC_LIMIT_OK	The value is not limited; that is, it moves freely up or down.
OPC_LIMIT_LOW	The value is stuck at a low limit.
OPC_LIMIT_HIGH	The value is stuck at a high limit.
OPC_LIMIT_CONST	The value is stuck constant.

ExtendedQuality

A flag word giving the DAIS extended quality. Each flag has a specific meaning as described below. These quality definitions are based on the revised IEC 61850-7-3 definitions of quality.

The following masks are defined for the **ExtendedQuality**.

Mask	Description
EXQ_REMOTE_SOURCE_MASK	Bit mask for the remote source.
EXQ_LOCAL_SOURCE_MASK	Bit mask for the local source.
EXQ_MASK	Bit mask for the extended quality.

Flags defining remote source

Flag	Description
EXQ_REMOTE_PROCESS	The remote source for this value is the process.
EXQ_REMOTE_SUBSTITUTED	The value is remotely substituted (manually).
EXQ_REMOTE_DEFAULTED	The value is remotely initialized by a default value.

Flags defining the local source

Flag	Description
EXQ_LOCAL_NONE	The value is not updated locally; that is, it has a remote source and the source is given according to the above remote source specification.
EXQ_LOCAL_SUBSTITUTED	The value is locally substituted (manually).
EXQ_LOCAL_SE_REPLACED	The value is locally substituted by State Estimator.

Extended quality for OPC uncertain

Flag	Description
EXQ_OCILLATORY	Measurement value is due to a transient or oscillatory condition.

Quality

The DAIS quality consists of **OPCQuality** and **ExtendedQuality**.

Member	Description
opc_quality	The quality as specified by OPC.
dais_ext_quality	The DAIS quality is intended for extensions independent of OPC.

3.1.5 Iterator Methods IDL

Methods that return information about more than one resource may 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.

3.1.6 DAISNode IDL

3.1.6.1 DAIS::Node overview

A node may represent a real world object such as a location or a piece of equipment. A node may also represent a schema item such as a type or property. Nodes correspond to “branches” in the **IOPCBrowseServerAddressSpace** or **IOPCEventAreaBrowser** interfaces. Nodes correspond to Resources in the RDF model and the DAF interface. Each node has a universal identity given by its **ResourceID**. The **ResourceID** of a node is the same in all views provided by a DAIS server. DAIS servers may be coordinated with DAF servers so that a node has the same **ResourceID** as the corresponding resource. Each node has zero or more child’s. A child may be another node or any other type of object (for example, data access items or alarms & event sources). Each node has a type. The type determines what other types of child objects a node has.

Nodes are arranged in a strict hierarchy for naming purposes. A DAIS server may provide more than one such hierarchy, each is called a view. (The view is selected when the session is initiated.) Within a view, each node, except for the root node, has a single parent node, a label that is unique among all nodes with the same parent, and a unique pathname. A node’s pathname is a string that contains its label and the pathname of its parent. The pathname must be a valid URI, but apart from that the syntax of pathnames is implementation dependent.

The Node IDL defines a main interface **DAIS::Node::Home** for browsing among the hierarchically structured nodes. Nodes are described by **DAIS::Node::Description** struct.

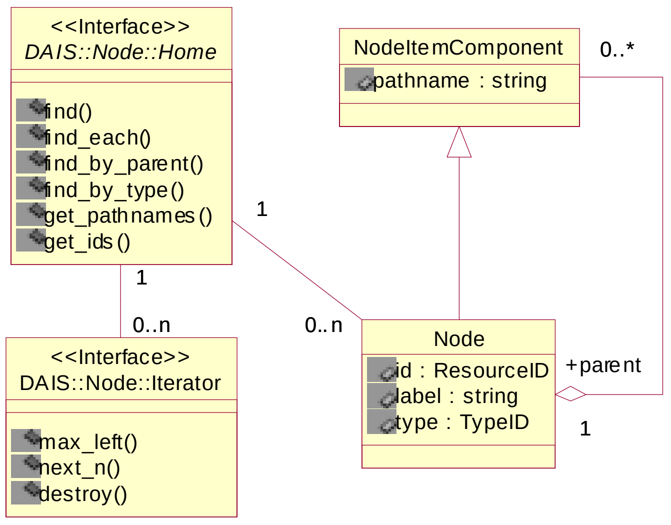


Figure 3-2 DAIS node IDL in UML

3.1.6.2 IDL

```
module DAIS {
module Node {

struct Description
{
	ResourceID		id;
	ResourceID		parent;
	string			label;
	TypeID			type;
};
typedef sequence< Description > Descriptions;

interface Iterator
{
	boolean next_n (
		in	unsigned long	n,
		out	Descriptions	nodes
	);
	void destroy();
};

abstract interface Home
{
	exception UnknownResourceID {string reason;};

	Description find (
		in	ResourceID		node
	) raises (UnknownResourceID);

	Descriptions find_each (
		in	ResourceIDs		nodes
	) raises (UnknownResourceID);

	Iterator find_by_parent (
		in	ResourceID		node
	) raises (UnknownResourceID);

	Iterator find_by_type (
		in	ResourceID			node,
		in	TypeIDs				type_filter
	) raises (UnknownResourceID);

	Strings get_pathnames (
		in	ResourceIDs		nodes
	);

	ResourceIDs get_ids (
	in	Strings			pathnames
```

```
      );
};};};
```

Description

A struct describing a node.

Member	Description
id	The identification of this node.
parent	The identification of the parent node.
label	The label (single level designation) of the node.
type	A reference to the type of the node.

Iterator

Refer to Section 3.1.5, “Iterator Methods IDL,” on page3-10. This interface corresponds to the OPC interface EnumString with the difference that the Iterator returns the Description struct instead of a string.

Home

An interface used for browsing nodes. The interface corresponds to the **IOPCBrowseServerAddressSpace** with the **BrowseFilterType** set to **OPC_BRANCH** or the **IOPCEventAreaBrowser**. A major difference to OPC is that the server does not provide a cursor for clients. Instead clients have to provide the browse position in each call.

UnknownResourceID

An exception telling that the **ResourceID** is unknown. For methods taking a sequence of resource ids the first found unknown id is reported. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

find()

For a given node return information about that node.

Parameter	Description
node	A node identification.
return	The node description.

find_each ()

For a sequence of nodes return information about each node.

Parameter	Description
nodes	A sequence of node identifications.
return	An iterator holding the node descriptions.

find_by_parent ()

For a given node return all child nodes.

Parameter	Description
node	The parent node identification.
return	An iterator holding the child node descriptions.

find_by_type()

For a sub-tree given by node return all child nodes of the specified type.

Parameter	Description
node	The identification of the node defining the sub-tree
type_filter	A list of **TypeID**s. Nodes matching any of the **TypeID**s will be held by the returned iterator.
return	An iterator holding descriptions for the found nodes.

get_pathnames()

Translate a sequence of node identifications to the corresponding sequence of pathnames. If a node fails to translate to a pathname (due to an unknown node identification) the corresponding pathname is an empty string.

Parameter	Description
nodes	The sequence of nodes.
return	The corresponding sequence of pathnames.

get_ids()

Translate a sequence of pathnames to the corresponding sequence of node identifications. If a pathname fails to translate to a node identification (due to an unrecognized pathname) the corresponding node identification is NULL.

Parameter	Description
pathnames	The sequence of pathnames.
return	The corresponding sequence of node identifications.

3.1.7 DAISType IDL

3.1.7.1 DAIS::Type Overview

A type represents a set of related properties and associations. Each node has a type and all the properties represented by that type apply to the node. Each type is identified by a **ResourceID** and has a label and description. A type may be obtained for any node using the node's **TypeID**. Related types may be grouped into a schema. A **ResourceID** identifies each schema. All the types in a schema may be obtained given the schema **ResourceID**. A schema and its type may be represented as nodes in one or more of views provided by a DAIS server. When a schema is represented as a node, the node's **ResourceID** and the schema **ResourceID** are identical. Similarly, when a type is represented as a node , the node's **ResourceID** and the type's **ResourceID** are identical. The type's parent is always the node representing its schema, the type's label is identical to the node label and the type's description is identical to the node description.

The type IDL defines a main interface **DAIS::Type::Home** for browsing supported types.

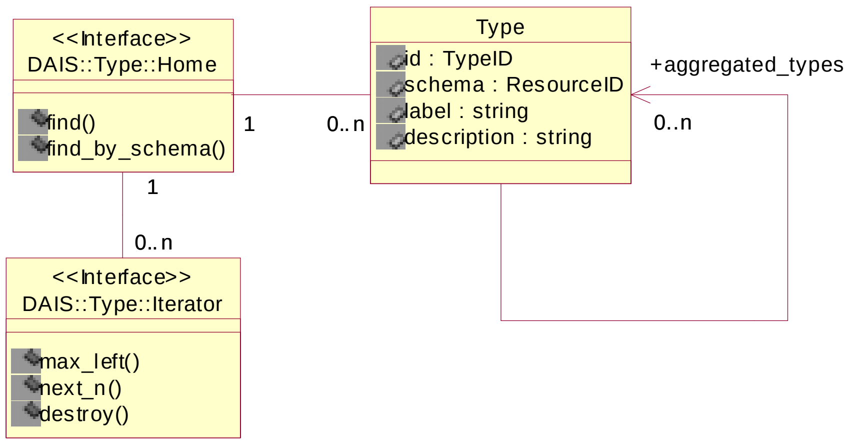


Figure 3-3 DAIS type IDL in UML

3.1.7.2 IDL

```
module DAIS {
module Type {

struct Description {
      TypeID                              id;
      ResourceID                          schema;
      string                              label;
      string                              descrip;
      TypeIDs                             aggregated_types;
};
typedef sequence<Description>Descriptions;

interface Iterator
{
      boolean next_n (
            in   unsigned long       n,
            out Descriptions         types
      );
      void destroy();
};

interface Home
{
      exception UnknownResourceID {string reason;};

      Description find (
            in   TypeID                       type
```

```
    ) raises (UnknownResourceID);

    Iterator find_by_schema (
        in   ResourceID              node
    );
};};};
```

Description

A struct describing a type.

Member	Description
id	The identification of this type.
schema	The identification of the schema where the type is defined.
label	The label of the type.
descrip	A description of the type.
aggregated_types	A sequence of type identifications that a node of this type may contain. This information is intended as a guide when the type filter is specified for the **find_by_type()** methods.

Home

An object used to browse the types. There is no corresponding interface in OPC.

UnknownResourceID

An exception telling that the **ResourceID** is unknown. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

find()

For a given type return information about that type.

Parameter	Description
type	A type identification.
return	The type description.

find_by_schema()

For a given schema find all types defined by that schema.

Parameter	Description
node	The identification of the schema.
return	A sequence of type descriptions.

3.1.8 DAISProperty IDL

3.1.8.1 DAIS::Property overview

A property represents a characteristic of a node that can be described with a value. A given property may apply to many nodes, for each such node there will be an item corresponding to the property. (See Section 4.2.4, "DAISItem IDL," on page 4-14).

A DAIS property corresponds to a property in RDF and the DAF. A DAIS property corresponds to the concept of a property in OPC. However, the six core OPC properties (timestamp, quality, value) do not correspond to properties in DAIS. They are given special treatment in OPC (they are not the same as other OPC properties). See Section 4.2.4, "DAISItem IDL," on page 4-14 for interfaces to handle these.

Each property is identified by a **ResourceID** and has a label, description, and canonical data type. The canonical data type is a member of the **SimpleType** enumeration and indicates the preferred CORBA atomic data type for values of this property. Every item of a given property has an identical canonical data type. A property may be obtained for any item via the property member of the **ItemID**. All properties that apply to a given node may be obtained, given the node's **ResourceID**. All properties that apply to the nodes of a given type may be obtained, given the **TypeID**. A property may be represented as a node in one or more of views provided by a DAIS server. In this case the node's **ResourceID** and the property's **ResourceID** are identical. The property's parent is always the node representing the type to which it belongs. The property's label is identical to the node label and the property's description is identical to the node description.

The property IDL defines a main interface **DAIS::Property::Home** for browsing properties. The information model describing how the properties are related to nodes and items is found in Section 4.1.1, "Nodes, Items, Types, and Properties," on page 4-2.

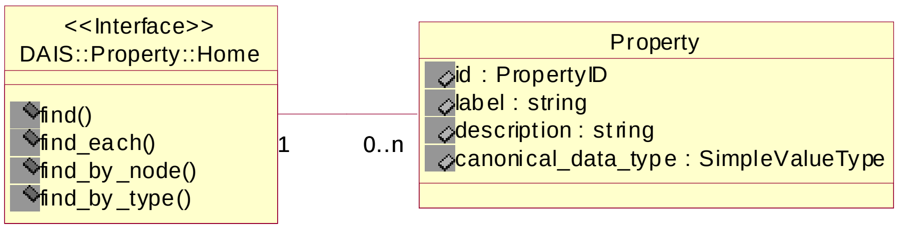


Figure 3-4 DAIS property IDL in UML

3.1.8.2 IDL

```
module DAIS {
module Property {

struct Description {
    PropertyID                  id;
    string                      label;
    string                      descrip;
    SimpleValueType             canonical_data_type;
};
typedef sequence<Description>Descriptions;

interface Home
{
    exception UnknownResourceID {string reason;};

    Description find (
        in   PropertyID         property
    ) raises (UnknownResourceID);

    Descriptions find_each (
        in   PropertyIDs        properties
    ) raises (UnknownResourceID);

    Descriptions find_by_node (
        in   ResourceID         node
    ) raises (UnknownResourceID);

    Descriptions find_by_type (
        in   TypeID                 type
    ) raises (UnknownResourceID);
};};};
```

Description

Describe a property.

Member	Description
id	The identification of this property.
label	The label (single level designation) of the property.
descrip	A description of the property.
canonical_data_type	The data type used for the property in the server.

Home

An object used for browsing properties defined for a type or existing at a node. It corresponds to the OPC interface **IOPCItemProperties**.

UnknownResourceID

An exception telling that the **ResourceID** is unknown. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

find()

For a given property return information about that property.

Parameter	Description
property	A property identification.
return	The property description.

find_each()

For a given property return information about that property.

Parameter	Description
properties	A sequence of property identifications.
return	The sequence of property descriptions.

find_by_node ()

For a node return information about each property describing its items. This method corresponds to **IOPCItemProperties::QueryAvailableProperties()**.

Parameter	Description
node	A node identification.
return	A sequence of property descriptions.

find_by_type ()

For a given type return all property descriptions.

Parameter	Description
type	A type identification.
return	A sequence of property descriptions.

3.1.9 DAISSession IDL

3.1.9.1 DAIS::Session overview

The session is an abstract interface inherited by data access and alarms & events sessions. A session represents a single conversation with the DAIS service. A session has a connection to a shut down callback used by a server to shut down clients in the case of an ordered shut down of the server. If the session object is destroyed or a failure is detected by the server when invoking an operation on any callback objects, then the session is terminated.

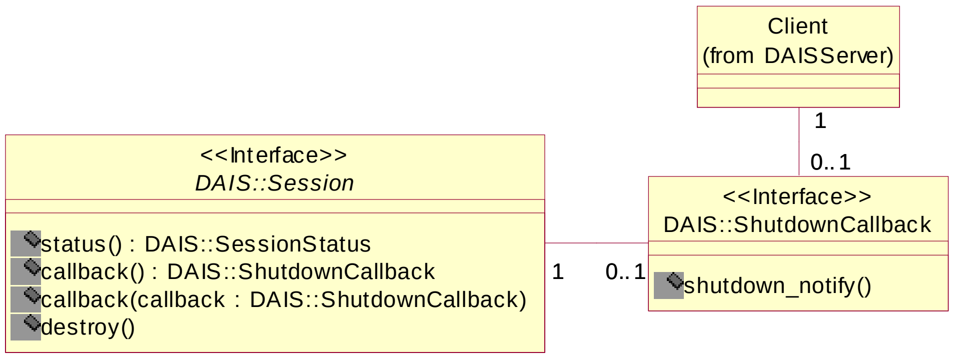


Figure 3-5 DAIS session IDL in UML

The **status()** method corresponds to the read only **SessionStatus** attribute in the IDL. The **callback()** get and set methods correspond to the **ShutDownCallback** attribute.

3.1.9.2 IDL

```
module DAIS {

struct SessionStatus {
	string				name;
	DateTime			start_time;
	DateTime			current_time;
	DateTime			last_update_time;
	unsigned long		group_count;
	long				band_width;
};

interface ShutdownCallback
{

	void shutdown_notify (
		in string			reason
	);
};

abstract interface Session
{

	readonly attribute SessionStatusstatus;

	attribute ShutdownCallbackcallback;

	void destroy();
};};
```

SessionStatus

A struct holding session status.

Parameter	Description
name	Within the server unique name of the session.
start_time	The time when the session was started. This time is not reset during the session lifetime.
current_time	The current time as known by the server.
last_update_time	The time when the server sent an event notification for this session.

group_count	The current number of groups for a data access session or the number of event subscriptions for a alarms & event session.
band_width	If held updated by the server the percentage bandwidth in use for communication with underlying RTUs or devices. A value of 100 or more indicate that more bandwidth for communication with devices is required than available. A value of -1 indicate the value is unknown by the server.

ShutdownCallback

An object implemented by clients and used by the server to indicate that it will shutdown soon. No further calls should be made and no further data callbacks should be expected.

shutdown_notify()

Parameter	Description
reason	The reason is an explanation of why the server is shutting down.

Session

An interface representing a single conversation with the DAIS service. The interface abstract and is inherited into interfaces representing sessions supporting specific services, (for example, data access or alarms & events).

status

A read only attributes holding the **SessionStatus**.

callback

An attribute holding a reference to a **ShutDownCallback** object. A client that want to receive shut down callbacks from a server shall update the attribute with a reference to a **ShutDownCallback** object.

destroy()

A method for deletion of the session object.

3.2 Server

3.2.1 DAISServer IDL Overview

The DAISServer IDL describe the DAIS server interface and depend on the types of session interfaces (for example, data access or alarms & events) it implement. A server not implementing a session type (data access or alarms & events) shall still be capable of reporting the type as not implemented. The IDL files and dependencies in Figure 3-6 are defined below.

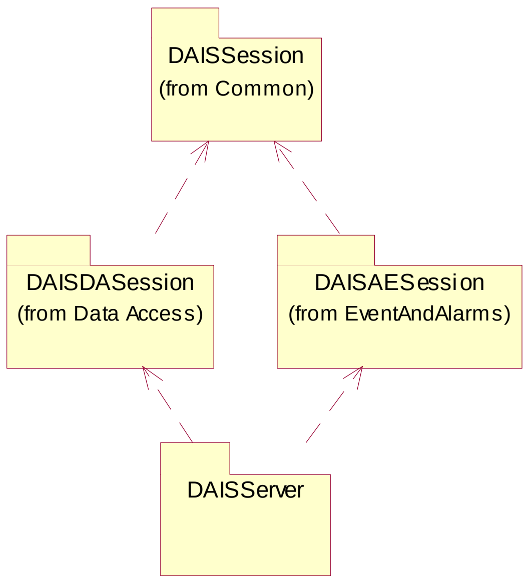


Figure 3-6 Dependencies between server IDL files

3.2.2 DAIS Server IDL

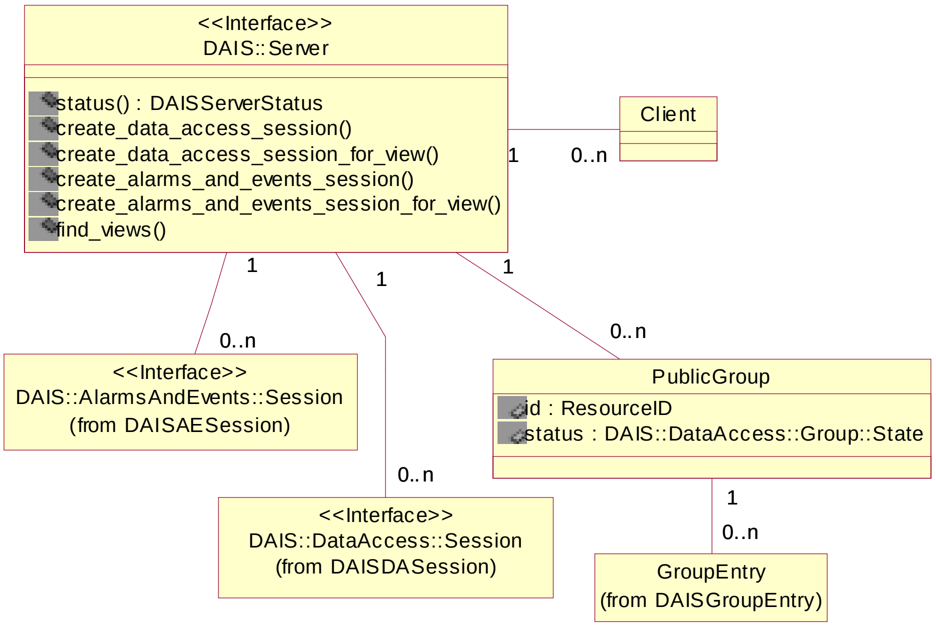
3.2.2.1 DAIS::Server objects overview

The fundamental DAIS service from which session objects may be obtained. **DAIS::Server** would normally be implemented as a persistent object accessed via the naming service or the trader service. From the **DAIS::Server** object the session objects for data access or alarms & events are created. A client may create as many session objects as wanted.

A session can be created for a view. A view corresponds to a specific hierarchical organization of objects (also called nodes in this specification). The same object may appear in multiple views and hence in different hierarchical structures. An example is a breaker appearing in a functional structure (having the function of breaking current) and in a location structure (the place where it is located), refer also to [10]. Another example is the same object appearing in different areas of authority.

A DAIS server supporting data access may have a number of persistent public groups. Public groups can be used and managed through the group interface, refer to Section 4.2.7, "DAISGroup IDL," on page 4-37.

Services that are not implemented will raise the CORBA standard exception **NO_IMPLEMENT**. Not implemented exceptions can be expected for a data access session, an alarms & event session, and inspection.



<<Interface>>
DAIS::Server
status() : DAISServerStatus
create_data_access_session()
create_data_access_session_for_view()
create_alarms_and_events_session()
create_alarms_and_events_session_for_view()
find_views()
Client
1
0..n
1
1
1
0..n
0..n
<<Interface>>
DAIS::AlarmsAndEvents::Session
(from DAISAESession)
PublicGroup
id : ResourceID
status : DAIS::DataAccess::Group::State
0..n
<<Interface>>
DAIS::DataAccess::Session
(from DAISDASession)
1
0..n
GroupEntry
(from DAISGroupEntry)

Figure 3-7 DAIS server IDL in UML

3.2.2.2 IDL

```
module DAIS {

enum ServerState {
      SERVER_STATE_RUNNING,
      SERVER_STATE_FAILED,
      SERVER_STATE_NOCONFIG,
      SERVER_STATE_SUSPENDED,
      SERVER_STATE_TEST
};

struct ServerStatus {
      DateTime                        start_time;
      DateTime                        current_time;
      ServerState                     server_state;
      unsigned long                   session_count;
      unsigned long                   major_version;
      unsigned long                   minor_version;
      unsigned long                   build_number;
      string                          vendor_info;
};

abstract interface Inspection {};

interface Server
{
      exception DuplicateName {string reason;};
      exception InvalidView {string reason;};

      readonly attribute ServerStatusstatus;

      DataAccess::Session create_data_access_session(
          in string                   session_name
      ) raises (DuplicateName);

      DataAccess::Session create_data_access_session_for_view(
          in string                   session_name,
          in string                   view_name
      ) raises (DuplicateName, InvalidView);

      AlarmsAndEvents::Session create_alarms_and_events_session(
          in string                   session_name
      ) raises (DuplicateName);

      AlarmsAndEvents::Session create_alarms_and_events_session_for_view(
          in string                   session_name,
          in string                   view_name
      ) raises (DuplicateName, InvalidView);

      Strings find_views();

      Inspection inspect();
};};
```

ServerState

An enumeration of the states a Server may have.

EnumValue	Description
SERVER_STATE_RUNNING	The server is running normally. This is the usual state for a server

SERVER_STATE_FAILED	A vendor specific fatal error has occurred within the server. The server is no longer functioning. The recovery procedure from this situation is vendor specific.
SERVER_STATE_NOCONFIG	The server is running but has no configuration information loaded and thus cannot function normally. Note this state implies that the server needs configuration information in order to function. Servers which do not require configuration information should not return this state.
SERVER_STATE_SUSPENDED	The server has been temporarily suspended via some vendor specific method and is not getting or sending data. Note that Quality will be returned as OPC_QUALITY_OUT_OF_SERVICE.
SERVER_STATE_TEST	The server is in Test Mode. The outputs are disconnected from the real hardware but the server will otherwise behave normally. Inputs may be real or may be simulated depending on the vendor implementation. Quality will generally be returned normally.

ServerStatus

Member	Description
start_time	Time the server was started. This is constant for the server instance and is not reset when the server changes states. Each instance of a server should keep the time when the process started.
current_time	The current time as known by the server.
server_state	The current status of the server. Refer to ServerState enumeration.
session_count	The total number of sessions created by clients for this server.
major_version	The major version of the server software.
minor_version	The minor version of the server software.
build_number	The ‘build number’ of the server software.

vendor_info	Vendor specific string providing additional information about the server. It is recommended that this mention the name of the company and the type of device(s) supported.

Inspection

An optional interface intended to be specialized into a vendor specific inspection object. The inspection object is intended to expose server internal details for debugging and inspection purposes.

Server

An object implementing the DAIS server. A DAIS server might provide more than one view on nodes. A view is a specific hierarchical organization of nodes and nodes may appear in more than one hierarchical structure (for example, a functional structure or a location structure as defined by IEC 1346-1), refer also to Section 2.4, “IEC 1346-1, Structuring and Naming,” on page 2-7.

DuplicateName

An exception raised when an object is created and the name already exists. No object is created if the exception is raised. Is used for session and group manager objects.

InvalidView

An exception raised when an invalid view is specified. An invalid view is if the view name does not exist or a view not intended for the type of session is used (for example, a view for data access is used for alarms & events).

status

An attribute holding the ServerStatus.

create_data_access_session()

A method for creation of a data access session object. The default view will be used.

Parameter	Description
name	The name of the session. If an empty name is supplied the server will create a name for the session. If a duplicate name is supplied no session is generated.
return	A reference to the created **DAIS::DataAccess::Session** object.

create_data_access_session_for_view()

A method for creation of a data access session object using a view of nodes.

Parameter	Description
name	The name of the session. If an empty name is supplied, the server will create a name for the session. If a duplicate name is supplied, no session is generated.
view_name	The name of the view to open. If no name is supplied the default view will be used.
return	A reference to the created **DAIS::DataAccess::Session** object.

create_alarms_and_events_session()

A method for creation of an alarms & events session object. The default view will be used.

Parameter	Description
name	The name of the session. If an empty name is supplied, the server will create a name for the session. If a duplicate name is supplied, no session is generated.
return	A reference to the created **DAIS::DataAccess::Session** object.

create_alarms_and_events_session_for_view()

A method for creation of an alarms & events session object using a view of areas. This allows a server to support different area structures for different purposes (for example, operational responsibility, workorder management).

Parameter	Description
name	The name of the session. If an empty name is supplied the server will create a name for the session. If a duplicate name is supplied no session is generated.
view_name	The name of the view to open.
return	A reference to the created **DAIS::DataAccess::Session** object.

find_views()

A method to get the names for the server supported views.

Parameter	Description
return	A sequence of view names.

inspect()

A method for creation of an inspection object.

Parameter	Description
return	The inspection object.

Session Management

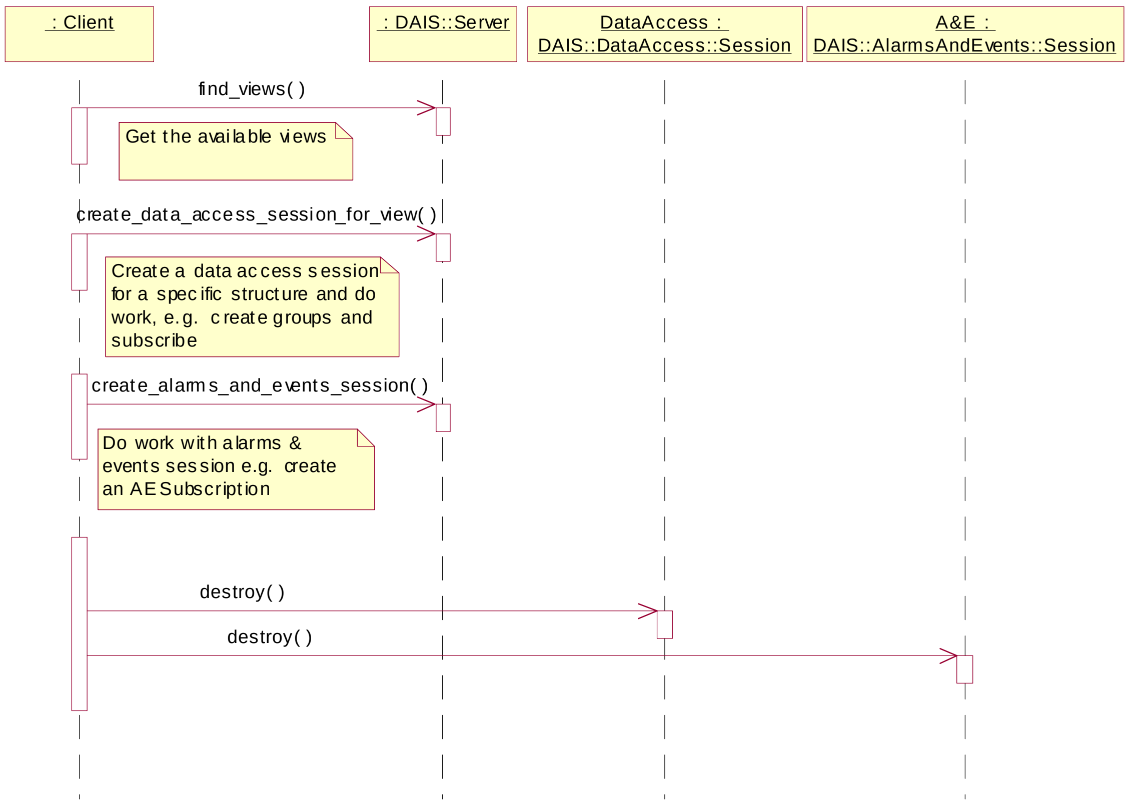

: Client
: DAIS::Server
DataAccess : DAIS::DataAccess::Session
A&E : DAIS::AlarmsAndEvents::Session
find_views( )
Get the available views
create_data_access_session_for_view( )
Create a data access session for a specific structure and do work, e.g. create groups and subscribe
create_alarms_and_events_session( )
Do work with alarms & events session e.g. create an AE Subscription
destroy( )
destroy( )

Figure 3-8 Session management interaction

DAIS Data Access 4

Contents

This chapter contains the following sections.

Section Title	**Page**
"Information Model"	4-1
"API"	4-8

The data access interface provides a client with a way to read, write, or subscribe for data (items) held by the server. It has a discovery interface where a client can browse and select available data. Selected data is used to compose groups and groups are used in read, write, or subscribe operations. To get subscribed data, a client has to connect a callback object to the server so that the server can notify the clients with changed data.

4.1 Information Model

Equipment in any industrial process is usually modeled as entities or classes in computerized systems. The class name usually reflects the type of equipment. Objects from such classes are in this document called Real World Objects (RWOs). RWO classes have properties describing equipment characteristics (for example, size, length, geometries, material, nameplate data) as well as application specific data (for example, impedance parameters, state estimated values). Depending on the purpose of a system and the managed (industrial) process the RWOs are of different types and contain different properties. Some RWOs are common between systems and independent of industrial process like process variables; that is, measurement (state) and control variables. Other RWOs like transmission lines and breakers are specific to power transmission.

RWO will typically have properties depending on the RWO type (for example, measurement, transformer) and what application or usage the RWO is involved in. Properties are defined per type of RWO.

4.1.1 Nodes, Items, Types, and Properties

In DAIS RWOs are represented by Nodes. Property instances at a node are represented by Items. An item may represent a measurement value, a control output value, or any parameter (for example, a limit value, a unit, a name, a description). A Type and properties belonging to a type represent an RWO type by **DAISProperties**. Nodes are hierarchically structured and the leafs are Items. This is shown in Figure 4-1.

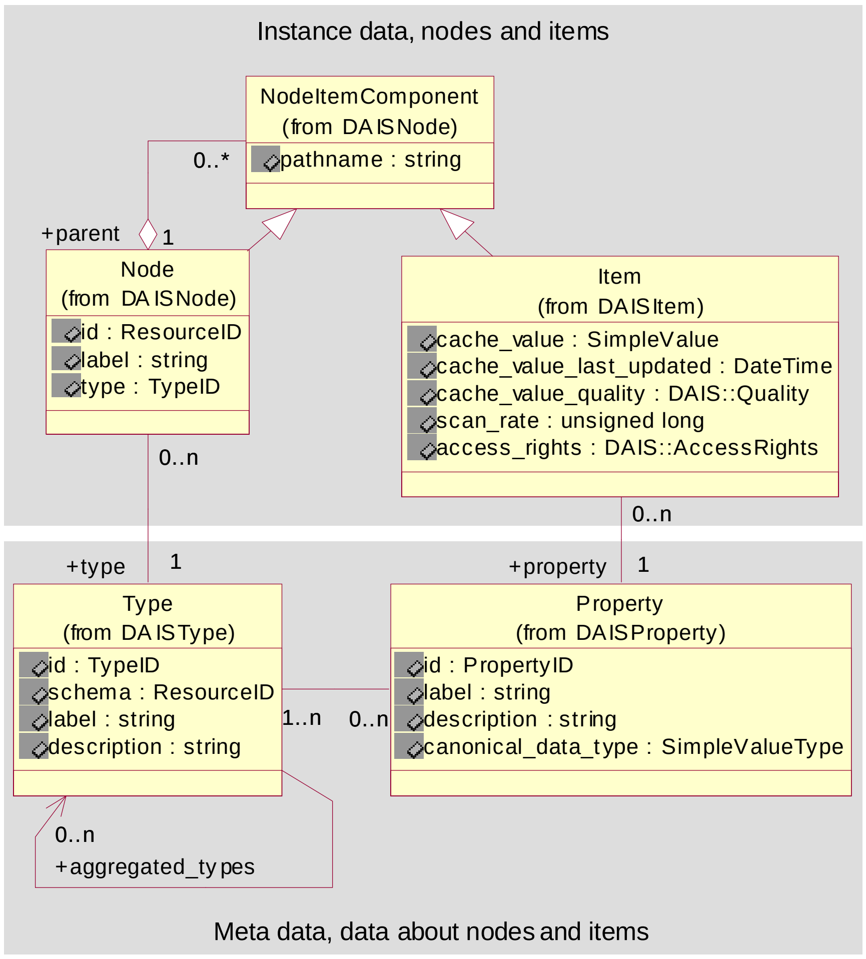


Figure 4-1 DAIS data access server information model

The Component class inherited by the Node and Item classes models the hierarchical structure. The Node may contain any number of Components as described by the **Contains** role having the cardinality many (role is a UML concept, refer to [9] for an explanation). A Component is a member of one Node as described by the **MemberOf** role having the cardinality of 1. A Component can be both a Node and an Item through the inheritance, which means a Node can contain other Nodes or Items. An Item cannot contain any Components as it only inherits the **MemberOf** role and not the **Contains** role.

4.1.2 Naming

Node and item names follow OPC [4] and IEC 1346-1 [10]. Each Node has a label unique among other Nodes having the same parent; that is, are **MemberOf** the same Node. An Item does not have an own label but uses the label from the Property. Each Item in a Node is associated with different **DAISProperties** so that the label is unique among other items at the same node. The labels in the path from an item or node to the root form a pathname. Labels and pathnames are explained in Figure 4-2.

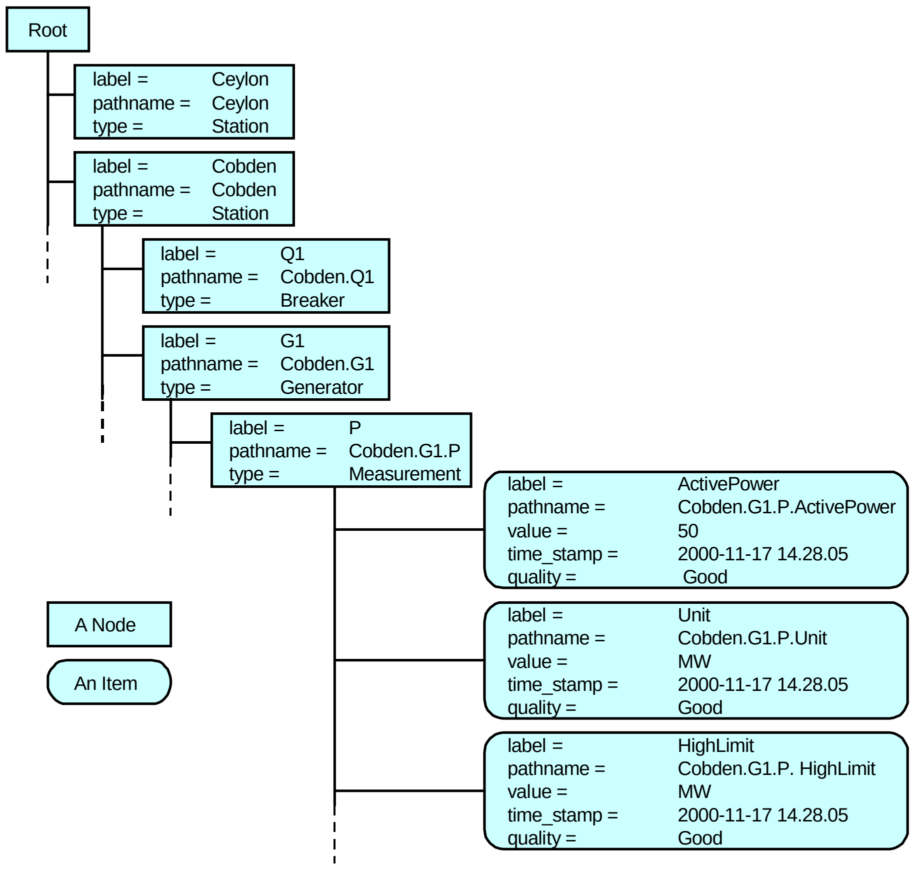


Figure 4-2 Labels and pathnames

A delimiter, the label delimiter, might separate the labels in a pathname. Assuming the label delimiter is a "." an example of an item pathname for a measurement value in Figure 4-2 is

Cobden.G1.P.Value

where the labels are:

- station; Cobden
- generator; G1
- active power measurement; P
- the actual measurement value property; Value

Exactly how the pathname is composed from the labels is server specific and outside the scope of this specification.

4.1.3 Item Values

Items are associated with values. Typically an item value provided by a DAIS server is read from a device and transferred to one or more clients. In a distributed control system involving remote devices (as indicated in Figure 4-3) communication failures might make item values not available. To cope with communication failures item values are associated with a quality. The quality indicates the reliability of the item value. Devices usually scan item values at a certain rate and item values will be transferred to the DAIS server at this rate or some other. In the server item values will appear as time stamped and quality coded samples. A server that keeps item values in a local cache is expected to always hold the latest sample. Other item related informations are access-rights and scan-rate. This information is shown in Figure 4-4. The **cache_value** is the latest sample received from a device, the **cache_value_last_updated** the time when the **cache_value** was last updated or validated, and **cache_value_quality** when the value was last updated or validated.

For items, a DAIS server exposes the following information to clients:

- the value and its data type,
- the quality of the value,
- the time stamp for the value,
- the fastest scan rate with which the value can be expected to be updated, and
- the access rights.

To make access of item values efficient and avoid reading the values from devices each time a client requests item values, a DAIS server is expected to have a local cache. The mechanism for keeping the cache up to date is server specific but a client shall expect the following from the server:

- Values delivered from the cache reflect the latest value considering update rate and update dead bands. Based on the agreed update rate (between a client and a server) a client can expect that the server will validate the values with devices with the agreed update rate.

- The dead band is expected to be checked at each update or validation. Values that don't transgress the deadband will not be reported.
- Time stamps delivered from the cache shall reflect when the values were updated or validated with the devices according to the agreed update rate. The time stamp gives the time for the latest successful update or validation.
- The quality shall reflect how successful the server has been in keeping the values updated or validated.

4.1.4 OPC Recommended Properties

DAIS is flexible in terms of what types and properties may exist in a server.

In OPC there is a recommendation of what properties are expected to be supported by a data access server. These recommended properties are shown in Table 4-1. The column names correspond to the attributes in the Property class seen in Figur e4-4.

Table 4-1

label	**id**	**canonical_data_type**	**description**
engineeringUnit	100	STRING_TYPE	Engineering unit.
description	101	STRING_TYPE	Description.
maxValue	102	DOUBLE_TYPE	Maximum value.
minValue	103	DOUBLE_TYPE	Minimum value.
sensorMaximum	104	DOUBLE_TYPE	Maximum value from an analog input sensor.
sensorMinimum	105	DOUBLE_TYPE	Minimum value from an analog input sensor.
closedLabel	106	STRING_TYPE	Text for the closed state for a discrete status input.
openLabel	107	STRING_TYPE	Text for the open state for a discrete status input.
itemTimeZone	108	UNSIGNED_TYPE	
	109-4999		Reserved by OPC

For a system having data described by a particular schema (for example, the CIM for power systems) the implementation of that schema as seen through DAIS has to be decided. Either a mapping to the recommended OPC properties can be made or the schema can be exposed as is through DAIS. This is however outside the scope of this specification.

4.1.5 Utility SCADA/EMS Measurement Model

The classes and properties defined in the IEC 61970-30x (the CIM) can be mapped onto the OPC recommended properties. As the CIM is highly structured this will create a complicated mapping. An alternative is to expose CIM as is through the DAIS. This means that the CIM classes and their properties will appear the same as seen through the DAIS as through the DAF interfaces. As the DAIS requires a hierarchical structuring of nodes the CIM equipment hierarchy is selected as the hierarchy exposed through the DAIS.

Besides the equipment hierarchy the CIM also has a number of other associations. Such associations may be made visible through the DAIS interface as properties holding **ResourceID**s.

The CIM has several associations between RWOs and ways to structure RWOs. The requirement from DAIS on the equipment hierarchy is that it is strictly hierarchical, else DAIS is transparent to any information model.

For RWOs in the equipment hierarchy its properties simply appear as items at the nodes representing the RWOs. The DAIS supports navigating across associations as **ResourceID**s can be conveyed by **SimpleValues**.

Utility SCADA/EMS systems have a number of different applications calculating alternate measurement values. In the IEC61970 draft standard this is modeled by Measurements containing one or more MeasurementValues as shown in Figure 4-5.

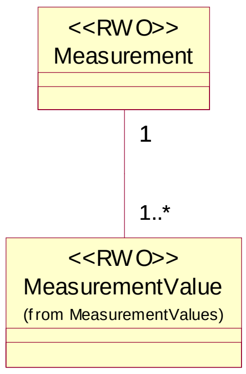


Figure 4-3 Utility SCADA Measurement Modeling

The Measurement and the MeasurementValue appear as Nodes and their properties as Items. In the example from Section 4.1.2, "Naming," on page 4-3, the Measurement again is represented by the pathname

Cobden.G1.P

Moving down to MeasurementValue will add its label to the pathname so that it may look like

Cobden.G1.P.Telem

If the MeasurementValue property for the value has the label Value finally the pathname for the item will be

Cobden.16kV.G1.P.Telem.Value

In a system also having a State Estimated MeasurementValue the pathname for state estimated value might be

Cobden.16kV.G1.P.SE.Value

Finally if the state estimated value is to replace the telemetered, this can be implemented picking the "best" MeasurementValue from Measurement itself shortcutting the MeasurementValue resulting in the path.

Cobden.G1.P.Value

The CIM classes that shall be navigable and whose data is exposed through DAIS are listed in the table below.

Table 4-2

CIM class	Properties and references
Measurement	All
MeasurementValue	All. MeasurementValues are expected to appear in the hierarchical structure so that all MeasurementValue instances are visible in the browser.
MeasurementUnit	All
LimitSet	All. The Measurement to LimitSet reference has a cardinality 1..*. It is expected that one LimitSet is the current used and exposed through DAIS.
Limit	As the LimitSet to Limit reference has the cardinality 1..* the mapping through DAIS is expected to be a number of properties where each property corresponds to one limit value.
ValueAliasSet	As the ValueAliasSet to ValueToAlias reference has cardinality 1..* the mapping through DAIS is expected to be a number of properties where each property corresponds to one translation from numeric to symbolic value.
Classes in the hierarchical structure above Measurement.	An implementation is free to expose any hierarchical structure of classes above the Measurement. Multiple and different structures are allowed using different views (for views refer to Section 3.2.2, "DAIS Server IDL," on page 3-24). An implementation is also free to expose any properties from these classes.

The table is based on the CIM version given by [11].

4.2 API

4.2.1 Data Access IDL Overview

The IDL is divided in files. Each file is modeled as a package in UML. A file that depends on declarations made in another file needs to include it. Figure 4-4 shows how the IDL files depend on each other.

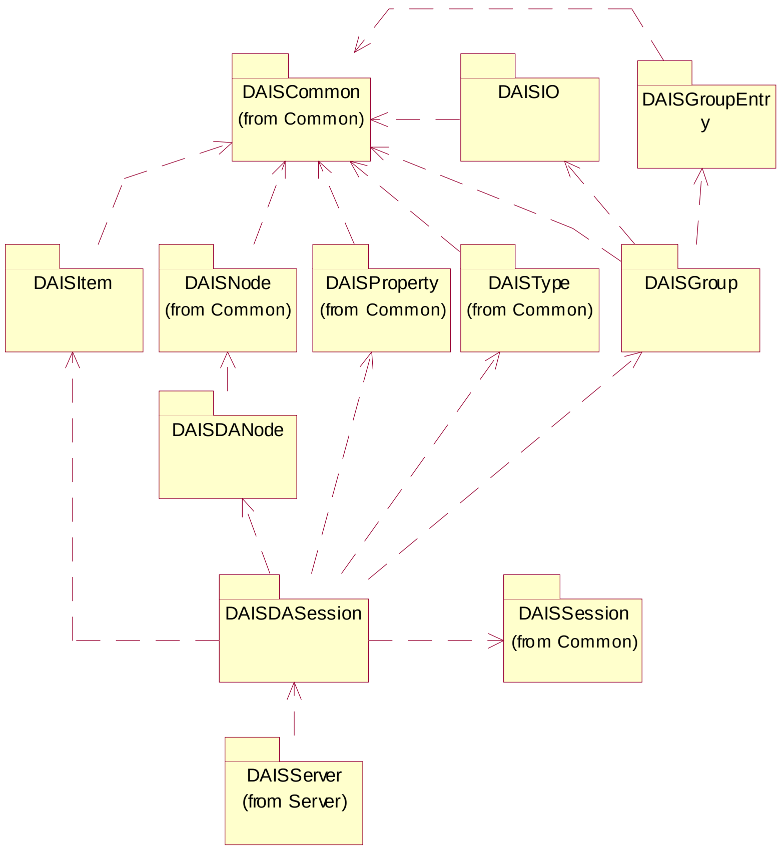


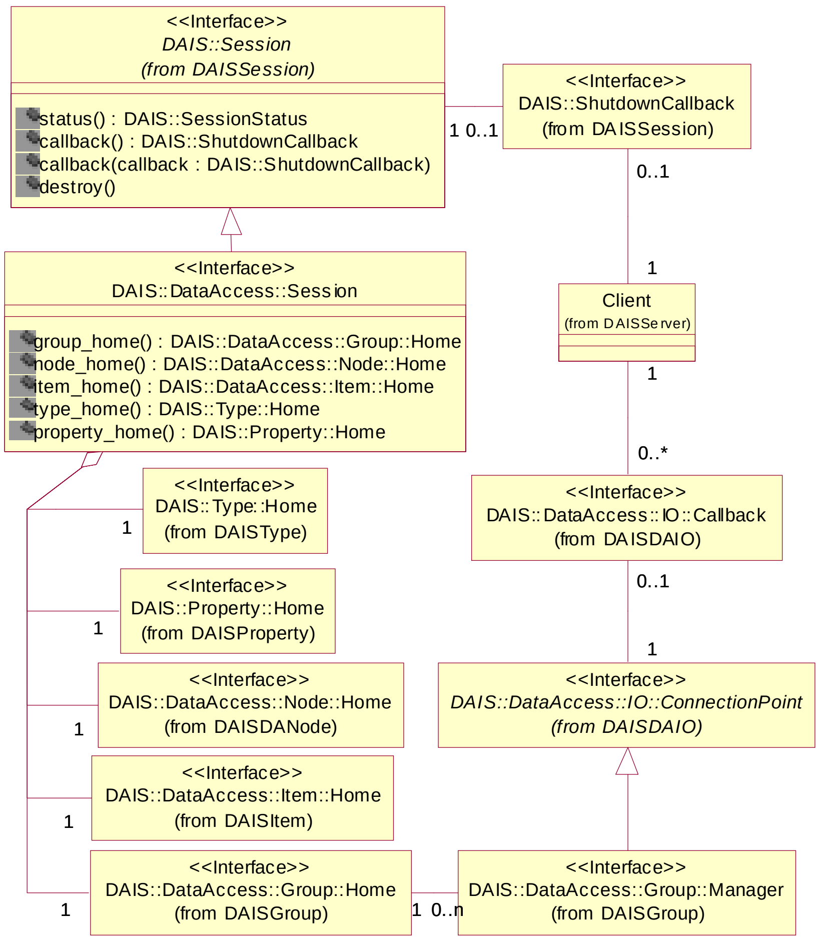
Figure 4-4 Dependencies between data access IDL files

4.2.2 DAISDASession IDL

4.2.2.1 DAIS::DataAccess::Session objects overview

The **DAIS::DataAccess::Session** object implements the data access service on a per client basis. A data access session object has a number of services provided by one singleton home object each. Each home object provides methods for manipulation of the data of the specific type they provide. Rather than exposing data as objects with interfaces it is exposed as structs or sequences of structs. The reason is that a large number of data items will become a performance bottleneck if instantiated as objects over an interface. The **DAIS::Group::Home** object is the only object to expose its data as objects; that is, the **DAIS::Gorup::Manager**. Each **DAIS::Group::Manager** is expected to connect to one callback object implemented by the client.

Each client may instantiate one or more Sessions. The Session objects have one Type and Property Home object each. The client shall expect that all Type and Property Home objects expose the same types and properties.



<<Interface>>
DAIS::Session
(from DAISSession)
status() : DAIS::SessionStatus
callback() : DAIS::ShutdownCallback
callback(callback : DAIS::ShutdownCallback)
destroy()
<<Interface>>
DAIS::ShutdownCallback
(from DAISSession)
1 0..1
0..1
1
Client
(from DAISServer)
1
0..*
<<Interface>>
DAIS::DataAccess::Session
group_home() : DAIS::DataAccess::Group::Home
node_home() : DAIS::DataAccess::Node::Home
item_home() : DAIS::DataAccess::Item::Home
type_home() : DAIS::Type::Home
property_home() : DAIS::Property::Home
<<Interface>>
DAIS::Type::Home
(from DAISType)
1
<<Interface>>
DAIS::DataAccess::IO::Callback
(from DAISDAIO)
0..1
1
<<Interface>>
DAIS::Property::Home
(from DAISProperty)
1
<<Interface>>
DAIS::DataAccess::Node::Home
(from DAISDANode)
1
<<Interface>>
DAIS::DataAccess::IO::ConnectionPoint
(from DAISDAIO)
<<Interface>>
DAIS::DataAccess::Item::Home
(from DAISItem)
1
<<Interface>>
DAIS::DataAccess::Group::Home
(from DAISGroup)
1
1 0..n
<<Interface>>
DAIS::DataAccess::Group::Manager
(from DAISGroup)

Figure 4-5 DAIS data access session IDL in UML

4.2.2.2 IDL

```
module DAIS {
module DataAccess {

interface Session : DAIS::Session
{

    readonly attribute Group::Homegroup_home;

    readonly attribute Node::Homenode_home;

    readonly attribute Item::Homeitem_home;
```

```
        readonly attribute Type::Hometype_home;

        readonly attribute Property::Homeproperty_home;
};};};
```

Session

Session is an object implementing the data access functions. It inherits common functionality as shut down callbacks and session status from **DAIS::Session**.

group_home

A read only attribute holding a reference to a singleton **Group::Home** object.

node_home

A read only attribute holding a reference to a singleton **Node::Home** object.

item_home

A read only attribute holding a reference to a singleton **Item::Home** object.

type_home

A read only attribute holding a reference to a singleton **Type::Home** object.

property_home

A read only attribute holding a reference to a singleton **Property::Home** object.

4.2.3 DAISDANode IDL

4.2.3.1 DAIS::DataAccess::Node overview

The **DAIS::DataAccess::Node** inherits most of the functionality from **DAIS::Node**. The only difference is that it supports to get the tree root node.

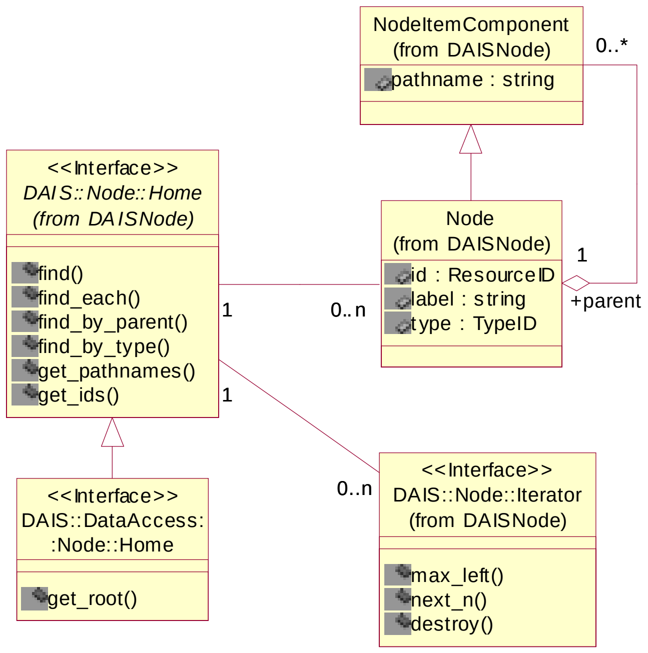


Figure 4-6 DAIS data access node IDL in UML

4.2.3.2 IDL

```
module DAIS {
module DataAccess {
module Node {

interface Home : DAIS::Node::Home
{

ResourceID get_root();
};
};};};
```

Home

An object used for browsing nodes. Most functionality is inherited from the **DAIS::Node::Home** interface.

get_root()

Get the root node of the whole tree of nodes.

Parameter	Description
return	The root node identification.

Hierarchical browsing

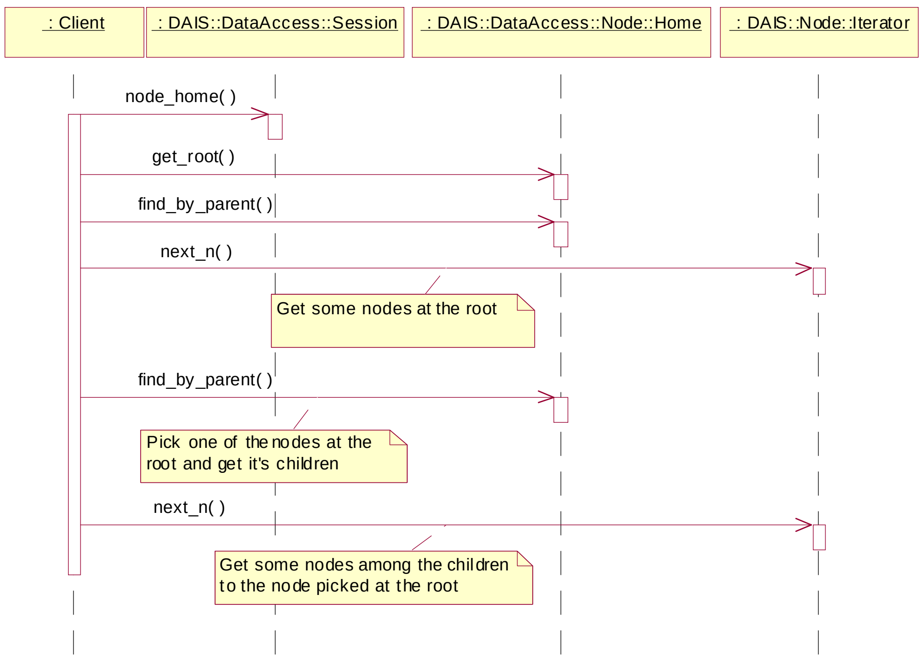

: Client
: DAIS::DataAccess::Session
: DAIS::DataAccess::Node::Home
: DAIS::Node::Iterator
node_home( )
get_root( )
find_by_parent( )
next_n( )
Get some nodes at the root
find_by_parent( )
Pick one of the nodes at the root and get it's children
next_n( )
Get some nodes among the children to the node picked at the root

Figure 4-7 Hierarchical browsing interaction

Browsing by type

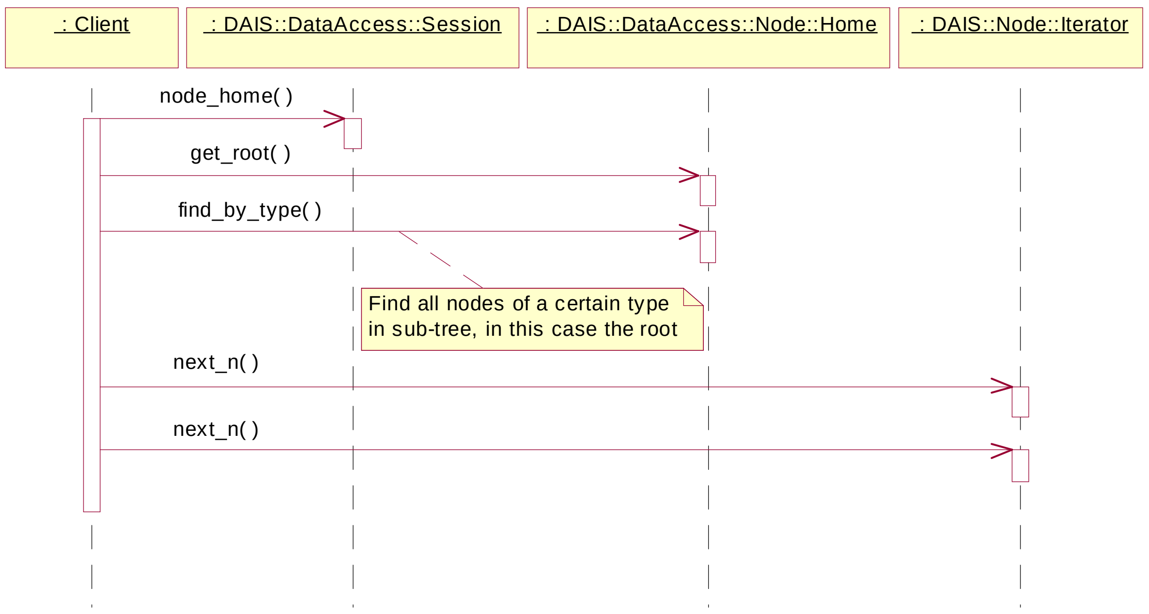

: Client
: DAIS::DataAccess::Session
: DAIS::DataAccess::Node::Home
: DAIS::Node::Iterator
node_home( )
get_root( )
find_by_type( )
Find all nodes of a certain type in sub-tree, in this case the root
next_n( )
next_n( )

Figure 4-8 Browsing by type interaction

4.2.4 DAISItem IDL

4.2.4.1 DAIS::Item overview

An item is a property of a node. While a node generally represents a real-world object, an item represents some characteristic of that object (for example, measurement value, control variable, or parameter related to the measurement/control process). The concept of an item in DAIS corresponds to the item in OPC. More precisely, it corresponds to the "leaf" in the OPC server namespace-browsing interface. In the RDF data model, the item corresponds to a combination of a subject and a predicate; that is, a resource and a property.

A node may have many items, each representing a different characteristic of the same real world object. An item might represent a measured variable, a calculated variable, a control variable, or a configuration parameter. An item will typically have many values where each value corresponds to a time stamped sample (a single item value corresponds to a single statement in the RDF model.). Each sample will also have its own quality. An item value is then qualified by a time stamp and a quality.

An item value and its qualifications are represented by six fixed attributes:

- value, the value
- time_stamp, the time when the value was last updated.

- quality, the quality of the value.
- canonical_data_type, the data type of the value. This actually belongs to the property (see Property) but it is mirrored at the item.
- access_rights, tells if the value is read only, write only, or both read/write. The access right is common for all item values and belongs to the item.
- scan_rate, the fastest rate with which the value can be expected to be updated. The scan rate is common for all item values and belongs to the item.

For an information model describing this refer to Section 4.1.1, “Nodes, Items, Types, and Properties,” on page 4-2 and Section 4.1.3, “Item Values,” on page 4-4.

Each item has a universal identity given by its **ItemID**. The **ItemID** is made up of the **ResourceID** of a node and the **PropertyID** of a property. The **ItemID** of an item the same in all views provided by a DAIS server. Clients may construct **ItemID**s given the identities of nodes and properties.

DAIS servers may be coordinated with DAF servers so that valid **ItemID**s can be constructed from DAF **ResoureID**s.

Within each view provided by the server, an item has a label that is unique among all items belonging to the same node. Within each view, an item has a unique pathname. The pathname is a string that contains the item’s label and the pathname of its node.

The pathname must be a valid URI, but apart from that the syntax of pathnames is implementation dependent.

Note – In OPC the pathname is the primary way to identify items and is called the ItemID. In DAIS the ResourceID, PropertyID pair is the primary identification and so it is called the ItemID. This is a potential point of confusion. The two ItemIDs play approximately the same role in OPC and DAIS respectively, but they are not the same type.

The item interfaces permit the stock of items to be browsed. Once an item is located, the group interfaces are used to deliver its values, selected by source, at successive times. In addition, the item interfaces provide the most current value for the default source. In either case the item value and qualifications are represented by the six fundamental attributes listed above.

The Item IDL defines a main interface **Home** for browsing among hierarchically structured items (leaf nodes). The information model describing the hierarchical structure is found in Section 4.1.1, “Nodes, Items, Types, and Properties,” on page 4-2.

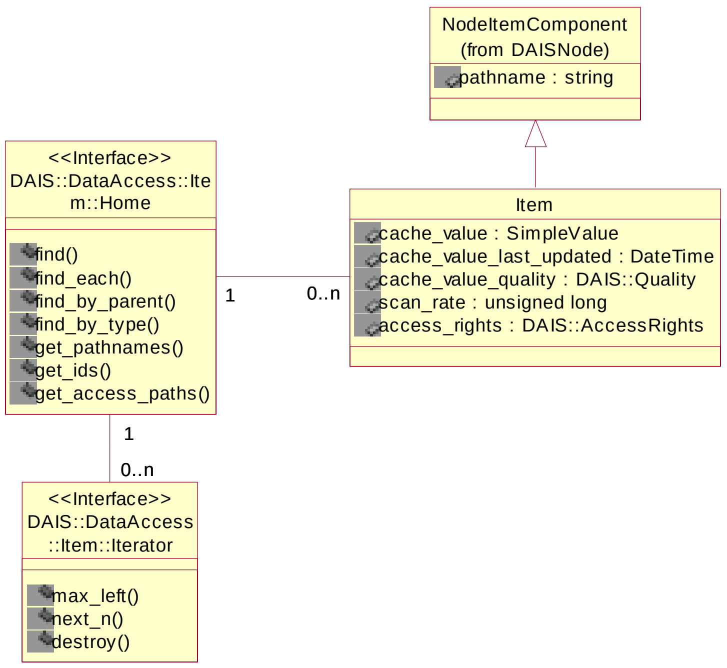

NodeItemComponent
(from DAISNode)
pathname : string
<<Interface>>
DAIS::DataAccess::Item::Home
find()
find_each()
find_by_parent()
find_by_type()
get_pathnames()
get_ids()
get_access_paths()
1
0..n
Item
cache_value : SimpleValue
cache_value_last_updated : DateTime
cache_value_quality : DAIS::Quality
scan_rate : unsigned long
access_rights : DAIS::AccessRights
1
0..n
<<Interface>>
DAIS::DataAccess::Item::Iterator
max_left()
next_n()
destroy()

Figure 4-9 DAIS data access item IDL in UML

4.2.4.2 IDL

```
module DAIS {
module DataAccess {
module Item {

struct Description
{
    ItemID                          id;
    string                          label;
    SimpleValue                 value;  //includes the canonical_data_type
    Quality                         dais_quality;
    DateTime                    time_stamp;
    AccessRights            access_rights;
    unsigned long           scan_rate;
};
typedef sequence< Description >Descriptions;

interface Iterator
{
    boolean next_n (
        in   unsigned long      n,
        out Descriptions        items
    );
    void destroy();
};

interface Home
{
    exception UnknownResourceID {string reason;};
    exception UnknownItemID {string reason;};
    exception InvalidFilter {string reason;};

    Description find (
        in  ItemID                    item
    ) raises (UnknownItemID);

    Descriptions find_each(
        in  ItemIDs                   items
    ) raises (UnknownItemID);

    Iterator find_by_parent (
        in  ResourceID            node,
        in  string                    filter_criteria,
        in  SimpleValueType    data_type_filter,
        in  AccessRights       access_rights_filter
    ) raises (UnknownResourceID, InvalidFilter);

    Iterator find_by_type (
        in  ResourceID            node,
        in  TypeIDs                   type_filter,
        in  string                    filter_criteria,
        in  SimpleValueType    data_type_filter,
        in  AccessRights       access_rights_filter
    ) raises (UnknownResourceID, InvalidFilter);

    Strings get_pathnames (
        in  ItemIDs                   items
    );

    ItemIDs get_ids (
        in  Strings                   pathnames
    );

    Strings get_access_paths (
        in  ItemID                    item
    ) raises (UnknownItemID);
};};};};
```

Description

A struct describing an item.

Member	Description
id	The identification of this item.
label	The label (single level designation) of the item.
value	The current value sample for the item. The SimpleValue also contains the data type.
dais_quality	The current quality of the value.
time_stamp	The time stamp for the value sample.
access_rights	Tells if the value is read, write, or both read and write.
scan_rate	Tells the highest update rate that can be expected.

Iterator

Refer to Section 4.1.5, "Utility SCADA/EMS Measurement Model," on page 4-6. This interface corresponds to the OPC interface EnumString with the difference that the Iterator return the Description struct instead of a single string.

ItemHome

An object used for browsing items and corresponds to the **IOPCBrowseServerAddressSpace** with the BrowseFilterType set to OPC_LEAF. A major difference to OPC is that the server does not provide a cursor for clients. Instead clients has to provide the browse position in each call.

UnknownResourceID

An exception telling that the **ResourceID** is unknown. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

UnknownItemID

An exception telling that the resource or property in the **ItemID** is unknown. For methods taking a sequence of item ids the first found unknown id is reported. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

InvalidFilter

An exception telling the **filter_criteria** string is not correct. The likely reason behind this exception is an erroneously entered string.

find()

For a given item browse position return information about that item.

Parameter	Description
item	An item identification.
return	The item description.

find_each ()

For a sequence of items return information about each item.

Parameter	Description
items	A sequence of item identifications.
return	An iterator holding the item descriptions.

find_by_parent ()

For a given node identification return child items to that node.

Parameter	Description
node	The parent node identification.
filter_criteria	A server specific filter string. This is entirely free format and may be entered by the user via a text field. An empty string indicates no filtering. Select only items matching the filter criteria.
data_type_filter	Select items having the specified canonical data type.
access_rights_filter	Select items having the specified access rights.
return	An iterator holding item descriptions for items • that are child to the parent node. • matching the **filter_criteria**, **data_type_filter** and **access_rights_filter**.

find_by_type()

For a sub-tree given by a node return all child items matching the filter criteria.

Parameter	Description
node	The identification for the node defininig the sub-tree.

type_filter	Select nodes in the sub-tree having a type matching any of the types listed in the **type_filter**.
filter_criteria	A server specific filter string. This is entirely free format and may be entered by the user via a text field. An empty string indicates no filtering. Select only items matching the filter criteria.
data_type_filter	Select items having the specified canonical data type.
access_rights_filter	Select items having the specified access rights.
return	An iterator holding item descriptions for items • that are child to nodes in the sub-tree and nodes having a type matching the **type_filter**. • matching the **filter_criteria**, **data_type_filter** and **access_rights_filter**.

get_pathnames()

Translate a sequence of item identifications to the corresponding sequence of pathnames. If an item fails to translate to a pathname (due to an unknown identification), the corresponding pathname is an empty string.

Parameter	**Description**
items	The sequence of items.
return	The corresponding sequence of pathnames.

get_ids()

Translate a sequence of pathnames to the corresponding sequence of node identifications. If a pathname fails to translate to node identification (due to an unrecognized pathname), the corresponding node identification is NULL.

Parameter	**Description**
pathnames	The sequence of pathnames.
return	The corresponding sequence of item identifications.

get_access_paths()

Get the possible communication paths how data can be retrieved for the node. An access path is expected to be human readable so that a human can pick one and feed it back to the server as the preferred path (via other interfaces).

Parameter	Description
item	An item identification.
return	A sequence of possible access paths for the item.

Browsing Items

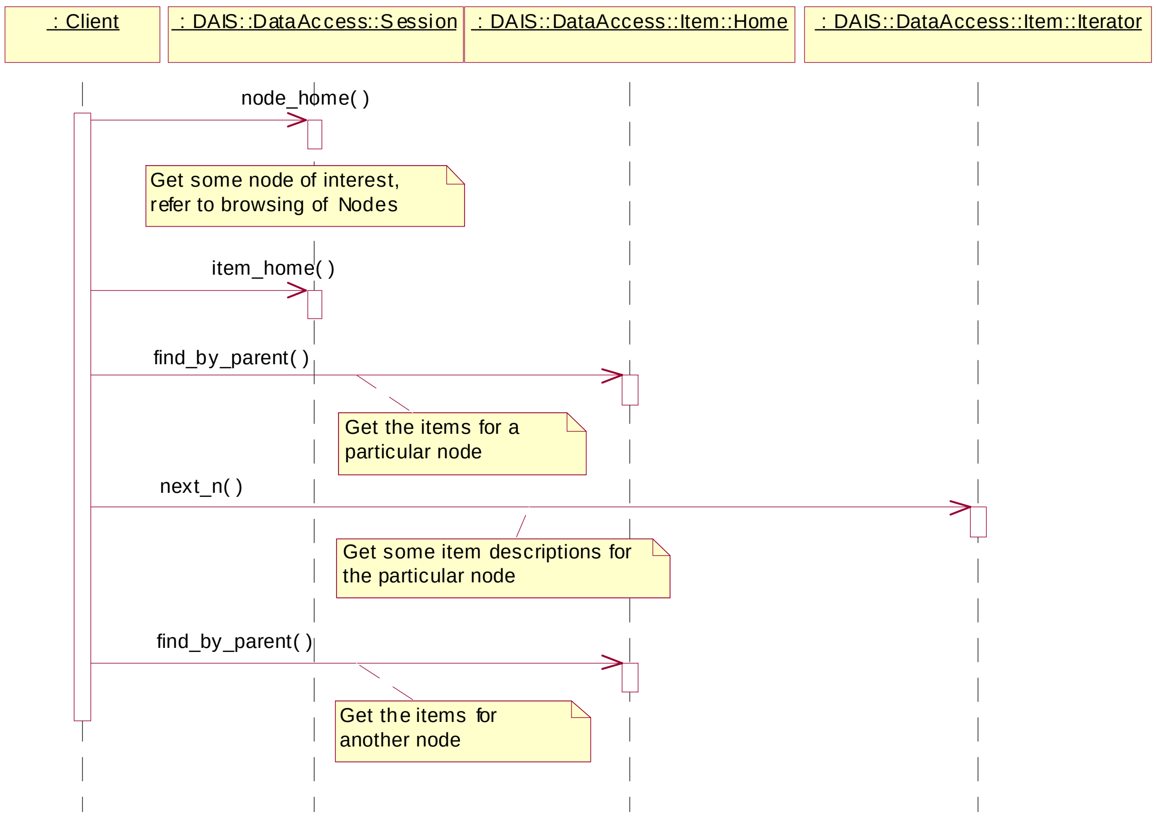
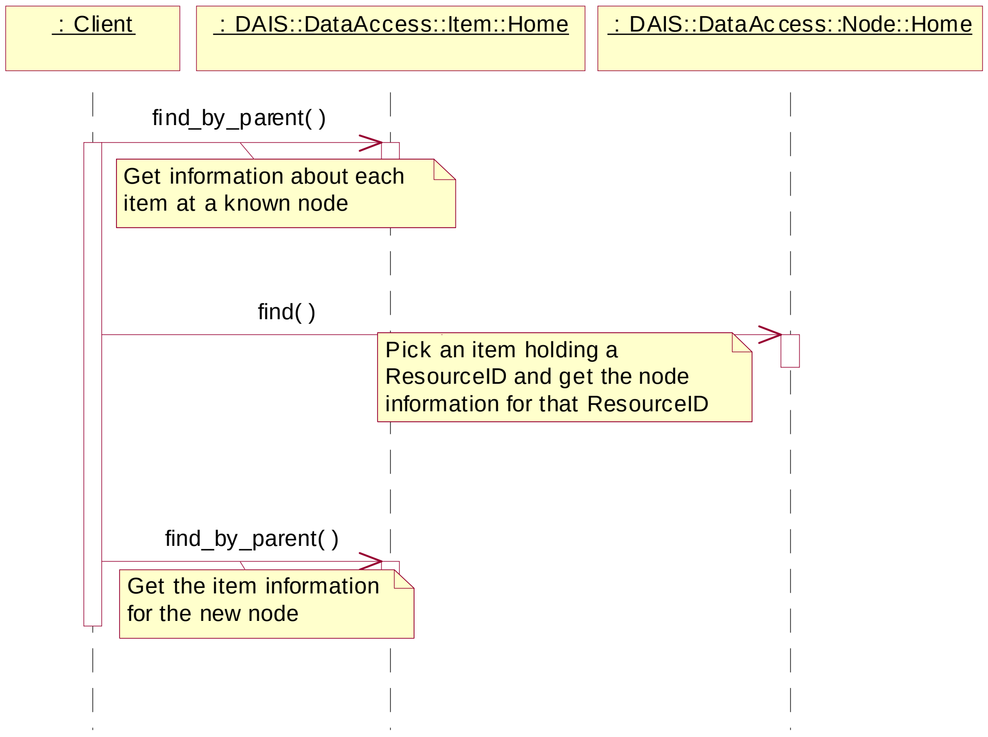

: Client
: DAIS::DataAccess::Session
: DAIS::DataAccess::Item::Home
: DAIS::DataAccess::Item::Iterator
node_home( )
Get some node of interest, refer to browsing of Nodes
item_home( )
find_by_parent( )
Get the items for a particular node
next_n( )
Get some item descriptions for the particular node
find_by_parent( )
Get the items for another node

Figure 4-10 Browsing items interaction

Navigate across associations



: Client
: DAIS::DataAccess::Item::Home
: DAIS::DataAccess::Node::Home
find_by_parent( )
Get information about each item at a known node
find( )
Pick an item holding a ResourceID and get the node information for that ResourceID
find_by_parent( )
Get the item information for the new node

Figure 4-11 Navigating across associations interaction

4.2.5 DAISDAIO IDL

4.2.5.1 DAIS::DataAccess::IO overview

These are definitions for transmitting item values to clients. Interfaces are defined for server side read and write operations and client side callback operations. Clients shall implement the **Callback** object for the server to use at transfer of data. A client may have any number of callback objects. The client shall connect each callback object to a server object implementing **ConnectionPoint**.

The IO interfaces support three different ways to read data and two different ways to write data.

Read data

- synchronous read where the data is received at return from the **sync_read()** method.
- asynchronous read where the data is returned at the **Callback** object.

- subscription where data is sent spontaneously by the server at the callback object.

Write data

- synchronous write returning to the client once the all written data has reached the devices.
- asynchronous write returning when the data is received by the DAIS server. A callback on **Callback** is sent by the server once the written data has reached the devices.

Each item value is transmitted in a struct with a timestamp and quality indication. A sequence of this struct is either sent via the callback object or directly in read or write calls.

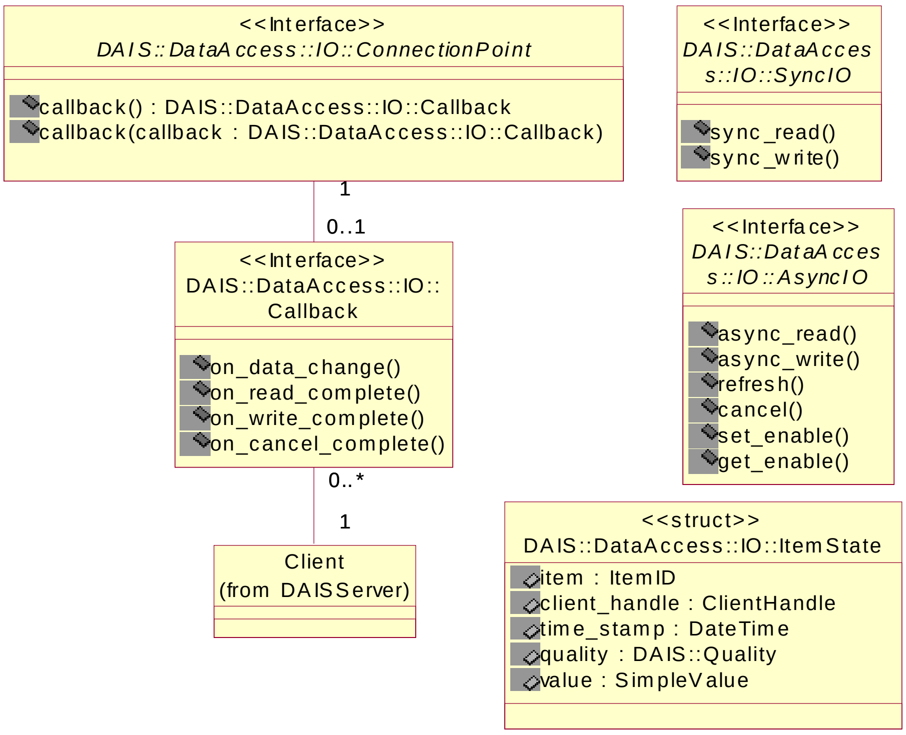


Figure 4-12 DAIS data access IO IDL in UML

The **DAISConnectionPoint callback()** methods correspond to a get or set method for the callback attribute.

4.2.5.2 IDL

```
module DAIS {

module DataAccess {
module IO {

enum DataSource {
   DS_CACHE,
   DS_DEVICE
};

struct ItemState
{
     ItemID                          item;
     ClientHandle                    client_handle;
     DateTime                        time_stamp;
     Quality                         dais_quality;
     SimpleValue                     value;
};
typedef sequence<ItemState>ItemStates;

interface SyncIO
{

     ItemStates sync_read (
          in   DataSource                     data_source,
          in   ItemIDs                             items,
          out ItemErrors                      errors
     );

     ItemErrors sync_write (
          in   ItemStates                     states
     );
};

typedef unsigned long CancelID;

interface AsyncIO
{
     exception NotConnected{string reason;};
     exception InvalidCancelID{string reason;};
     exception NotActive{string reason;};

     CancelID async_read (
          in   ItemIDsitems,
          in   DataSource                     data_source,
          in   unsigned long        transaction_id
     ) raises (NotConnected, NotActive);
```

```
    CancelID async_write (
        in  ItemStates              states,
        in  unsigned long       transaction_id
    ) raises (NotConnected);

    CancelID refresh (
        in  DataSource              data_source,
        in  unsigned long       transaction_id
    ) raises (NotConnected,NotActive);

    void cancel (
        in  CancelID                cancel_id
    ) raises (InvalidCancelID);

    attribute boolean enabled;
};

interface Callback
{

    void on_data_change (
        in  unsigned long       transaction_id,
        in  boolean                     all_quality_good,
        in  ItemStates              states
    );

    void on_read_complete (
        in  unsigned long       transaction_id,
        in  boolean                     all_quality_good,
        in  ItemStates              states,
        in  ItemErrors              errors
    );

    void on_write_complete (
        in  unsigned long       transaction_id,
        in  ItemErrors              errors
    );

    void on_cancel_complete (
        in  unsigned long       transaction_id
    );
};

interface ConnectionPoint
{

    attribute Callback      cllbck;
};};};};
```

DataSource

Member	Description
DS_CACHE	Data cached in the server is requested.
DS_DEVICE	Data from the device is requested. This will force a read from the device or RTU. A read from device will be made regardless of the group or item active status and no group NotActive exception will be forced.

ItemState

The struct is the major carrier of data conveyed over the interface. It is the “message” holding the payload.

Member	Description
item	The identification of the item.
client_handle	A client side handle enabling the client to make a quick look up of the item in its internal data structures.
time_stamp	The time stamp when the value was last updated.
quality	The quality for the value.
value	The value itself.

SyncIO

An interface for the synchronous operations.

sync_read()

Synchronous read of items. Inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.

Parameter	Description
data_source	The source from where to read the data.
items	A sequence specifying the whole or a subset of the items defined via the **DAIS::GroupEntry::Manager** interface.
errors	A sequence reporting items for which the read failed. An empty sequence indicates all read operations succeeded. Reported errors are: ERROR_BAD_RIGHTS (item is write only) ERROR_UNKNOWN_ITEMID

return	A sequence of ItemStates for the items.

sync_write()

Synchronous write of item values to devices (not the internal server cache). The active state of the group or the items is ignored.

Parameter	**Description**
states	A sequence of ItemStates specifying all or a subset of the items defined for a **GroupEntry::Manager**. The **ItemStates::value member** is used to update the items in devices.
return	A sequence reporting items for which the write failed. An empty sequence indicates write operations for all items succeeded. Reported errors are: ERROR_BAD_RIGHTS (item is read only) ERROR_UNKNOWN_ITEMID ERROR_CLAMPED ERROR_OUT_OF_RANGE ERROR_BAD_TYPE

AsyncIO

An interface for asynchronous read or write operations.

NotConnected

An exception telling there is no callback object connected by the client.

InvalidCancelID

An exception telling the supplied cancel id number is not recognized.

NotActive

An exception telling the group or all items in the group is inactive. Only issued when read from cache.

async_read()

Asynchronous read of items from devices. OPC may report read errors both at return from **async_read()** and at **on_read_complete()**. DAIS will report all errors at **on_read_complete()**.

.

Parameter	Description
items	A sequence specifying the whole or a subset of the items defined for a **DAIS::GroupEntry::Manager**.
data_source	The source from where to read the data. When reading from cache inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.
transaction_id	A transaction number unique for the client. The number is returned in the corresponding **on_read_complete** call.
return	A cancellation number unique for the client. The number is used by a client to cancel an ongoing asynchronous read operation.

async_write()

Asynchronous write of item values to devices (not the internal server cache). OPC may report write errors both at return from **async_write()** and at **on_write_complete()**. DAIS will report all errors at **on_write_complete()**.

Parameter	Description
states	A sequence of ItemStates specifying all or a subset of the items defined for a **DAIS::GroupEntry::Manager**. The **ItemStates::value** member is used to update the items in devices.
transaction_id	A transaction number unique for the client. The number is returned in the corresponding **on_read_complete** call.
return	A cancellation number unique for the client. The number is used by a client to cancel an ongoing asynchronous write operation.

refresh()

Initiate a complete asynchronous read transfer for all item entries defined via the **DAIS::DataAccess::GroupEntry::Manager** interface. Inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.

The cyclic **on_data_change** reporting continues unaffected by a refresh call. However, items still unchanged after a refresh will not be reported in a succeeding **on_data_change** call.

Parameter	Description
data_source	The source from where to read the data.
transaction_id	A transaction number unique for the client. The number is returned in the corresponding **on_data_change** call.
return	A cancellation number unique for the client. The number is used by a client to cancel an ongoing asynchronous refresh operation.

cancel()

Cancel on ongoing refresh, async read, or async write operations. The server is expected to acknowledge a successfully initiated cancel operation with an **on_cancel_complete()** callback.

Parameter	Description
cancel_id	The server generated cancellation number for the operation to cancel.

enable

An attribute used to enable or disable the spontaneous **on_data_change()** callbacks. The enable state does not affect on data change response to refresh calls. When a group is created it is enabled by default.

Callback

An interface implemented by the client and used by the server to send data to the client.

on_data_change()

The method is called by the server when spontaneous changes occur or when the client has requested an explicit refresh. Only active items are reported in spontaneous calls.

Parameter	Description
transaction_id	If the call is in response to a refresh, the transaction number for that refresh call. If the call is autonomous due to one or more spontaneous changes, the number is zero.
all_quality_good	All item quality values are good.

item_states	A sequence of requested or spontaneously changed ItemStates.

on_read_complete()

The method is used by the server to report data in response to an asynchronous read.

Parameter	**Description**
transaction_id	The transaction number for the corresponding read.
all_quality_good	All item quality values are good. This requires that no errors are reported in the error parameter below.
item_states	A sequence of ItemStates matching the read operation.
errors	A sequence reporting items for which the read failed. An empty sequence indicates all read operations initially succeeded. Reported errors are: ERROR_BAD_RIGHTS, item is write only ERROR_UNKNOWN_ITEMID

on_write_complete()

The method is used to report the success of an asynchronous write operation.

Parameter	**Description**
transaction_id	The transaction number for the corresponding write.
errors	A sequence reporting items for which the write failed. An empty sequence indicates all write operations initially succeeded. Reported errors are: ERROR_BAD_RIGHTS (item is read only) ERROR_UNKNOWN_ITEMID ERROR_CLAMPED ERROR_OUT_OF_RANGE ERROR_BAD_TYPE

on_cancel_complete()

The method is used to acknowledge the completion of a successfully initiated cancel call.

Parameter	**Description**
transaction_id	The transaction number for the corresponding cancel.

ConnectionPoint

An interface used by the client to connect or disconnect a client callback object at the server.

callback

An attribute referencing the callback object.

In an implementation one get and one set method will implement the callback attribute. Due to limitation in the UML tool used to draw the diagrams the attribute is represented by the two methods connect and disconnect.

4.2.6 DAISGroupEntry IDL

4.2.6.1 DAIS::DataAccess::GroupEntry overview

A group has a collection of group entries. Each group entry associates the group with an item. An **ItemID** identifies a group entry within its group. The pathname of an item may be used as an alternative to the **ItemID** when the group entry is created (see Section 4.2.4, “DAISItem IDL,” on page 4-14).

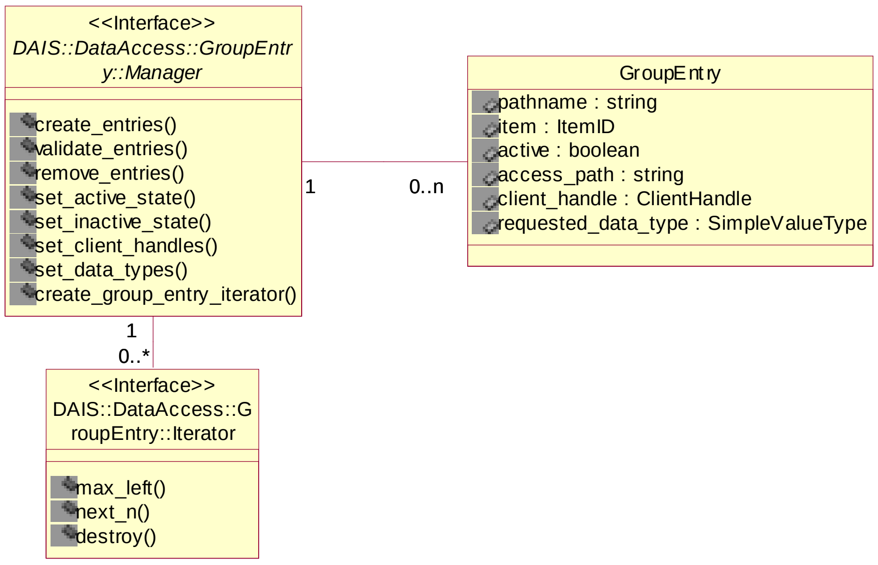


Figure 4-13 DAIS data access group entry IDL in UML

4.2.6.2 IDL

```
module DAIS {
module DataAccess {
module GroupEntry {

struct Description
{
    string                      pathname;
    ItemID                      item;
    boolean                     active;
    string                      access_path;
    ClientHandle                client_handle;
    SimpleValueType             requested_data_type;
};
typedef sequence<Description>Descriptions;

struct DetailedDescription
{
    Description                     entry;
    AccessRights                access_rights;
    SimpleValueType             canonical_data_type;
};
typedef sequence<DetailedDescription> DetailedDescriptions;

interface Iterator
{
    boolean next_n (
        in  unsigned long       n,
        out DetailedDescriptionsentries
    );
    void destroy();
};

interface Manager
{

    DetailedDescriptions create_entries (
        in  Descriptionsentries,
        out ItemErrors                  errors
    );

    DetailedDescriptions validate_entries (
        in  Descriptionsentries,
        out ItemErrors                  errors
    );

    ItemErrors remove_entries (
        in  ItemIDs                         items
    );
    ItemErrors set_active_state (
        in   ItemIDs                          items
    );


    ItemErrors set_inactive_state (
        in   ItemIDs                          items
    );

    ItemErrors set_client_handles (
        in   Descriptions          entries
    );

    ItemErrors set_data_types (
        in   Descriptions          entries
    );

    Iterator create_group_entry_iterator ();
};};};};
```

Description

The struct describe a group entry for an item. The client to configure new entries in a group uses it. It directly corresponds to the **IOPCItemDef** struct.

Member	Description
pathname	A string concatenating the labels for all nodes in the path from the item up to the root.
item	The identification of the item.
active	Tells if the item is active and data from devices is updated in the cache.
access_path	The access path used by the server to connect to the device and sensor. An empty string as input tells the server to select the access path.
client_handle	The client provided handle to the item.
requested_data_type	The data type requested by the client for the value.

DetailedDescription

The struct is the same as **Description** but with more information. The struct is used to deliver group entry information to the client. In OPC this is made with the two structs **IOPCItemAttributes** and **IOPCItemResult**.

Description is used for both these OPC structs.

Member	Description
entry	The above described **Description**.
access_rights	The access rights (read, write and read-write)
canonical_data_type	The data type the server use internally for the value.

Iterator

Refer to Section 3.1.5, "Iterator Methods IDL," on page 3-10. The interface directly corresponds to the **EnumOPCItemAttributes** interface.

Manager

An interface for creation and browsing of group entries. The interface directly corresponds to the **IOPCItemMgt** interface.

create_entries()

Adds one or more entries to a group.

Parameter	Description
entries	Group entry descriptions for entries to be created.
errors	A sequence of structs reporting the items that was not entered due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID ERROR_UNKNOWN_PATHNAME ERROR_BAD_TYPE ERROR_UNKNOWN_ACCESS_PATH
return	A sequence of detailed group entry descriptions for the entries that was entered in the group.

Depending on how the Description members item and pathname are specified the behavior is different

- only item specified -> the corresponding pathname is returned in the **DetailedDescription** struct.
- only pathname specified -> the corresponding item is returned in the **DetailedDescription** struct.

- both item and pathname specified -> only item is used by the server and the corresponding pathname is returned in the **DetailedDescription** struct whatever pathname was specified as input.

validate_entries()

Is used to check if Descriptions a client holds are still valid without affecting the existing group.

Parameter	**Description**
entries	Group entry descriptions for entries to be validated.
errors	A sequence of structs reporting the items that could not be validated due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID ERROR_UNKNOWN_PATHNAME ERROR_BAD_TYPE ERROR_UNKNOWN_ACCESS_PATH
return	A sequence of detailed group entry descriptions for the entries that were validated.

The same rules for management of the item and pathname as described for **create_entries()** is used.

remove_entries()

Used to remove entries from a group.

Parameter	**Description**
items	Item identifications for entries that shall be removed.
return	A sequence of structs reporting the items that were not recognized or could not be removed due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID

set_active_state()

Used to activate individual items in a group. Activate state means that the server acquires data from devices. Inactive state means that the server does not acquire any data. The group enable state control if acquired data shall be sent further to subscribers via **on_data_change**.

Parameter	Description
items	Item identifications for items to activate.
return	A sequence of structs reporting the items that were not recognized or could not be activated due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID

set_inactive_state()

Used to deactivate individual items in a group.

Parameter	Description
items	Item identifications for items to deactivate.
return	A sequence of structs reporting the items that were not recognized or could not be deactivated due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID

set_client_handles()

Used to change the client handles for items.

Parameter	Description
entries	Group entry descriptions for group entries that will get the client handles changed. The new client handles are taken from the group entry descriptions.
return	A sequence of structs reporting the items that was not recognized or could not be updated due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID

set_data_types()

Used to change the requested data types for items.

Parameter	Description
entries	Group entry descriptions for group entries that will get the client handles changed. The new data types is taken from the group entry descriptions.
return	A sequence of structs reporting the items that were not recognized or could not be updated due to an error. Reported errors are: ERROR_UNKNOWN_ITEMID ERROR_BAD_TYPE

create_group_entry_iterator()

Used to create a group entry iterator. Used by clients to inspect existing group entries.

Parameter	Description
return	The GroupEntry Iterator.

4.2.7 DAISGroup IDL

4.2.7.1 DAIS::DataAccess::Group overview

A group is a collection of items and a connection to one or more consumers of item values. Clients create groups and their lifetime is bounded by the session to which they belong. (See Section 4.2.2, “DAISDASession IDL,” on page 4-9).

The purpose of a group is to convey selected item values to a client. A callback object may be connected to a group to receive item value information (see Section 4.2.5, “DAISDAIO IDL,” on page 4-22). Items may be added and removed from a group as group entries (see Section 4.2.6, “DAISGroupEntry IDL,” on page 4-31). A group has an update rate that determines how frequently updated values for its entries are notified to its connected callback objects. A group also has other state that controls its notification behavior.

A group may also be initialized with a predefined set of entries. A set of entries is called a public group and is identified by a **ResourceID**. A client can create or remove public groups. A server may represent a public group as a node such that the **ResourceID** of the public group and the node are identical. This would allow clients to locate public groups by name.

The **DAIS:: DataAccess::Group::Manager** object implement interfaces from the **DAISDAIO** and **DAISGroupEntry** IDLs. This is specified by inheritance of interfaces as seen in Figure 4-14. A **DAIS::DataAccess::Group::Manager** has a state given by the **DAIS::DataAccess::Group::State** struct and a **DAIS::DataAccess::Group::Manager** object is created from the **DAIS::DataAccess::Group::Home** object.

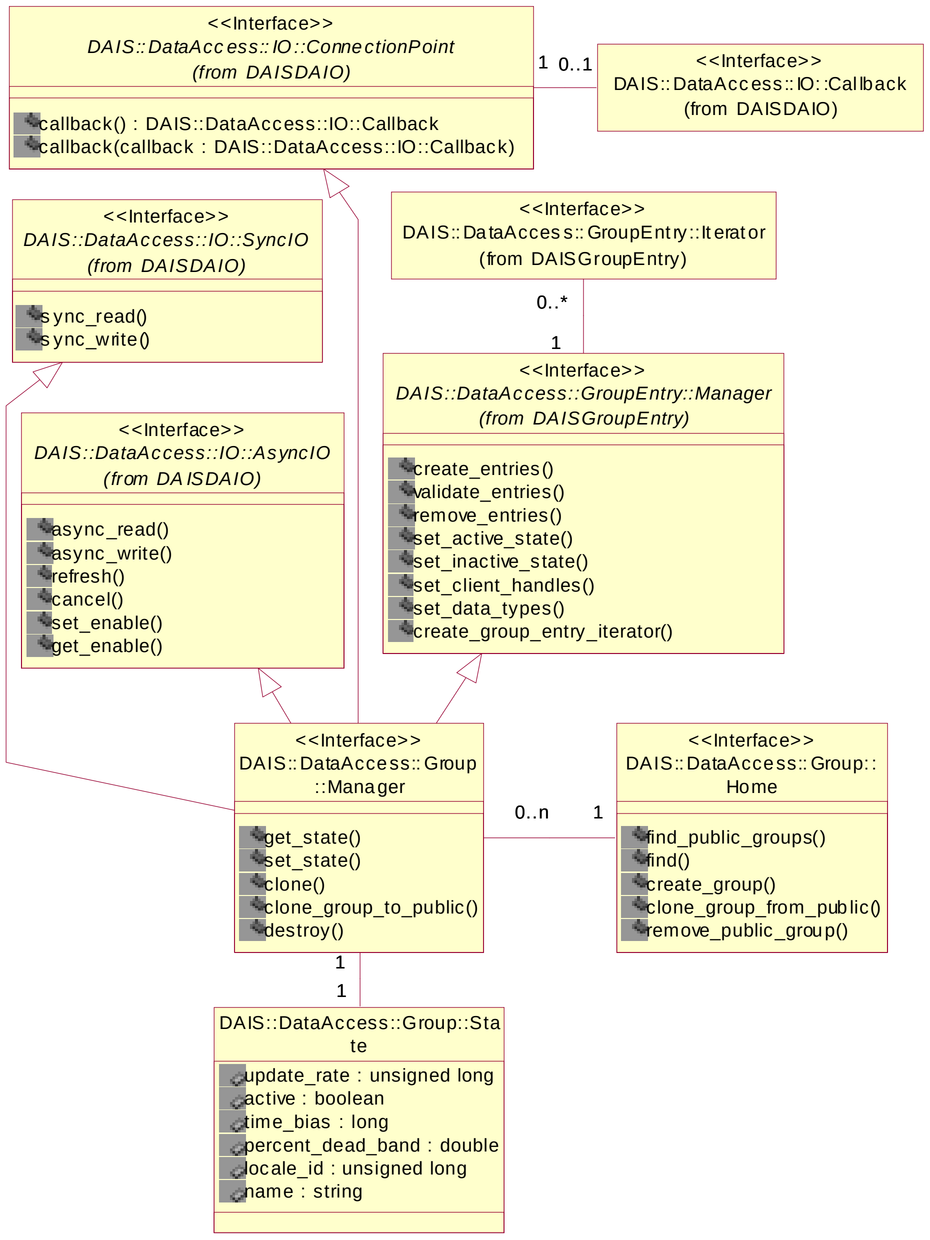


Figure 4-14 DAIS data access group IDL in UML

4.2.7.2 IDL

```
module DAIS {
module DataAccess {
module Group {

exception DuplicateName {string reason;};

struct State
{
    string                  name;
    unsigned long           update_rate;
    boolean                 active;
    long                    time_bias;
    double                  percent_deadband;
    unsigned long           locale_id;
};

interface Manager :
        GroupEntry::Manager
        ,IO::AsyncIO
        ,IO::SyncIO
        ,IO::ConnectionPoint
{

    State get_state ();

    unsigned long set_state (
        in   State                      group_state
    ) raises (DuplicateName);

    Manager clone (
        in   string                     name
    ) raises (DuplicateName);

    Manager clone_group_to_public (
        in   string                     name
    ) raises (DuplicateName);

    void destroy ();
};

struct PublicGroupDescription
{
    ResourceID                  id;
    State                           group_state;
};
typedef sequence<PublicGroupDescription>PublicGroupDescriptions;

interface Home
{
    exception UnknownResourceID {string reason;};

    PublicGroupDescriptions find_public_groups();

    PublicGroupDescription find (
        in   ResourceID                  public_group
    ) raises (UnknownResourceID);

    Manager create_group (
        in   State                           group_state,
        out unsigned long       revised_update_rate
    ) raises (DuplicateName);

    Manager clone_group_from_public (
        in   ResourceID                  public_group,
        in   string                          name
    ) raises (DuplicateName, UnknownResourceID);

    void remove_public_group (
        in   ResourceID                  public_group
    ) raises (UnknownResourceID);
};};};};
```

DuplicateName

An exception raised when an object is created and the name already exists. No object is created if the exception is raised. Is used for session and group manager objects.

State

The struct contain information about the group state.

Members	**Description**
name	Within the session and public groups unique name of the group.
update_rate	Update rate for the group in milliseconds. When used as input it specifies the fastest rate at which data changes may be sent to **on_data_change()** for items in this group. This also indicates the desired accuracy of cached data. This is intended only to control the behavior at the interface. How the server deals with the update rate and how often it actually polls the hardware internally is an implementation detail. Passing 0 indicates the server should use the fastest practical rate.

active	Indicate if the group is active and data from devices are updated in the cache.
time_bias	The time bias in minutes for the group. A zero value when used as input will tell the server to use the default system time bias. This bias behaves like the Bias field in the Win32 TIME_ZONE_INFORMATION structure.
percent_deadband	The percent change for an item value that will cause a call back for that value. This parameter only applies to items in the group that are of analog type. If a client specifies a zero deadband, the value will be reported with the update rate.
locale_id	The localization number for the language used when returning string values.

Manager

An object used to manage a group. It has a set of methods related to the group itself. It also inherits methods from interfaces for group entry management and data transfer. For group entry management refer to Section 4.2.6, "DAISGroupEntry IDL," on page 4-31 and for data transfer refer to Section 4.2.4, "DAISItem IDL," on page 4-14. The **DAIS::DataAccess::Group::Manager** interface corresponds to the **IOPCGroupStateMgt** interface.

get_state()

The method gets the group status.

Parameter	Description
return	State

set_state()

The method sets the group status.

Parameter	Description
group_state	The State with the updates. All members will be updated. If the name already exists, a **DuplicateName** exception is raised and no update is made.
return	The closest update rate the server is able to provide for the group.

clone()

Create a copy of a group.

Parameter	Description
name	The name to be given the new group. If the name already exists, a **DuplicateName** exception is raised and no clone is created.
return	The created copy of the group.

clone_group_to_public ()

Create a public copy of a group including all items and the group state.

Parameter	Description
name	The name to be given the new group. If the name already exists, a **DuplicateName** exception is raised and no public group is created.
return	The created copy of the group.

destroy()

Delete the group.

PublicGroupDescription

A struct describing public groups.

Member	Description
id	A **ResourceID** identifying the public group.
group_state	The group state struct including the group name.

Home

The factory object for groups. The corresponding OPC interface is **IOPCServer**.

UnknownResourceID

An exception telling that the **ResourceID** is unknown. For methods taking a sequence of resource ids the first found unknown id is reported. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

find_public_groups()

Find all public groups defined in the server.

Member	Description
return	A sequence of public group descriptions.

find()

Find the description of a public group with know identification.

Member	Description
public_group	A **ResourceID** identifying the public group.
return	A public group description.

create_group()

Create a new initially empty group.

Parameter	Description
group_state	The State to be set for the new group.
revised_update_rate	The closest update rate the server is able to provide for the group.
return	The new group.

clone_group_from_public()

Create a copy from a public group having an existing set of entries and state.

Parameter	Description
public_group	The identification of the public group.
name	The name of the new group.
return	The new group.

remove_public_group()

Remove a public group.

Parameter	Description
public_group	The identification of the public group.

return	None.

Group management

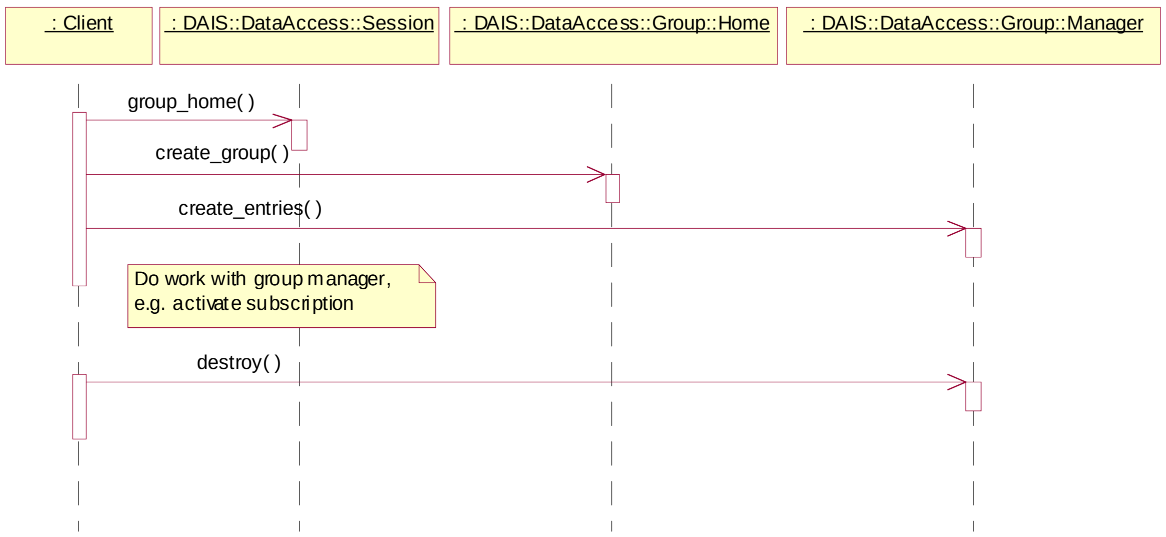

: Client
: DAIS::DataAccess::Session
: DAIS::DataAccess::Group::Home
: DAIS::DataAccess::Group::Manager
group_home( )
create_group( )
create_entries( )
Do work with group manager, e.g. activate subscription
destroy( )

Figure 4-15 Group management interaction

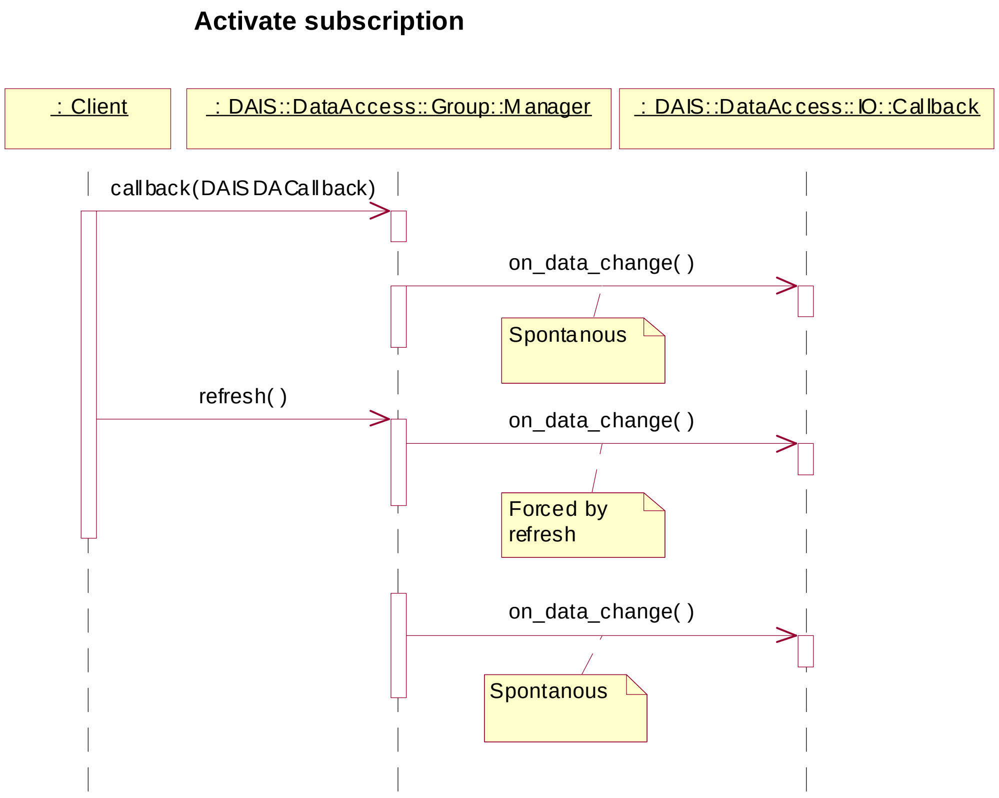


Figure 4-16 Active subscription interaction

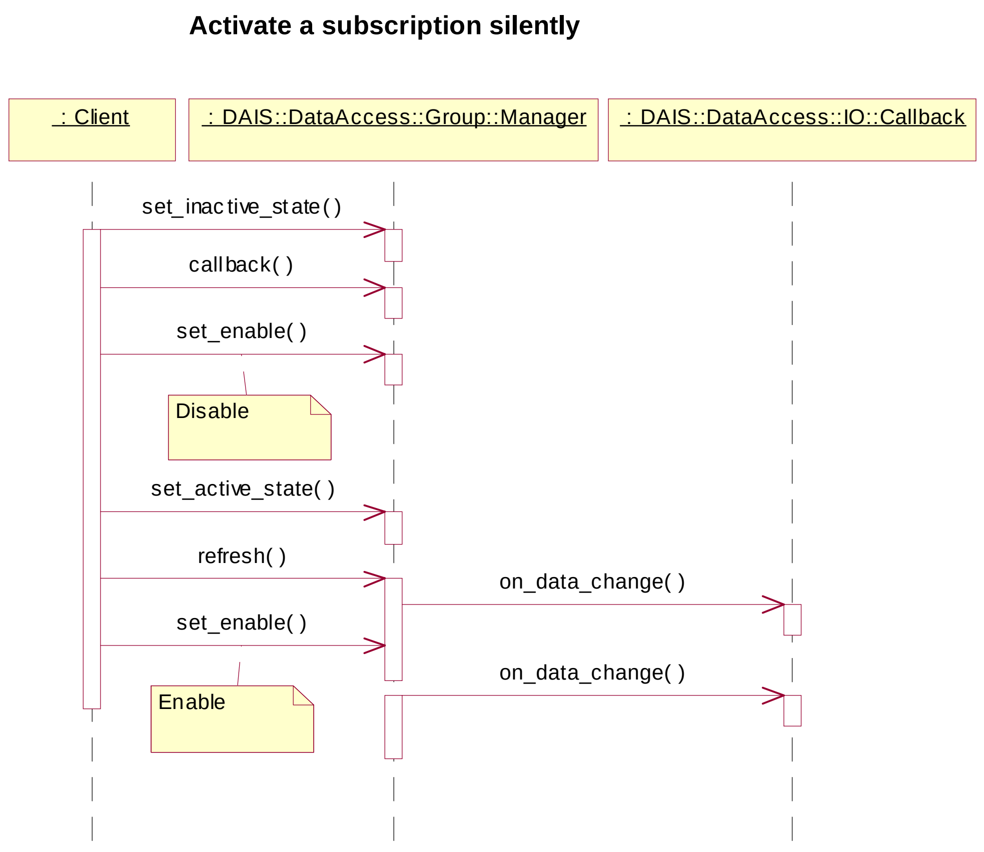


Figure 4-17 Activate a subscription silently interaction

Cancel

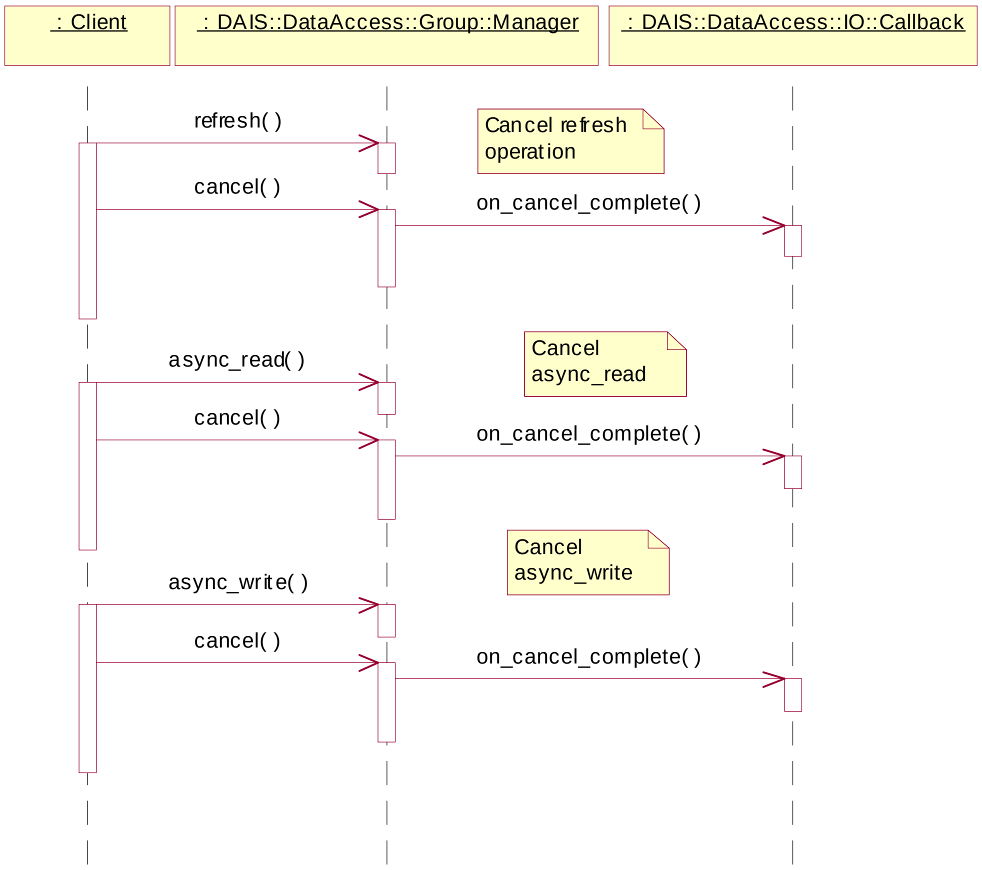


Figure 4-18 Cancellation interaction

Alarms & Events 5

Contents

This chapter contains the following sections.

Section Title	**Page**
"Information Model"	5-1
"API"	5-5

The alarms & events interface provides a client with a way to subscribe for alarms and events generated within RTUs, devices, or any software. The server supports various filter functions so that the client can compose a filter matching its current interest in alarms and events. Once the client sets up a filter specification it has to supply the server a callback object used by the server to notify the client with generated alarms and events.

5.1 Information Model

Alarms and events are generated by a source represented by **Source**. A source might be a single measurement, a collection of measurements or some other object. A collection of measurements may represent a complex real world object like a generator or some control function for the generator. A source has a name used to identification. A source is associated with a type and has correspondingly a number of properties. If a DAIS server implements both data access and alarms & events, a client shall expect there is a mapping between sources and nodes. A client shall however not make assumptions on how the mapping is made and use the method **translate_to_item_ids()** to get the mapping from the server.

Sources are organized in areas represented by **Area**. An area typically represents area of responsibility concerning supervision and operation of sources. Areas may however also be used for other groupings of sources. Areas can be hierarchically structured to allow creation of hierarchically organized responsibilities. Multiple views of areas are supported.

Alarms and events are generated due to a reason. Reason represents reasons. Three main reasons are defined and must be implemented by a server

- simple reason describing events that does not have an explicitly modeled condition space.
- tracking reason describing events generated due to an operator action.
- condition reason describing events generated based on an explicitly modeled condition space.

For each main reason it is possible to define a number of sub-reasons. A server is free to implement any sub-reasons. Each type of source is expected to be associated with at least one main reason and one or more sub-reasons for each main reason. The main reasons are not expected to have an association with a source type. Sub-reasons however are all expected to have an associated source type.

A Condition reason is associated with one or more condition spaces. **ConditionSpace** represents a condition space. Depending on how limits are applied to the properties defined by the type associated with a condition reason it is possible to create a space consisting of different discrete conditions. Condition describes each discrete condition. A condition space will have a number of conditions defined for it. A Transition describes each possible transition between a pair of conditions. The alarms & event session does not however provide any methods for direct access of transitions. When a transition is traversed a condition event is generated.

A source may be associated with one or more condition spaces. Each such association has status information describing the current condition. The association and its data are called **SourceCondition**. The source condition is identified by its associated source and condition space.

The described classes are shown in Figure 5-1. The attributes are described later with the description of the interfaces.

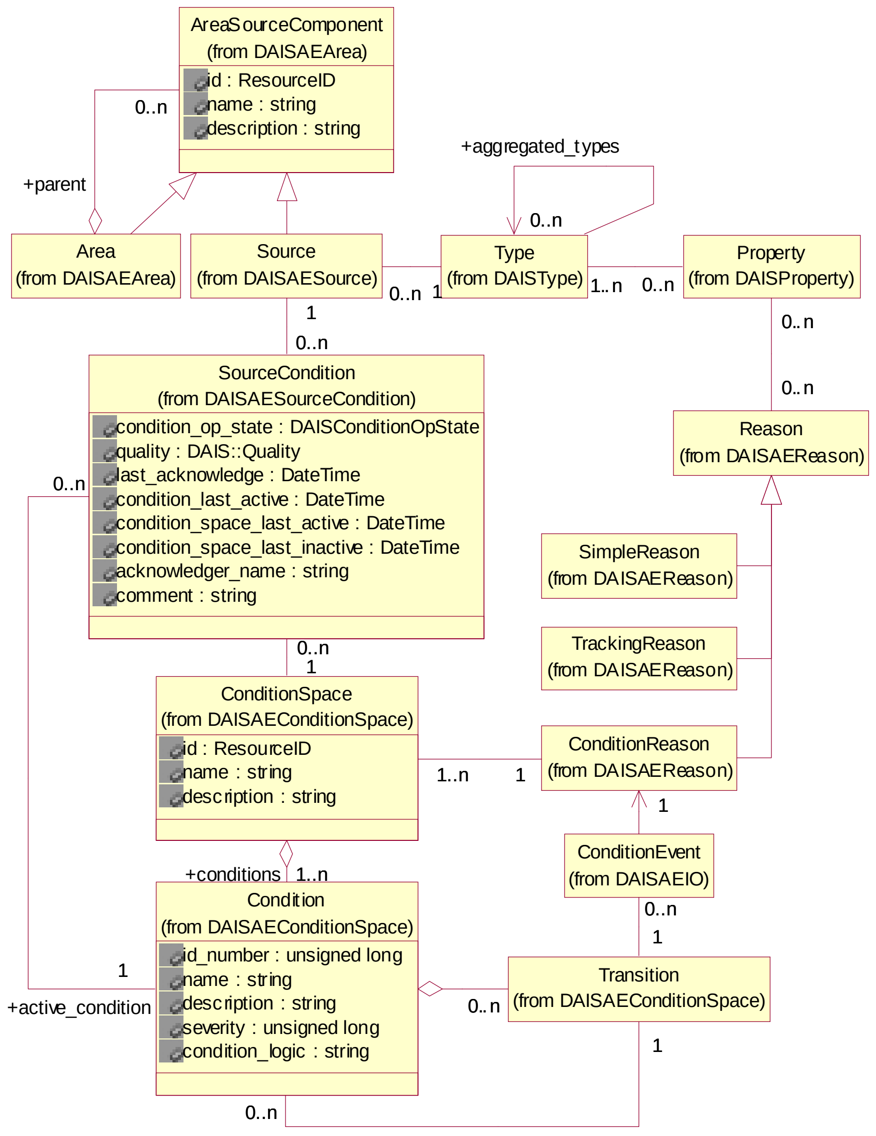


Figure 5-1 OPC recommended properties

5.1.1 OPC Recommended Properties

In OPC there is a recommendation of what properties are expected to be supported by an alarm & event server. The recommended properties are shown in Table 5-1. The column names correspond to the attributes in the Property class.

Table 5-1

label	**id**	**canonical_data_type**	**description**
Condition Status	300	STRING_TYPE	The current alarm or condition status associated with the Item (for example, “NORMAL,” “ACTIVE,” “HI ALARM”).
Alarm Quick Help	301	STRING_TYPE	A short text string providing a brief set of instructions for the operator to follow when this alarm occurs.
Alarm Area List	302	STRING_TYPE sequence	An array of stings indicating the plant or alarm areas that include this ItemID.
Primary Alarm Area	303	STRING_TYPE	A string indicating the primary plant or alarm area including this ItemID
Condition Logic	304	STRING_TYPE	An arbitrary string describing the test being performed (for example, “High Limit Exceeded” or “TAG.PV >= TAG.HILIM”).
Limit Exceeded	305	STRING_TYPE	For multistate alarms, the condition exceeded (for example, HIHI, HI, LO, LOLO).
Deadband	306	DOUBLE_TYPE	Deadband
HiHi Limit	307	DOUBLE_TYPE	HiHi Limit
Hi Limit	308	DOUBLE_TYPE	Hi Limit
Lo Limit	309	DOUBLE_TYPE	Lo Limit
LoLo Limit	310	DOUBLE_TYPE	LoLo Limit
Rate of Change Limit	311	DOUBLE_TYPE	Rate of Change Limit
Deviation Limit	312	DOUBLE_TYPE	Deviation Limit
	312-4999		Reserved by OPC

5.2 API

5.2.1 Alarms & Events IDL Overview

The dependencies among the different IDL files are shown in Figure 5-2.

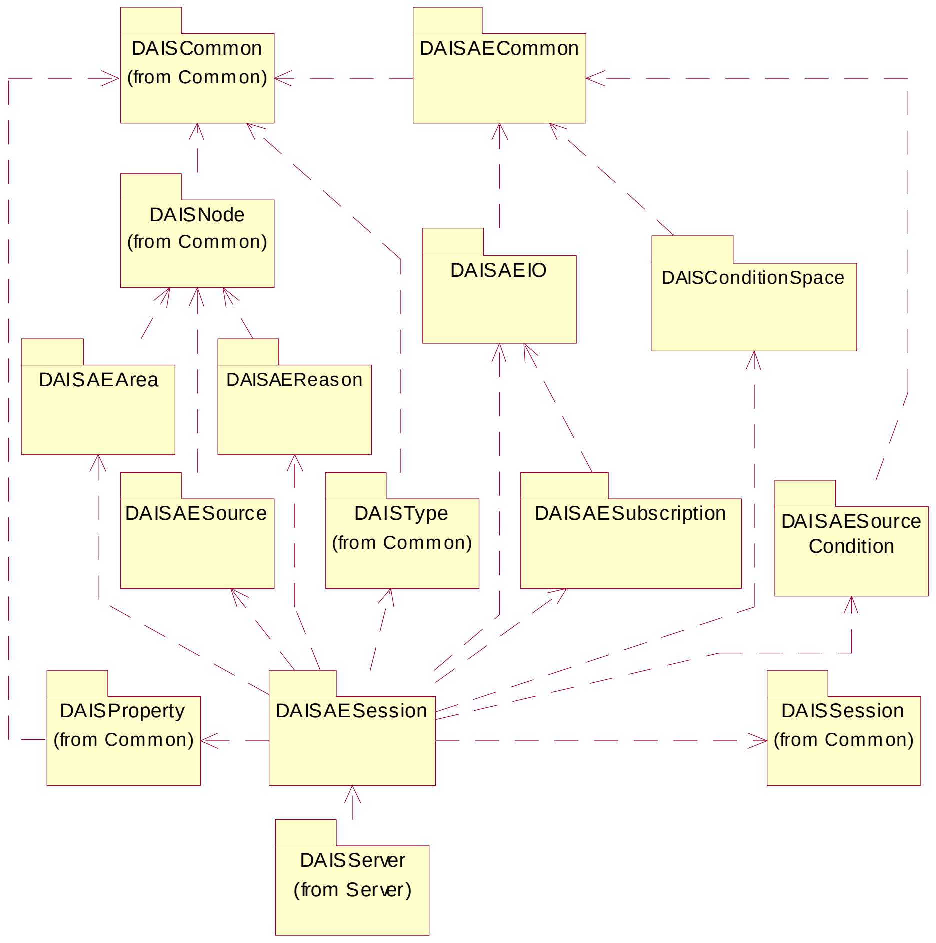


Figure 5-2 Dependencies between alarms and events IDL files

5.2.2 Alarms and Events Common IDL Definitions

5.2.2.1 IDL

```
module DAIS {
module AlarmsAndEvents {

typedef ResourceID          EventID;

typedef unsigned long SourceConditionOpState;
const            SourceConditionOpStateCONDITION_ENABLED= 0x0001;
const            SourceConditionOpStateCONDITION_ACTIVE= 0x0002;
const            SourceConditionOpStateCONDITION_ACKED= 0x0004;

};};
```

EventID

A **ResourceID** uniquely identifying an event notification.

SourceConditionOpState

Flag word holding for the operational state of a **SourceCondition**. The definitions of the state variable in the flag word are;

Flag	Description
CONDITION_ENABLED	The Condition is enabled and supervision is active.
CONDITION_ACTIVE	The Condition is active; that is, the supervision has determined that a fault has activated the condition.
CONDITION_ACKED	The Condition alarm has been acknowledged.

The combinations of the state variables result in eight states. The valid **SourceCondition** operational states are:

State	Description
Disabled	Not supervised by server.
Enabled, Inactive, Acked	Supervised by server, no fault detected and all alarms are acknowledged.

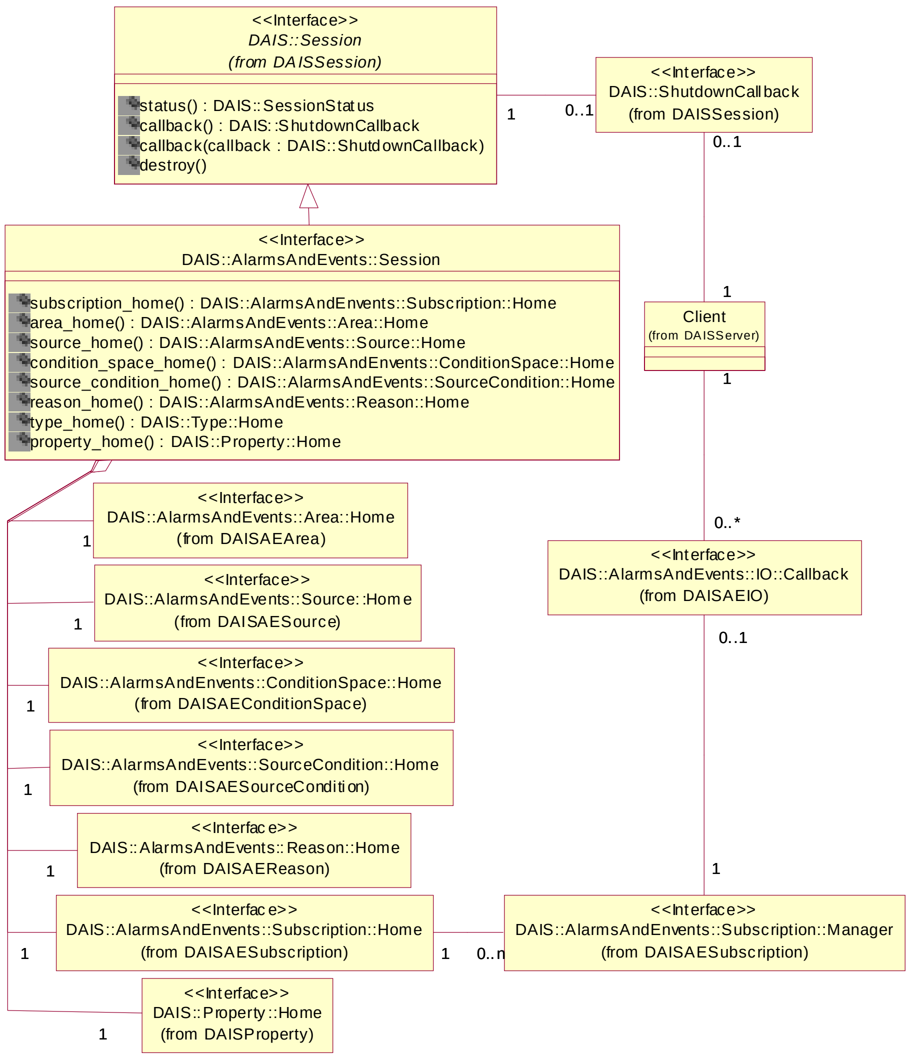
Enabled, Inactive, Unacked	Supervised by server, no fault detected and unacknowledged alarms exist.
Enabled, Active, Unacked	Supervised by server, a fault is detected and unacknowledged alarms exist.
Enabled, Active, Acked	Supervised by server, a fault is persistent and all alarms are acknowledged.

When enabled the state {Enabled, Inactive, Acked} is entered and from there the supervision will generate the appropriate state depending on the result of the supervision. Each state change results in sending an alarm and event notification. All notifications contain the state.

5.2.3 DAISAESession IDL

5.2.3.1 DAIS::AlarmsAndEvents::Session Objects overview

The **DAIS::AlarmsAndEvents::Session** object implements the alarms & events service on a per client basis. An alarm & event session object has a number of services provided by one singleton home object each. Each home object provides methods for manipulation of the data of the specific type they provide.



<<Interface>>
DAIS::Session
(from DAISSession)
status() : DAIS::SessionStatus
callback() : DAIS::ShutdownCallback
callback(callback : DAIS::ShutdownCallback)
destroy()
1
0..1
<<Interface>>
DAIS::ShutdownCallback
(from DAISSession)
0..1
<<Interface>>
DAIS::AlarmsAndEvents::Session
subscription_home() : DAIS::AlarmsAndEnvents::Subscription::Home
area_home() : DAIS::AlarmsAndEvents::Area::Home
source_home() : DAIS::AlarmsAndEvents::Source::Home
condition_space_home() : DAIS::AlarmsAndEnvents::ConditionSpace::Home
source_condition_home() : DAIS::AlarmsAndEvents::SourceCondition::Home
reason_home() : DAIS::AlarmsAndEvents::Reason::Home
type_home() : DAIS::Type::Home
property_home() : DAIS::Property::Home
1
Client
(from DAISServer)
1
0..*
<<Interface>>
DAIS::AlarmsAndEvents::Area::Home
(from DAISAEArea)
1
<<Interface>>
DAIS::AlarmsAndEvents::IO::Callback
(from DAISAEIO)
<<Interface>>
DAIS::AlarmsAndEvents::Source::Home
(from DAISAESource)
1
0..1
<<Interface>>
DAIS::AlarmsAndEnvents::ConditionSpace::Home
(from DAISAEConditionSpace)
1
<<Interface>>
DAIS::AlarmsAndEvents::SourceCondition::Home
(from DAISAESourceCondition)
1
<<Interface>>
DAIS::AlarmsAndEvents::Reason::Home
(from DAISAEReason)
1
1
<<Interface>>
DAIS::AlarmsAndEvents::Subscription::Home
(from DAISAESubscription)
1
1
0..n
<<Interface>>
DAIS::AlarmsAndEvents::Subscription::Manager
(from DAISAESubscription)
<<Interface>>
DAIS::Property::Home
(from DAISProperty)
1

Figure 5-3 DAIS alarms and events session IDL in UML

5.2.3.2 IDL

```
module DAIS {
module AlarmsAndEvents {

interface Session : DAIS::Session
{
    readonly attribute Subscription::Homesubscription_home;

    readonly attribute Area::Homearea_home;
    readonly attribute Source::Homesource_home;

    readonly attribute ConditionSpace::Homecondition_space_home;

    readonly attribute SourceCondition::Homesource_condition_home;

    readonly attribute Reason::Homereason_home;

    readonly attribute Type::Hometype_home;

    readonly attribute Property::Homeproperty_home;
};};};
```

Session

Session is an object implementing the alarms & events functions. It inherits common functionality as shut down callbacks and session status from **DAIS::AlarmsAndEvents::Session.**

subscription_home

A read only attribute holding a reference to a singleton **Subscription::Home** object.

area_home

A read only attribute holding a reference to a singleton **Area::Home** object.

source_home

A read only attribute holding a reference to a singleton **Source::Home** object.

condition_space_home

A read only attribute holding a reference to a singleton **ConditionSpace::Home** object.

source_condition_home

A read only attribute holding a reference to a singleton **SourceCondition::Home** object.

reason_home

A read only attribute holding a reference to a singleton **Reason::Home** object.

type_home

A read only attribute holding a reference to a singleton **Type::Home** object.

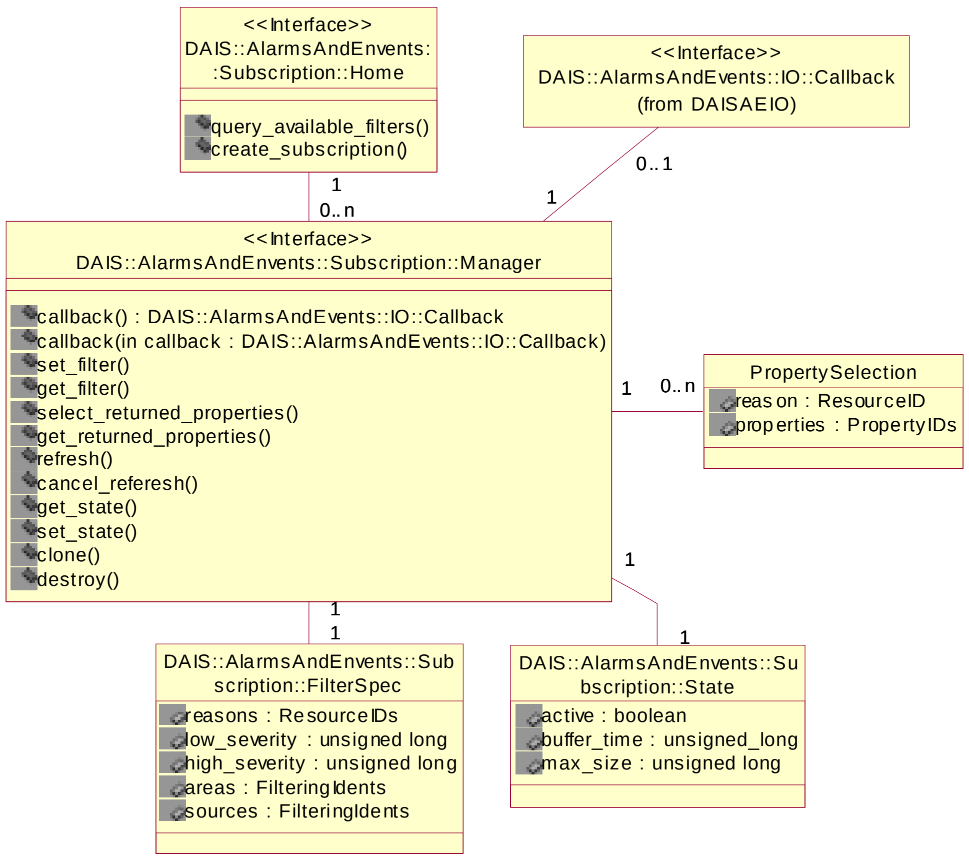
property_home

A read only attribute holding a reference to a singleton **Property::Home** object.

5.2.4 DAISAESubscription IDL

5.2.4.1 DAIS::AlarmsAndEvents::Subscription overview

A **DAIS::AlarmsAndEvents::Subscription::Manger** is an object holding a filter specification set up by a client. The filter is used to specify what notifications shall be sent to the client. A server can support various filter functions and a client can ask the **DAIS::AlarmsAndEvents::Subscription::Home** object what filter functions are supported. The subscription home is also used to create any number of subscription manager objects. Each subscription manager shall be associated with a client implemented callback object so that the server can send alarm and event notifications to the client.



<<Interface>>
DAIS::AlarmsAndEnvents::Subscription::Home
query_available_filters()
create_subscription()
<<Interface>>
DAIS::AlarmsAndEvents::IO::Callback
(from DAISAEIO)
1
0..n
0..1
1
<<Interface>>
DAIS::AlarmsAndEnvents::Subscription::Manager
callback() : DAIS::AlarmsAndEvents::IO::Callback
callback(in callback : DAIS::AlarmsAndEvents::IO::Callback)
set_filter()
get_filter()
select_returned_properties()
get_returned_properties()
refresh()
cancel_referesh()
get_state()
set_state()
clone()
destroy()
1
0..n
PropertySelection
reason : ResourceID
properties : PropertyIDs
1
1
1
1
DAIS::AlarmsAndEnvents::Subscription::FilterSpec
reasons : ResourceIDs
low_severity : unsigned long
high_severity : unsigned long
areas : FilteringIdents
sources : FilteringIdents
DAIS::AlarmsAndEnvents::Subscription::State
active : boolean
buffer_time : unsigned_long
max_size : unsigned long

Figure 5-4 DAIS alarms and events subscription IDL in UML

5.2.4.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module Subscription {

struct State {
    boolean             active;
    unsigned long       buffer_time;
    unsigned long       max_size;
};

typedef short FilteringIdentType;
const FilteringIdentType FILTERID_RESOURCE_TYPE= 1;
const FilteringIdentType FILTERID_STRING_TYPE= 2;

union FilteringIdent switch(FilteringIdentType) {
    case FILTERID_RESOURCE_TYPE  : ResourceID resource_value;
    case FILTERID_STRING_TYPE    : string string_value;
};
typedef sequence<FilteringIdent>FilteringIdents;

struct FilterSpec {
    ResourceIDs         reasons;
    unsigned long       low_severity;
    unsigned long       high_severity;
    FilteringIdents     areas;
    FilteringIdents     sources;
};

struct PropSelection {
    ResourceID              reason;
    PropertyIDs             properties;
};
typedef sequence<PropSelection> PropSelections;

interface Manager
{
    exception BusyDueToRefresh{string reason;};
    exception HistoryNotImplemented{string reason;};
    exception InvalidStateSpecified{string reason;};

    attribute IO::Callbackcallback;

    void set_filter (
        in  FilterSpec      filer_spec
    ) raises (BusyDueToRefresh);

    FilterSpec get_filter ();
    void select_returned_properties (
        in  PropSelectionsprop_selections
    );

    PropSelections get_returned_properties (
        in  ResourceIDs     reasons
    );

void refresh () raises (BusyDueToRefresh);

    void refresh_with_history (
        in   DateTime               start_time,
        in   DateTime               end_time
    ) raises (HistoryNotImplemented);

    void cancel_refresh ();

    State get_state ();

    State  set_state (
        in  State                   subscription_spec
    ) raises (InvalidStateSpecified);

    Manager clone ();

    void destroy ();
};

typedef unsigned longFilter;
const   Filter   FILTER_BY_MAIN_REASON    = 0x0001;
const   Filter   FILTER_BY_REASON         = 0x0002;
const   Filter   FILTER_BY_SEVERITY       = 0x0004;
const   Filter   FILTER_BY_AREA           = 0x0008;
const   Filter   FILTER_BY_SOURCE         = 0x0010;

interface Home
{
    exception InvalidStateSpecified{string reason;};

    Filter query_available_filters ();

    Manager create_subscription (
        in  Statesubscription_spec,
        out State   revised_subscr_spec
    ) raises (InvalidStateSpecified);

};
};};};
```

State

A struct describing the state for the subscription.

Member	Description
active	Indicate if the subscription is active; that is, is sending data on the call back interface. Events that are appearing in the server will be lost for clients connected to inactive subscriptions.
buffer_time	The requested buffer time. This is a minimum time - do not send event notifications any faster that this UNLESS **max_size** is greater than 0, in which case the server will send an event notification sooner to obey the **max_size** value. A value of 0 for **buffer_time** means that the server should send event notifications as soon as it gets them. This parameter along with the **max_size** parameter are used to improve communications efficiency between client and server. This parameter is a recommendation from the client, and the server is allowed to ignore the parameter. The server will return the buffer time it is actually providing in **revised_buffer_time**.
max_size	The requested maximum number of events that will be sent in a single callback. A value of 0 means that there is no limit to the number of events that will be sent in a single callback. Note that a value of **max_size** greater than 0, may cause the server to make callbacks more frequently than specified by **buffer_time** when a large number of events are being generated in order to limit the number of events to the **max_size**. This parameter is a recommendation from the client and the server is allowed to ignore this parameter. The server will return the actual number of events it is actually providing in **revised_max_size**.

FilteringIdentType

A short used to define the constants for the types allowed in the **FilteringIdent** union below.

FilteringIdent

A union holding the identification or the path-name identifying an area or a source included in the **FilterSpec** below. A union instead of a struct is used to avoid to possibility to specify both identification and path-name.

Member	Description
id	A resourceID.
path	The path-name string for the object. A server may allow to use a wild card syntax in the path name. The wildcard syntax is server specific and outside the scope of this specification.

FilterSpec

A struct holding the specification for how the server shall filter notifications sent to the client. The struct is used to specify the filtering a client wants a server to do for it.

Member	Description
reasons	A sequence of **ResourceID**s for the reasons for which the client wants to get notifications. Observe that specifying a main reason (for example, SimpleReason, TrackingReason, ConditionReason) will result in notifications for all sub-reasons specified for that main reason.
low_severity	The client wants all events with greater or equal severity.
high_severity	The client wants all events with less or equal severity.
areas	A sequence of **ResourceID**s for the areas that shall be notified. All sources below an area node will be notified.
sources	A sequence of **ResourceID**s for sources that shall be notified. An empty sequence mean all sources shall be notified.

PropSelection

A struct identifying what properties shall be included in an event notification for a specific reason.

Member	Description
reason	The reason for which the property selection is made.
properties	A sequence of **PropertyID**s for the selected properties.

Manager

An object managing subscription and filtering of event notifications per client.

BusyDueToRefresh

An exception telling a refresh is already going on.

HistoryNotImplemented

An exception telling the server does not record the event history.

callback

An attribute holding a reference to the callback object. The client is required to update the attribute with a by the client implemented callback object to get event notifications from the server. The attribute corresponds to the **IConnectionPoint** interface in OPC.

set_filter()

The method updates the filter specification. A try to change the filter spec during an ongoing refresh will give an exception. The method corresponds to **IOPCEventSubscriptionMgt::SetFilter()**.

Member	Description
filer_spec	The filter specification.
return	none.

get_filter()

The method gets the filter specification. The method corresponds to **IOPCEventSubscriptionMgt::GetFilter()**.

Member	Description
return	The filter specification.

select_returned_properties()

The method sets what properties shall be included in a notification specified per reason. The method corresponds to **IOPCEventSubscriptionMgt::SelectReturnedAttributes()**.

Member	Description
prop_selections	A sequence specifying what properties to include per reason.
return	none.

get_returned_properties()

The method gets what properties currently are included in a notification specified per reason. The method corresponds to **IOPCEventSubscriptionMgt::GetReturnedAttributes()**..

Member	Description
return	A sequence specifying what properties currently are included per reason.

refresh()

The method forces notifications for all currently active source conditions. The notifications will not include the history but only the current source condition operational state. The method corresponds to **IOPCEventSubscriptionMgt::Refresh()**.

Member	Description
return	none.

refresh_with_history()

The method forces notifications for all alarms and events recorded by the server. If the server does not support recording of alarms and events, an exception is raised. No corresponding method exists in OPC.

Member	Description
start_time	The time where to start the refresh. An unspecified time (=0) will make the refresh to start from the oldest recorded event.
end_time	The time where to stop the refresh. An unspecified time will (=0) will make the refresh to continue until the latest recorded event is reported.
return	none.

cancel_refresh()

The method cancels an ongoing refresh operation and corresponds to **IOPCEventSubscriptionMgt::CancelRefresh()**.

Member	Description
return	none.

get_state()

The method gets the current subscription state and corresponds to **IOPCEventSubscriptionMgt::GetState()**.

Member	Description
return	The current subscription state.

set_state()

The method sets the current subscription state and corresponds to **IOPCEventSubscriptionMgt::SetState()**. The actual parameters set are returned and an exception is raised if any input parameters are not accepted and hence changed.

Member	Description
subscription_spec	New state wanted for the subscription.
return	The accepted subscription state.

clone()

The method creates a copy of the subscription.

Member	Description
return	The copied subscription.

destroy()

The method destroys the subscription object.

Filter

A flag word holding flags specifying the filters that are implemented by the server.

Flag	Description
FILTER_BY_MAIN_REASON	Filtering by main reason is supported.
FILTER_BY_REASON	Filtering by sub-reason is supported.
FILTER_BY_SEVERITY	Filtering by severity is supported.
FILTER_BY_AREA	Filtering by area is supported.
FILTER_BY_SOURCE	Filtering by source is supported.

Home

A factory object for creation of subscription management objects.

query_available_filters()

Get by the server implemented filter functions. The method corresponds to **IOPCEventServer::QuearyAvailableFilters()**.

Member	Description
return	Filter specification flagword.

create_subscription()

Create a subscription management object. An invalid state will give an exception.

Member	Description
subscription_spec	State wanted for the subscription.
revised_subscr_spec	State accepted by the server.
return	The new subscription manager.

Set up Subscription

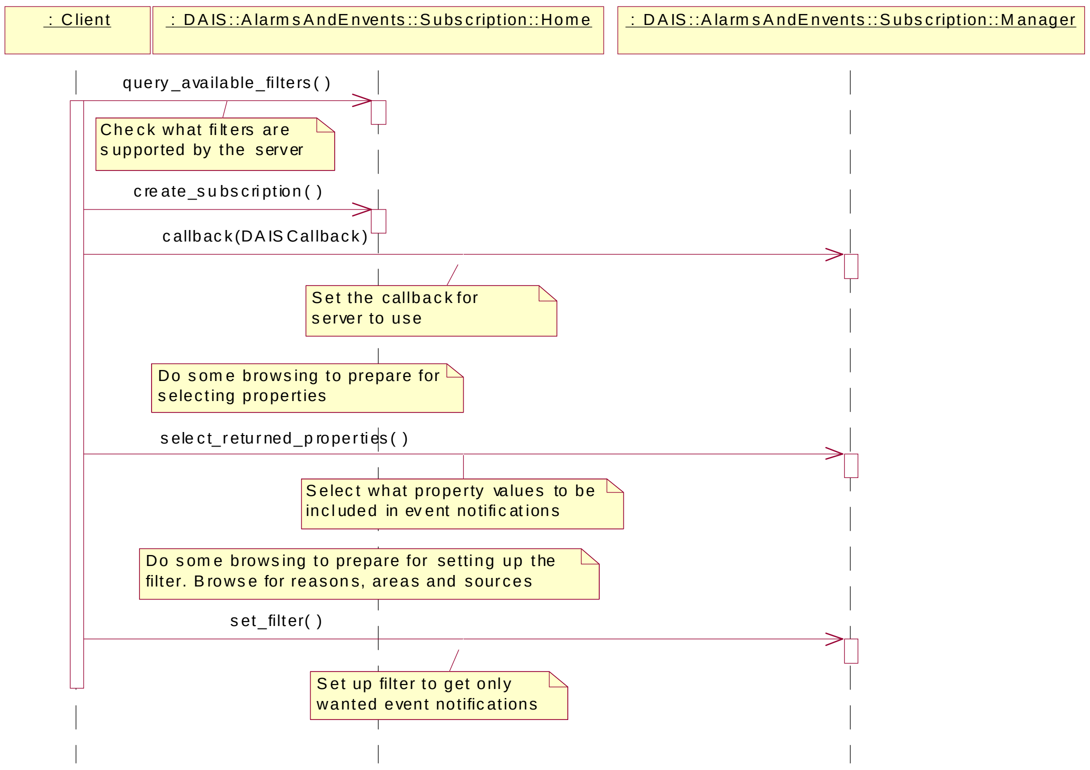

: Client
: DAIS::AlarmsAndEnvents::Subscription::Home
: DAIS::AlarmsAndEnvents::Subscription::Manager
query_available_filters( )
Check what filters are supported by the server
create_subscription( )
callback(DAISCallback)
Set the callback for server to use
Do some browsing to prepare for selecting properties
select_returned_properties( )
Select what property values to be included in event notifications
Do some browsing to prepare for setting up the filter. Browse for reasons, areas and sources
set_filter( )
Set up filter to get only wanted event notifications

Figure 5-5 Set up subscription interaction

Refresh

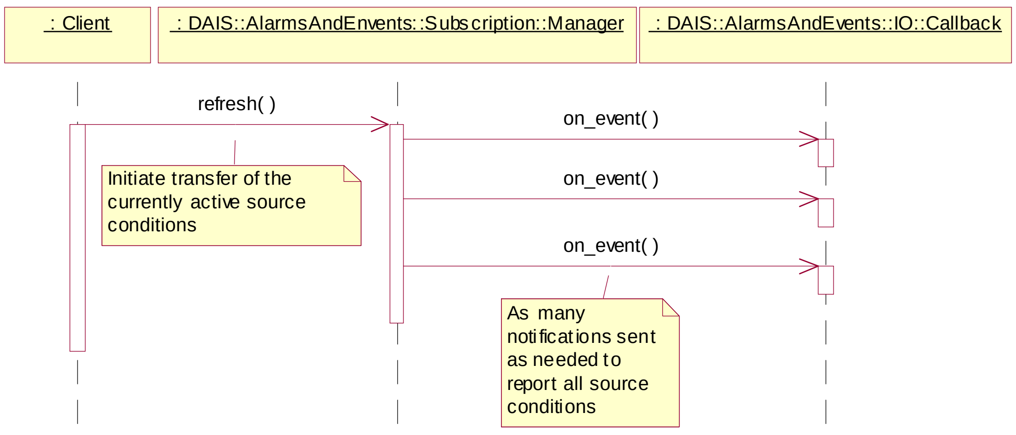

: Client
: DAIS::AlarmsAndEnvents::Subscription::Manager
: DAIS::AlarmsAndEvents::IO::Callback
refresh( )
Initiate transfer of the currently active source conditions
on_event( )
on_event( )
on_event( )
As many notifications sent as needed to report all source conditions

Figure 5-6 Refresh interaction

5.2.5 DAISAEArea IDL

5.2.5.1 DAIS::AlarmsAndEvents::Area overview

An area is a specialization of a node and is a collection of other areas or sources. Areas are intended for arbitrary hierarchical structuring of sources for various usages (for example, areas for authority or responsibility).

DAIS::AlarmsAndEvents::Area::Home is used for browsing the area hierarchy. The find methods in the interface correspond to the **IOPCEventAreaBrowser** with the filter type parameter set to OPC_AREA.

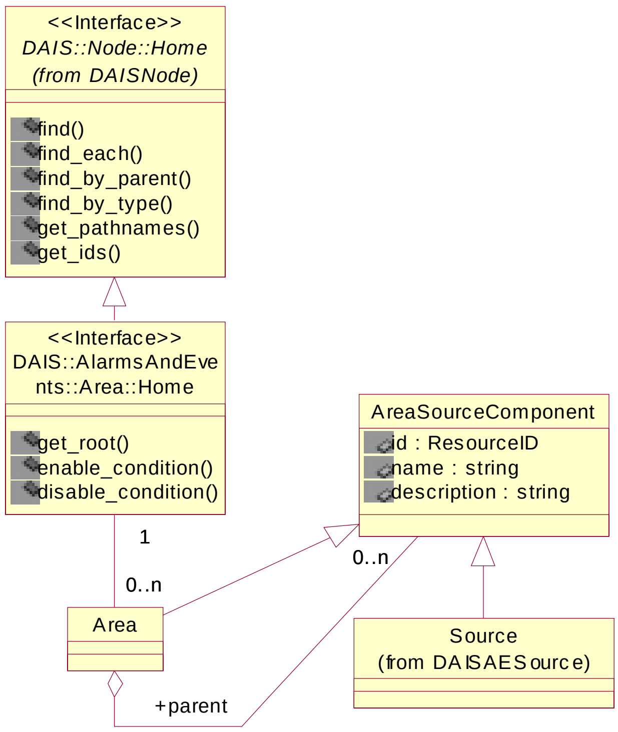


Figure 5-7 DAIS alarms and events area IDL in UML

5.2.5.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module Area {

interface Home : Node::Home
{
        ResourceID get_root();

        void enable_condition (
            in   ResourceIDs           areas
        );

        void disable_condition (
            in   ResourceIDs           areas
        );
    };
    };};};
```

Home

An object for browsing areas.

get_root()

A method to get the root node for the area tree.

Member	Description
return	The root area node.

enable_condition()

A method for enabling the sources contained by the specified areas. The corresponding OPC method is **IOPCEventServer::EnableConditionByArea()**.

Member	Description
areas	A sequence of area identifications.
return	none.

disable_condition()

A method for disabling the sources contained by the specified areas. The corresponding OPC method is **IOPCEventServer::DisableConditionByArea()**.

Member	Description
areas	A sequence of area identifications.
return	none

Browse areas

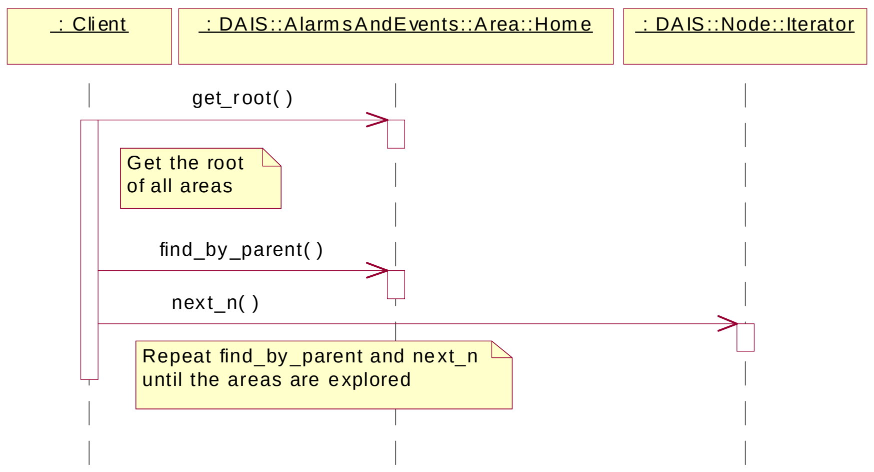

: Client
: DAIS::AlarmsAndEvents::Area::Home
: DAIS::Node::Iterator
get_root( )
Get the root of all areas
find_by_parent( )
next_n( )
Repeat find_by_parent and next_n until the areas are explored

Figure 5-8 Browse areas interaction

5.2.6 DAISAESource IDL

5.2.6.1 DAIS::AlarmsAndEvents::Source overview

A source is a specialization of a node and is contained by areas. A source represents an object that generates alarms and events. A source may have source conditions. The find methods in the interface correspond to the **IOPCEventAreaBrowser** with the filter type parameter set to OPC_SOURCE.

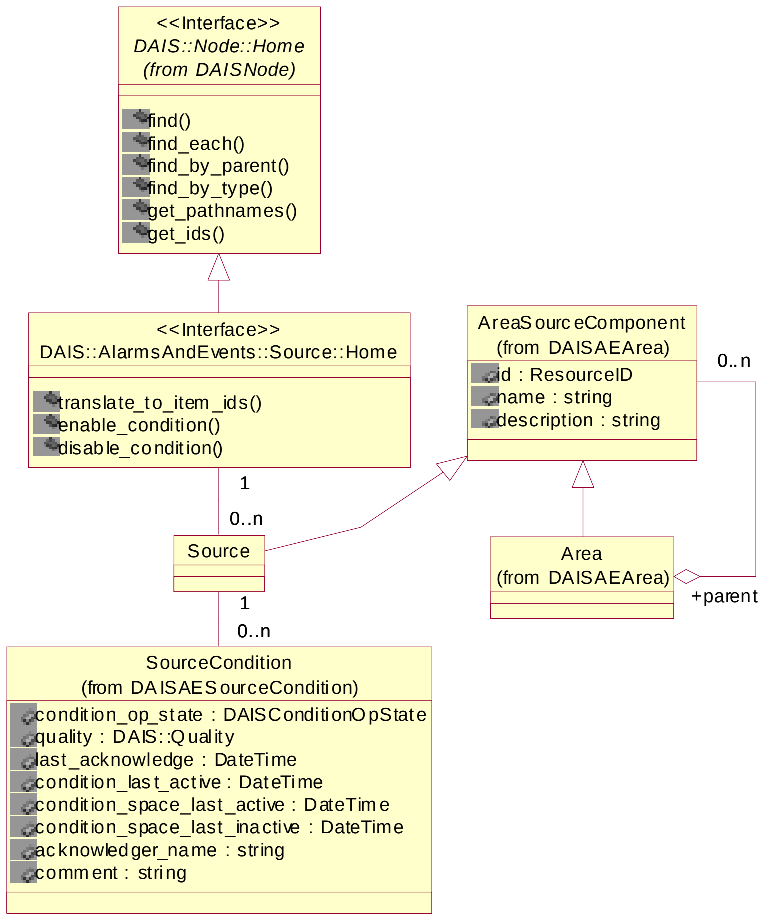


Figure 5-9 DAIS alarms and events source IDL in UML

5.2.6.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module Source {

interface Home : Node::Home
{
    exception PropertyDidNotTranslate{string reason;};

    ItemIDs translate_to_item_ids (
        in  ResourceID          source,
        in  ResourceID          reason,
        in  PropertyIDs       properties
    ) raises (PropertyDidNotTranslate);

    void enable_condition (
        in  ResourceIDs         sources
    );

    void disable_condition (
        in  ResourceIDs         sources
    );
};};};};
```

Home

An object for browsing sources at areas.

PropertyDidNotTranslate

An exception telling that one or more properties did not translate to **ItemID**s.

translate_to_item_ids()

A method for translation of information about a source to **ItemID**s for use with the data access interface. If one or more properties did not translate to **ItemID**s, an exception is raised. The corresponding OPC method is **IOPCEventServer::TranslateToItemIDs()**.

Member	**Description**
source	The identification of the source.
reason	The identification of the reason.
properties	A sequence of properties for which **ItemID**s are wanted.
return	A sequence of **ItemID**s. Properties that did not translate to **ItemID**s are returned as empty **ItemID**s.

enable_condition()

A method for enabling the specified sources. The corresponding OPC method is **IOPCEventServer::EnableConditionBySource()**.

Member	**Description**
sources	A sequence of area identifications.
return	none

disable_condition()

A method for disabling specified sources. The corresponding OPC method is **IOPCEventServer::DisableConditionBySource()**.

Member	Description
sources	A sequence of area identifications.
return	none

Browse sources

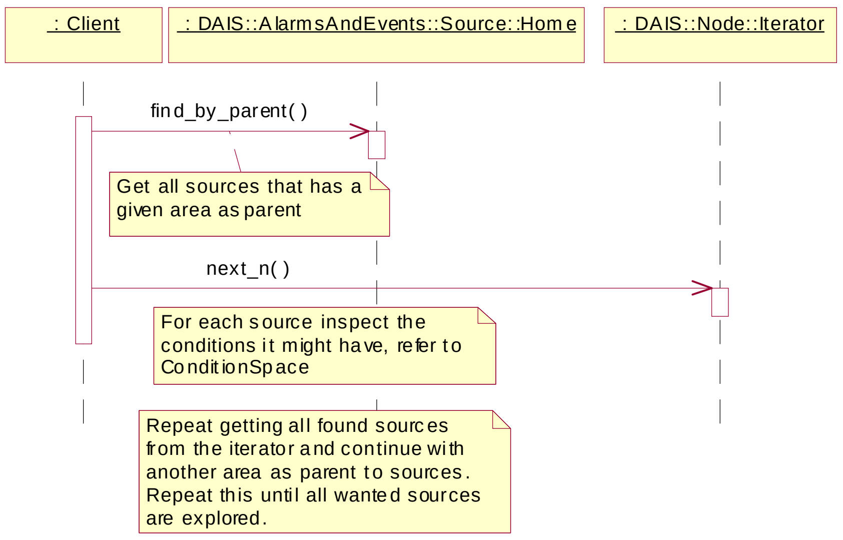

: Client
: DAIS::AlarmsAndEvents::Source::Home
: DAIS::Node::Iterator
find_by_parent( )
Get all sources that has a given area as parent
next_n( )
For each source inspect the conditions it might have, refer to ConditionSpace
Repeat getting all found sources from the iterator and continue with another area as parent to sources. Repeat this until all wanted sources are explored.

Figure 5-10 Browse sources interaction

5.2.7 DAISConditionSpace IDL

5.2.7.1 DAIS::AlarmsAndEvents::ConditionSpace overview

A condition space describes a set of conditions; that is, it is a space of conditions. Each condition space is associated with a particular sub-reason of the main reason ConditionReason. Each condition has logic describing when the condition is active. The logic is described in a little language having the following constructs

- arithmetic operators.
- logic operators.
- references to properties.

The referred properties must be included in the set of properties defined by the associated reason. The little language grammar is server specific and is outside the scope of this specification. Transitions describe what transitions between conditions are allowed. The interface does not support exploration of the transitions.

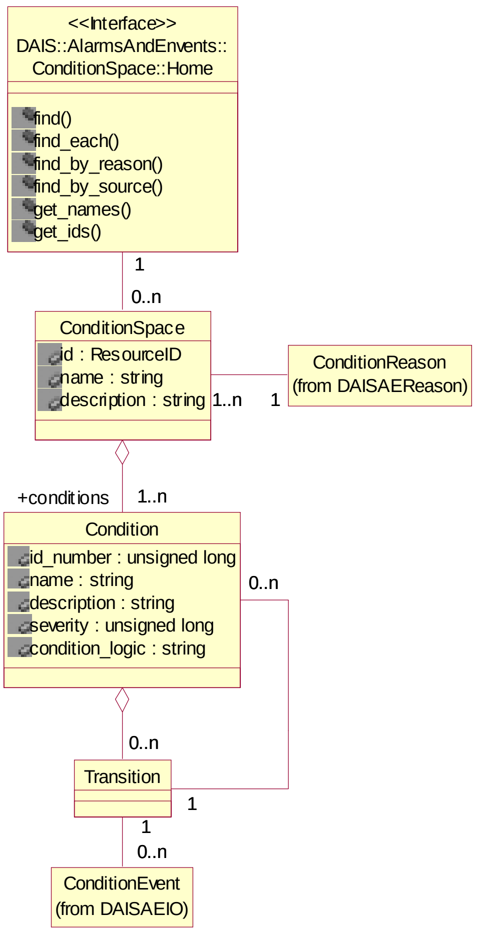


Figure 5-11 DAIS alarms and events condition space IDL in UML

5.2.7.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module ConditionSpace {

struct ConditionDescription {
    unsigned long        id_number;
    string                    name;
    string                    condition_logic;
    unsigned long        severity;
    string                          descrip;
};
typedef sequence<ConditionDescription> ConditionDescriptions;

struct Description
{
    ResourceID                  id;
    string                          name;
    string                          descrip;
    ConditionDescriptionsconditions;
};
typedef sequence< Description >Descriptions;

interface Home
{
    exception UnknownResourceID {string reason;};

    Description find (
        in   ResourceID         condition_space
    ) raises (UnknownResourceID);

    Descriptions find_each (
        in   ResourceIDs        condition_spaces
    ) raises (UnknownResourceID);

    Descriptions find_by_reason (
        in   ResourceID         reason
    ) raises (UnknownResourceID);

    Descriptions find_by_source (
        in   ResourceID         source
    ) raises (UnknownResourceID);

    Strings get_names (
        in   ResourceIDs        condition_spaces
    );

    ResourceIDs get_ids (
        in   Strings                names
    );

};};};};
```

ConditionDescription

A struct describing a condition.

Member	Description
id_number	A numeric identification unique within the condition space.
name	The name of the condition.
condition_logic	The logic telling when the condition is active.
severity	Severity is a number between 1 and 1000 having the following meaning: • Low severity 1-200 • Medium low severity 201-400 • Medium severity 401-600 • Medium high severity 601-800 • High severity 801-1000
description	A text that to be included in event notifications when the condition is active.

ConditionSpaceDescription

A struct describing the condition space.

Member	Description
id	A **ResourceID** identifying the condition space.
name	The name of the condition space.
description	A description of the condition space.
conditions	A sequence of the conditions creates the condition space. In OPC the conditions are called sub-conditions and are retrieved by the method **IOPCEventServer::QuerySubConditionNames()**.

Home

An object for browsing the condition spaces defined by a server.

find()

A method for getting the description of a condition space.

Parameter	Description
condition_space	A **ResourceID** identifying a condition space.
return	The condition space description.

find_each()

A method for getting the descriptions for a number of condition spaces.

Parameter	Description
condition_spaces	A sequence of **ResourceID** identifying condition spaces.
return	A sequence of condition space descriptions.

find_by_reason()

A method for finding all condition spaces defined for a reason. The corresponding OPC method is **IOPCEventServer::QueryConditionNames()**.

Parameter	Description
reason	A **ResourceID** identifying the reason for which to get the condition spaces.
return	A sequence of condition space descriptions.

find_by_source()

A method for finding all condition spaces defined for a source. The corresponding OPC method is **IOPCEventServer::QuerySourceConditions()**.

Parameter	Description
source	A **ResourceID** identifying the source for which to get the condition spaces.
return	A sequence of condition space descriptions.

get_names()

A method translating a number of condition space identifications into name strings.

Parameter	Description
condition_spaces	A sequence of **ResourceID** identifying condition spaces.

return	A sequence of condition space names. Non translated identifications are returned as empty strings.

get_ids()

A method translating a number of condition space names into **ResourceID**s.

Parameter	Description
names	A sequence of condition space names.
return	A sequence of condition space **ResourceID**s. Non translated identifications are returned as NULL IDs.

Browse condition space by source

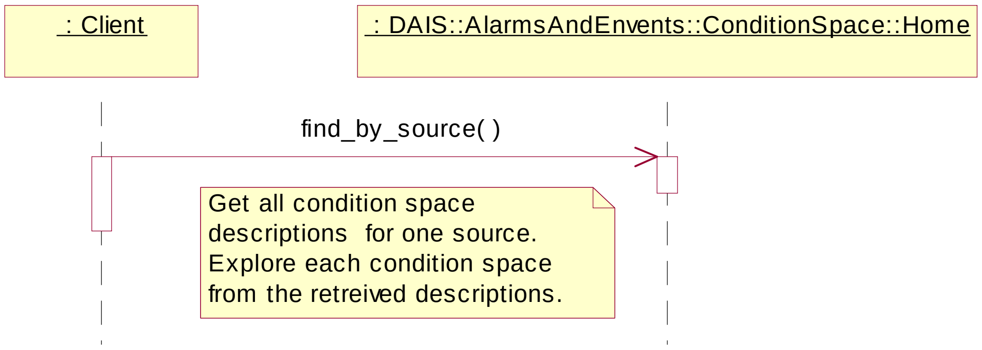

: Client
: DAIS::AlarmsAndEnvents::ConditionSpace::Home
find_by_source( )
Get all condition space descriptions for one source. Explore each condition space from the retreived descriptions.

Figure 5-12 Browse condition space by source interaction

Browse condition space by reason

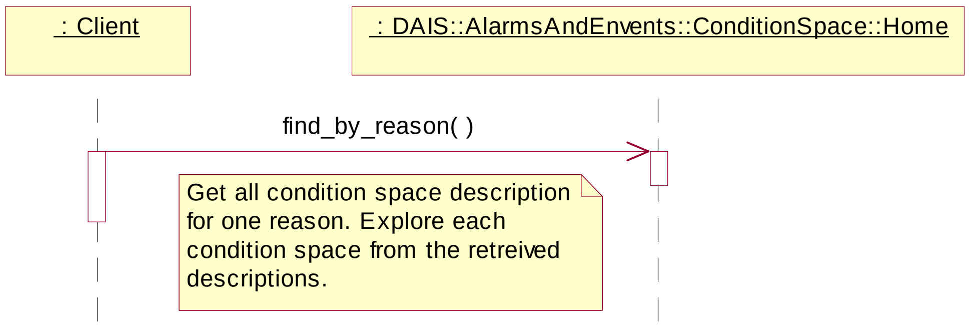
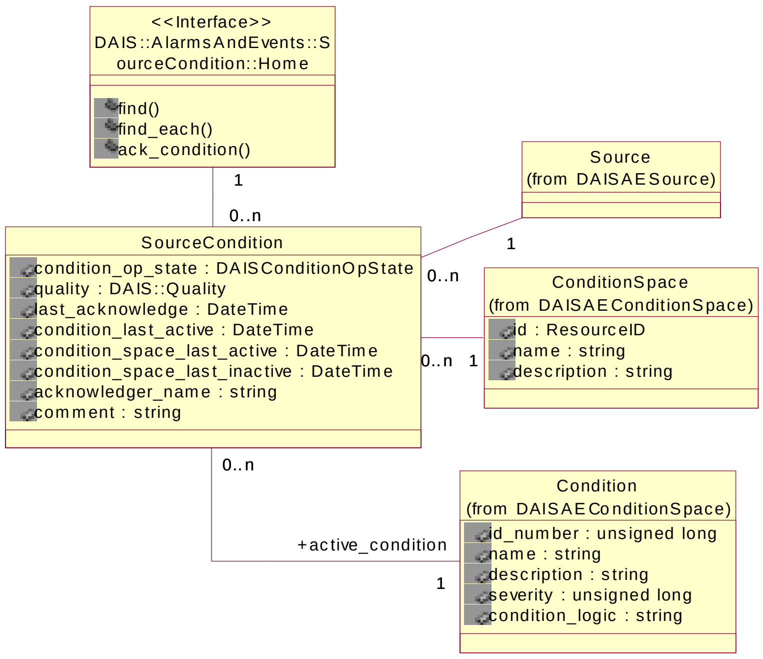

: Client
: DAIS::AlarmsAndEnvents::ConditionSpace::Home
find_by_reason( )
Get all condition space description for one reason. Explore each condition space from the retreived descriptions.

Figure 5-13 Browse condition space by reason interaction

5.2.8 DAISAESourceCondition IDL

5.2.8.1 SourceCondition overview

A source condition associates a source with a condition space. A source condition holds the current information specific to a source using a particular condition space for the supervision. A source condition has no own identification and is identified by its associated source and condition space. Information about a source condition is accessed through its home object. In OPC a source condition is sometimes called an instance of a condition.



<<Interface>>
DAIS::AlarmsAndEvents::SourceCondition::Home
find()
find_each()
ack_condition()
1
0..n
SourceCondition
condition_op_state : DAISConditionOpState
quality : DAIS::Quality
last_acknowledge : DateTime
condition_last_active : DateTime
condition_space_last_active : DateTime
condition_space_last_inactive : DateTime
acknowledger_name : string
comment : string
Source
(from DAISAESource)
1
0..n
ConditionSpace
(from DAISAEConditionSpace)
id : ResourceID
name : string
description : string
0..n
1
0..n
+active_condition
1
Condition
(from DAISAEConditionSpace)
id_number : unsigned long
name : string
description : string
severity : unsigned long
condition_logic : string

Figure 5-14 DAIS alarms and events source condition IDL in UML

5.2.8.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module SourceCondition {

struct Id {
      ResourceID                  source;
      ResourceID                  condition_space;
};
typedef sequence<Id>Ids;
```

```
struct Description
{
    Id                        source_condition;
    SourceConditionOpState    source_condition_op_state;
    unsigned long             active_condition;
    string                    ac_logic;
    unsigned long             ac_severity;
    string                    ac_description;
    Quality                   dais_quality;
    DateTime                  last_acknowledge;
    DateTime                  condition_last_active;
    DateTime                  condition_space_last_active;
    DateTime                  condition_space_last_inactive;
    string                    acknowledger_name;
    string                    comment;
    PropertyValues            property_values;
};
typedef sequence< Description > Descriptions;


struct AcknowledgeSpec {
    Id                             source_condition;
    DateTime                       active_time;
    EventID                        cookie;
};
typedef sequence<AcknowledgeSpec>AcknowledgeSpecs;

interface Iterator
{
    boolean next_n (
        in      unsigned longn,
        out     Descriptionsc_descriptions
    );
    void destroy();
};

interface Home
{
    exception UnknownId {string reason;};
    exception UnknownPropertyID {string reason;};

    Description find (
        in  Id              source_condition,
        in  PropertyIDs     properties
    ) raises (UnknownId, UnknownPropertyID);

    Iterator find_each(
        in  Ids             source_conditions,
        in  PropertyIDs     properties
    ) raises (UnknownId, UnknownPropertyID);
```

```
        Descriptions ack_condition (
            in  string                  acknowledger_name,
            in  string                  comment,
            in  AcknowledgeSpecs        ack_spec
        );
    };
};};};
```

Id

A struct that identify a source condition.

Member	Description
source	The **ResourceID** identifying the associated source.
condition_space	The **ResourceID** identifying the associated condition space.

Description

A struct describing the source condition.

Member	Description
id	The Id identifying the source condition.
source_condition_op_state	The **DAISSourceConditionOpState** as described in Section 5.2.2, “Alarms and Events Common IDL Definitions,” on page 5-6.
active_condition	The identification number of the currently active condition.
ac_logic	The condition logic from the active condition.
ac_severity	The severity from the active condition.
ac_description	The description from the active condition.
dais_quality	The quality is evaluated from the qualities from the properties used to evaluate the condition logic.
last_acknowledge	The last time the condition was acknowledged.
condition_last_active	Time for the latest condition transition.

condition_space_last_active	The last time when the condition space became active. After this time more condition transitions may occur. The **condition_last_active** will then be later than **condition_space_last_active**.
condition_space_last_inactive	The last time when the condition space became inactive; that is, no conditions are active.
acknowledger_name	The name of the client making an acknowledge.
comment	A comment passed by the client making an acknowledge.
property_values	A sequence of property values as selected by the **Manager::select_returned_properties** call.

AcknowledgeSpec

A struct specifying an alarm to acknowledge.

Member	Description
source_condition	The Id identifying the source condition for which to acknowledge an alarm.
active_time	The time when the alarm activated.
cookie	A identification of the alarm.

Iterator

A standard iterator refer to Section 3.1.5, “Iterator Methods IDL,” on page 3-10.

Home

An object for browsing and accessing source conditions.

UnknownDAISSourceConditionID

An exception telling that the source condition identification was not recognized.

UnknownPropertyID

An exception telling that a property identification was not recognized.

find()

A method for getting the description of a source condition. The corresponding OPC method is **IOPCEventServer::GetConditionState()**.

Parameter	Description
source_condition	A ResourceID identifying a source condition.
return	The source condition description.

find_each()

A method for getting the descriptions for a number of source conditions.

Parameter	Description
condition_spaces	A sequence identifying source conditions.
return	A sequence of source condition descriptions.

ack_condition()

A method to acknowledge a number of source condition alarms. The corresponding OPC method is **IOPCEventServer::AckCondition()**.

Parameter	Description
acknowledger_name	The name of the client making the acknowledge.
comment	A comment to be added to source condition and the event.
ack_spec	A sequence specifying the alarms to acknowledge.
return	A sequence containing descriptions for the acknowledged source conditions.

Inspect a specific source condition

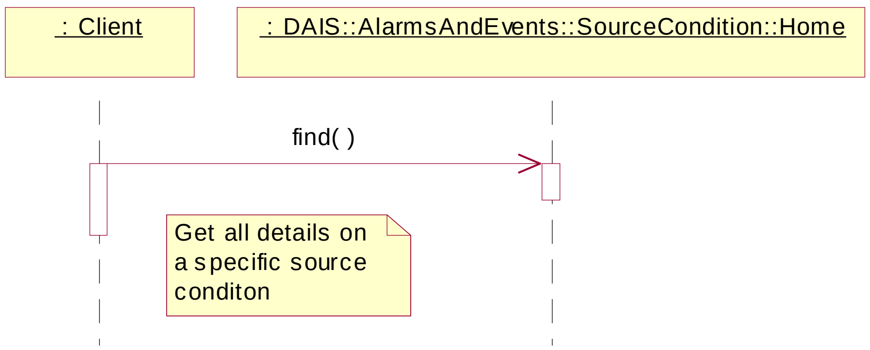

: Client
: DAIS::AlarmsAndEvents::SourceCondition::Home
find( )
Get all details on a specific source conditon

Figure 5-15 Inspect a specific source condition interaction

Acknowledge alarm

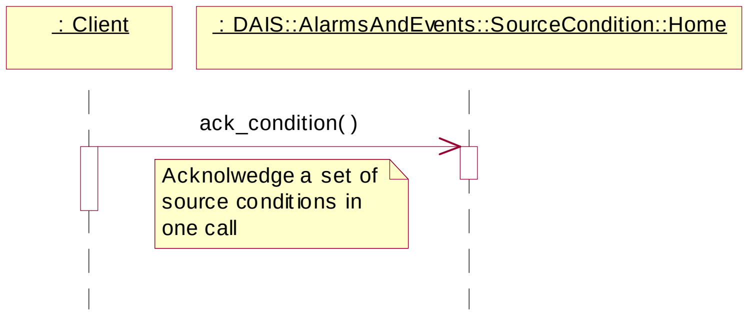

: Client
: DAIS::AlarmsAndEvents::SourceCondition::Home
ack_condition( )
Acknolwedge a set of source conditions in one call

Figure 5-16 Acknowledge alarm interaction

5.2.9 *DAISReason IDL*

5.2.9.1 *DAIS::AlarmsAndEvents::Reason overview*

These are definitions for manipulating reasons. A reason describe what cause an event. Reasons are organized in a two level hierarchy where the first level has the three mandatory main reasons; simple reason, condition reason, and tracking reason.

Simple reason does not have a condition space (refer to ConditionSpace). Generation of simple reason alarms or events is coded into some software function and cannot be configured. Typical simple reasons are program errors, hardware device failures.

Sources of different types (for example, breaker position, breaker current, generator active power) will have different reasons why an alarm or event is generated (for example, breaker trip, breaker over current, generator active power generation overload). Condition reasons are used to describe source type specific reasons. The alarm and event generation for a condition reason is configured using condition spaces and several condition spaces can be created for the same same condition reason. This correspond to variations in the way the alarms and events are generated using different condition logics (refer to Section 5.2.7, “DAISConditionSpace IDL,” on page 5-25).

Alarms or events due to operator actions or control functions (intended alarms or events rather than spontaneous) are recorded as tracking reasons.

A reason has a set of properties associated with it. Some or all of the properties may be included in event notifications. The properties comes from the source type and may be used by the logic generating an event notification.

The **get_main_reasons()** method is used to obtain the three main reasons. The **find_by_parent()** is used to get the reasons. Reasons at this second level correspond to the OPC Categories. The labels for the three main reasons are “DAIS_CONDITION_REASON,” “DAIS_TRACKING_REASON,” and “DAIS_SIMPLE_REASON".”

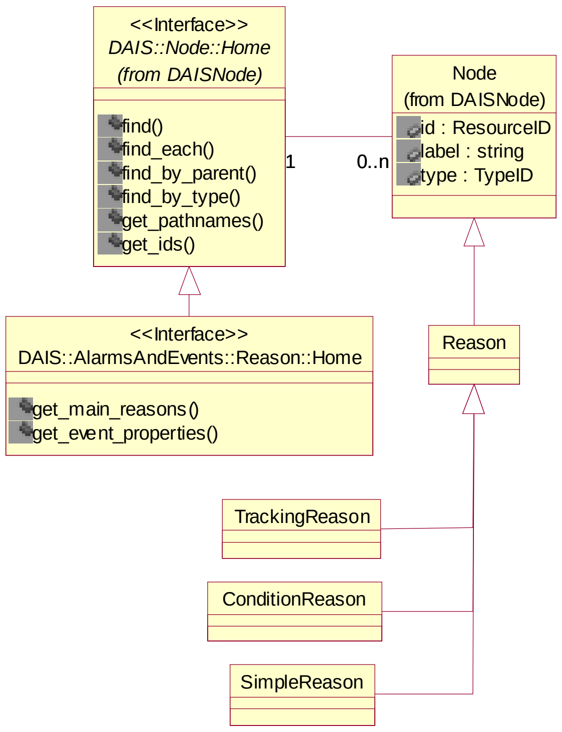


Figure 5-17 DAIS alarms and events reason IDL in UML

5.2.9.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module Reason {

interface Home : Node::Home
{
    Node::Descriptions get_main_reasons ();

    PropertyIDs get_event_properties (
        in   ResourceID                  reason
    );
};
};};};
```

Home

An object for browsing reasons. Most of the browsing functionality is inherited from **DAIS::Node**.

get_main_reasons()

Get the three main reasons; simple reason, condition reason, and tracking reason. The three main reasons appear as three different roots.

Parameter	Description
return	The descriptions for the three main reasons. The labels for them are “DAIS_CONDITION_REASON,” “DAIS_TRACKING_REASON,” and “DAIS_SIMPLE_REASON.”

get_event_properties()

Get all properties that are used in the supervision of the source type for a specific reason. The corresponding OPC method is **IOPCEventServer::QueryEventAttributes()**.

Parameter	Description
reason	The identification of the reason to get the properties for.
return	A sequence of property identifications.

Browse reasons

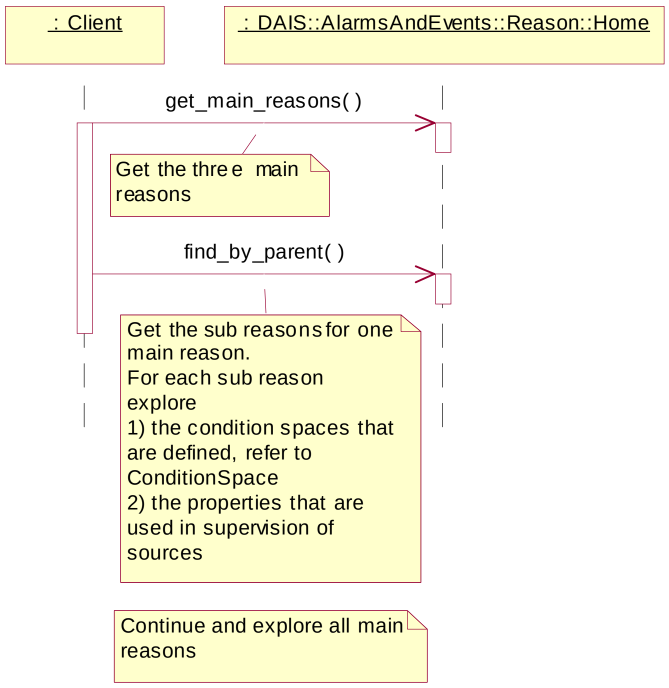

: Client
: DAIS::AlarmsAndEvents::Reason::Home
get_main_reasons( )
Get the three main reasons
find_by_parent( )
Get the sub reasons for one main reason. For each sub reason explore 1) the condition spaces that are defined, refer to ConditionSpace 2) the properties that are used in supervision of sources
Continue and explore all main reasons

Figure 5-18 Browse reason interaction

Browse properties

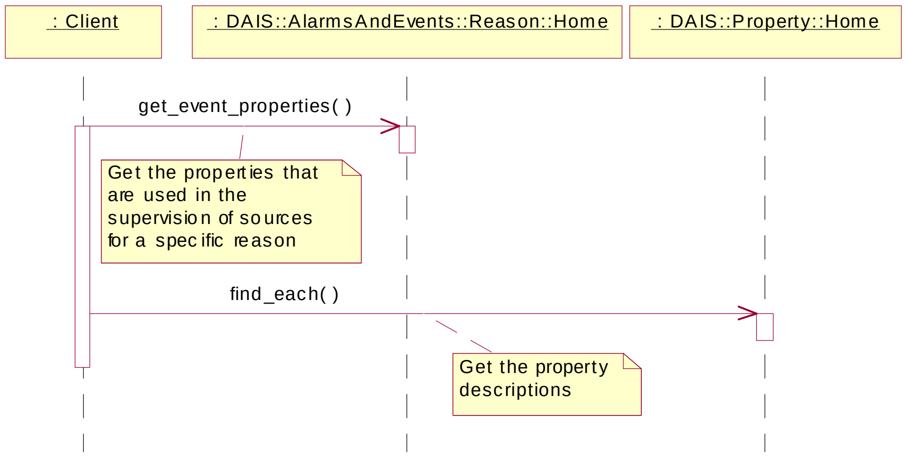

: Client
: DAIS::AlarmsAndEvents::Reason::Home
: DAIS::Property::Home
get_event_properties( )
Get the properties that are used in the supervision of sources for a specific reason
find_each( )
Get the property descriptions

Figure 5-19 Browse properties interaction

5.2.10 DAISAEIO IDL

5.2.10.1 DAIS::AlarmsAndEvents::IO overview

The IO interface for alarms & events does only support callbacks and the client shall implement callback.

The event notifications sent over the callback are of three different types corresponding to the three main reasons and are; simple event, condition event, and tracking event. All three event types have the common part Event. The tracking event has additional information on the operator and the condition event has additional information on the source condition causing the notification.

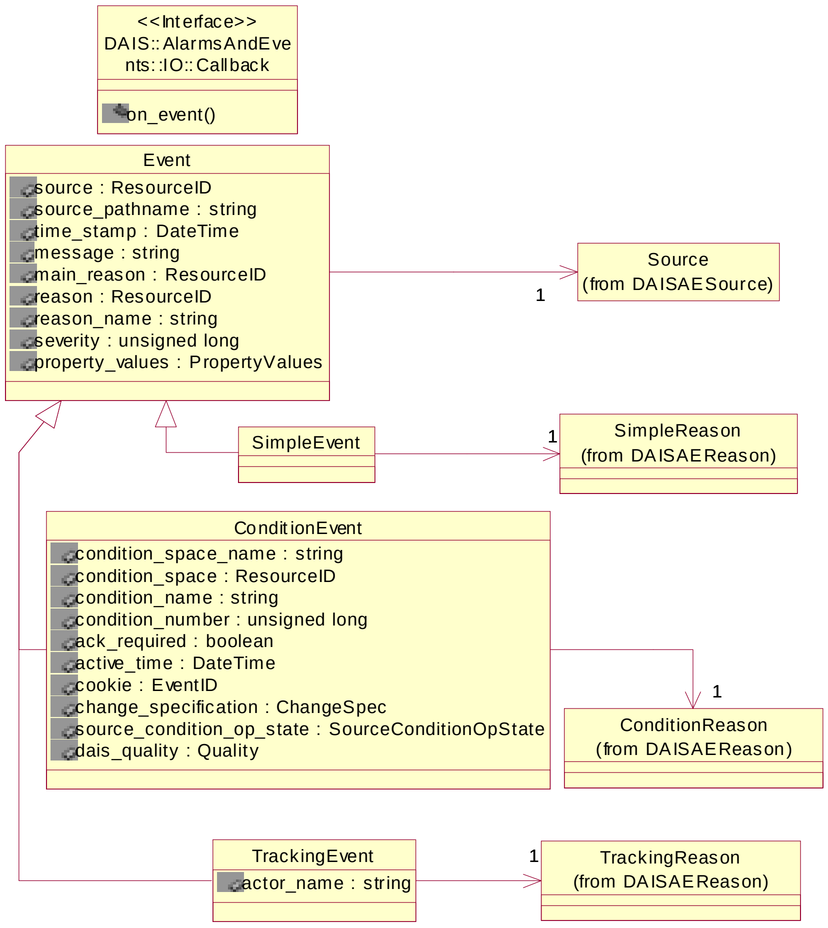


Figure 5-20 DAIS alarms and events IO IDL in UML

5.2.10.2 IDL

```
module DAIS {
module AlarmsAndEvents {
module IO {

typedef unsigned long ChangeSpec;
const ChangeSpec CHANGE_ACTIVE_STATE = 0x0001;
const ChangeSpec CHANGE_ACK_STATE      = 0x0002;
const ChangeSpec CHANGE_ENABLE_STATE      = 0x0004;
const ChangeSpec CHANGE_QUALITY              = 0x0008;
const ChangeSpec CHANGE_SEVERITY             = 0x0010;
const ChangeSpec CHANGE_CONDITION           = 0x0020;
const ChangeSpec CHANGE_MESSAGE             = 0x0040;
const ChangeSpec CHANGE_ATTRIBUTE           = 0x0080;

struct SimpleEvent {
    ResourceID                  source;
    string                      source_pathname;
    DateTime                    time_stamp;
    string                      message;
    ResourceID                  main_reason;
    ResourceID                  reason;
    string                      reason_name;
    unsigned long               severity;
    PropertyValues              property_values;
};

struct TrackingEvent {
    string                      actor_name;
};

struct ConditionEvent {
    string                      condition_space_name;
    ResourceID                  condition_space;
    string                      condition_name;
    unsigned long               condition_number;
    boolean                     ack_required;
    DateTime                    active_time;
    EventID                     cookie;
    ChangeSpec                  change_specification;
    SourceConditionOpState      source_condition_op_state;
    Quality                     dais_quality;
};

struct OnEvent {
    SimpleEvent             simple_event;
    TrackingEvent           tracking_event;
    ConditionEvent          condition_event;
};
typedef sequence<OnEvent>  OnEvents;

interface Callback
{
    void on_event (
        in  boolean          refresh,
        in  boolean          last_refresh,
        in  OnEvents         events
    );
};
};};};
```

ChangeSpec

A flag word having a number of flags telling what change caused the event notification.

Flag	Description
CHANGE_ACTIVE_STATE	Alarm active has changed.
CHANGE_ACK_STATE	Alarm acknowledge has changed.
CHANGE_ENABLE_STATE	Enable has changed.
CHANGE_QUALITY	The quality has changed.
CHANGE_SEVERITY	The severity has changed.
CHANGE_CONDITION	A condition has become active/inactive.
CHANGE_MESSAGE	The message has been updated.
CHANGE_ATTRIBUTE	An attribute value has changed.

SimpleEvent

A struct holding the simple event data.

Member	Description
source	The identification of the source for which the event notification was created.
source_pathname	The full pathname of the source.

time_stamp	Time of the event occurrence - for conditions, time that the condition transitioned into the new state or condition. For example, if the event notification is for acknowledgment of a condition, this would be the time that the condition became acknowledged.
message	Event notification message describing the event.
main_reason	The identification for one of the three main reasons.
reason	The identification of the sub-reason why the event notification was sent.
reason_name	The label for the sub-reason.
severity	The severity for the event, a number between 0 and 1000.
property_values	A sequence of property values as selected by the **select_returned_properties()** method.

TrackingEvent

A struct holding tracking event data.

Member	**Description**
actor_name	The name of the actor or operator causing the event notification.

ConditionEvent

A struct holding the condition event data.

Member	**Description**
condition_space_name	The name of the condition space that caused the event notification.
condition_space	Identification of the condition space.
condition_name	The name of the condition that caused the event notification.
condition_number	Identification of the condition.
ack_required	An indication that an alarm is generated and that an acknowledgment is required.
active_time	The time when the condition became active.
cookie	An identification of the event notification.

change_specification	Indicates to the client which properties of the condition have changed
source_condition_op_state	The new state for the source condition.
quality	The quality.

OnEvent

A struct composed of the three above event structs. The simple event struct always contains valid information and which of the tracking or condition event structs are valid is decided from the **main_reason** simple event member.

Member	Description
simple_event	The simple event.
tracking_event	The tracking event.
condition_event	The condition event.

Callback

The callback object implemented by the client and used by the server to send event notifications.

on_event()

The callback method.

Parameter	Description
refresh	Indicate if this callback is due to a refresh.
last_refresh	Indicate if this callback is the last in a sequence initiated by a refresh.
events	A sequence of event notifications.
return	None.

Event notification callbacks

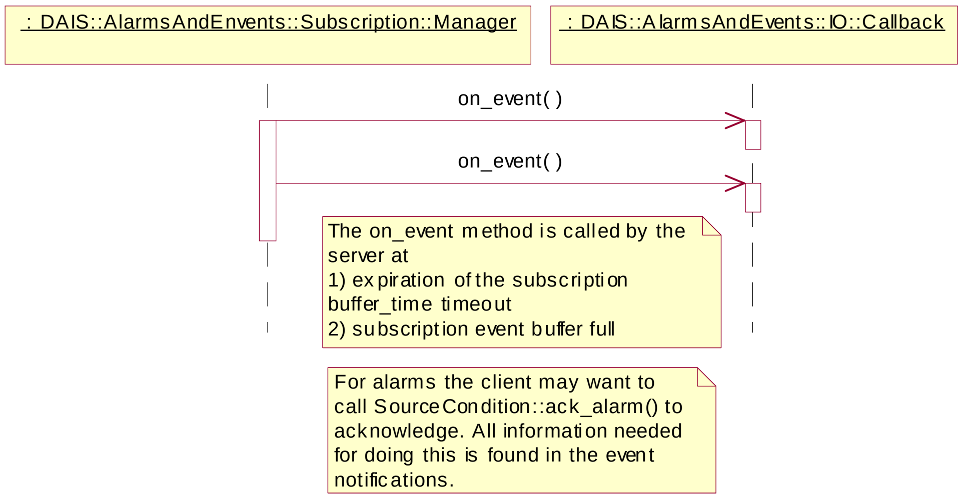

: DAIS::AlarmsAndEnvents::Subscription::Manager
: DAIS::AlarmsAndEvents::IO::Callback
on_event( )
on_event( )
The on_event method is called by the server at
1) expiration of the subscription buffer_time timeout
2) subscription event buffer full
For alarms the client may want to call SourceCondition::ack_alarm() to acknowledge. All information needed for doing this is found in the event notifications.

Figure 5-21 Event notification callbacks interaction

Refresh callbacks

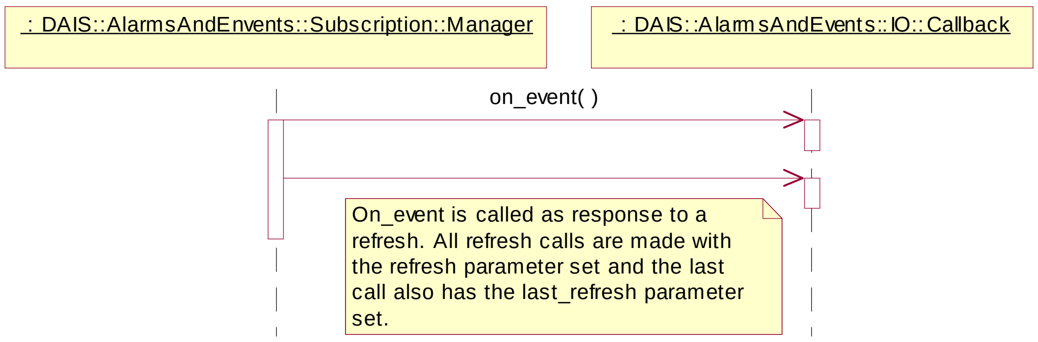

: DAIS::AlarmsAndEnvents::Subscription::Manager
: DAIS::AlarmsAndEvents::IO::Callback
on_event( )
On_event is called as response to a refresh. All refresh calls are made with the refresh parameter set and the last call also has the last_refresh parameter set.

Figure 5-22 Refresh callbacks interaction

References A

A.1 List of References

1. OMG DAIS RFP dtc/99-01-02
2. OMG Utility Management Systems Data Access Facility (DAF) utility/99-11-02
3. OPC Overview; OPC
4. OPC Data access version 2.03; OPC
5. OPC Alarm and events 1.02; OPC
6. OPC Access to Historical data; OPC
7. Guidelines for Control Center APIs; EPRI TR-106324
8. Energy management system APIs; IEC standard 61970 and 61970-30x.
9. UML Toolkit; Eriksson & Penker
10. Structuring principles and reference designations; IEC standard 1346-1
11. CIM UML model from the Rose file cimu09a.mdl.

OMG IDL B

The complete IDL can be found in the zip archive mfg/2001-01-04.

UML Model C

The complete UML model can be found in the Rose file mfg/2001-01-05.

This file is produced in Rose 2000 and is backward compatible at least to Rose 98. It is also possible to open the file with Microsoft Visual Studio 6 Enterprise, Visual Modeler.

Glossary

List of Definitions

DAF - The Utility Management System Data Access Facility.

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.

OPC - OLE for Process Control.

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.

Power System - The integrated facilities and resources that produce, transmit and/or distribute electric energy.

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.