Notification Service Specification

New Edition: June 2000
Version 1.0

Copyright 1998, BEA Systems, Inc.
Copyright 1998, Borland International
Copyright 1998, Cooperative Research Centre for Distributed Systems Technology (DSTC Pty Ltd).
Copyright 1998, Expersoft Corporation
Copyright 1998, FUJITSU LIMITED
Copyright 1998, GMD Fokus
Copyright 1998, International Business Machines Corporation
Copyright 1998, International Computers Limited
Copyright 1998, Iona Technologies Ltd.
Copyright 1998, NEC Corporation
Copyright 1998, Nortel Technology
Copyright 1998, Oracle Corporation
Copyright 1998, TIBCO Software, Inc.

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

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

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG®and

Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, COSS, and IIOP are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

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

Contents

Preface iii

About the Object Management Group iii

What is CORBA? iii

Associated OMG Documents. iv

Acknowledgments iv

1. Service Description 1-1

1.1 Overview 1-1

1.2 Conformance Issues 1-3

1.2.1 Compliance 1-3

2. Architectural Features 2-1

2.1 Overview 2-1

2.1.1 The Notification Service Event Channel Factory 2-4

2.1.2 The Notification Service Event Channel 2-5

2.1.3 Notification Service Style Admin Objects . . . 2-6

2.1.4 Notification Service Style Proxy Interfaces . . 2-7

2.1.5 Sending Events within a Transaction 2-12

2.2 Structured Events 2-12

2.3 Event Filtering with Filter Objects 2-17

2.3.1 Mapping Filter Objects 2-21

2.4 The Default Filter Constraint Language 2-23

2.4.1 Issues with the Trader Constraint Language . 2-23

2.4.2 Trader Constraint Language Extensions for Notification 2-25

2.4.3 Arithmetic Conversions for Mixed Data Types 2-26

Contents

2.4.4 Support for Name-Value Pairs 2-28
2.4.5 A Short-hand Notation for Filtering a Generic Event 2-29
2.4.6 Positional Notation and Intended Applications 2-30
2.4.7 Examples of Notification Service Constraints . 2-31
2.4.8 Extensions to Trader Constraint Language BNF 2-32

2.5 Quality of Service Administration 2-34
2.5.1 Model Components 2-35
2.5.2 QoS Property Representation 2-35
2.5.3 Setting QoS 2-35
2.5.4 End-to-End QoS 2-36
2.5.5 Notification QoS Properties 2-37
2.5.6 Negotiating QoS and Conflict Resolution 2-42
2.5.7 Notification Channel Administrative Properties 2-48

2.6 Sharing Subscriptions 2-49
2.6.1 Sharing Subscriptions Between Channels and Clients 2-49
2.6.2 Offer 2-49
2.6.3 Subscription Change 2-50
2.6.4 Notifications on Demand 2-50
2.6.5 Obligations on Filter Objects 2-51
2.6.6 Special Event Types 2-51

2.7 Filtering Typed Events 2-52

2.8 The Event Type Repository 2-55

2.9 Issues with Interoperability 2-56

3. Modules and Interfaces 3-1

3.1 The CosNotification Module 3-2
3.1.1 The StructuredEvent Data Structure 3-6
3.1.2 The EventBatch Data Type 3-7
3.1.3 QoS and Administrative Constant Declarations 3-8
3.1.4 The QoSAdmin Interface 3-8
3.1.5 The AdminPropertiesAdmin Interface 3-9

3.2 The CosNotifyFilter Module 3-9
3.2.1 The Filter Interface 3-14
3.2.2 The MappingFilter Interface 3-20
3.2.3 The FilterFactory Interface 3-26
3.2.4 The FilterAdmin Interface 3-27

3.3 The CosNotifyComm Module 3-28
3.3.1 The NotifyPublish Interface 3-30

3.3.2 The NotifySubscribe Interface 3-31
3.3.3 The PushConsumer Interface 3-31
3.3.4 The PullConsumer Interface 3-32
3.3.5 The PullSupplier Interface 3-32
3.3.6 The PushSupplier Interface 3-32
3.3.7 The StructuredPushConsumer Interface 3-32
3.3.8 The StructuredPullConsumer Interface 3-34
3.3.9 The StructuredPullSupplier Interface 3-34
3.3.10 The StructuredPushSupplier Interface 3-36
3.3.11 The SequencePushConsumer Interface 3-37
3.3.12 The SequencePullConsumer Interface 3-38
3.3.13 The SequencePullSupplier Interface 3-38
3.3.14 The SequencePushSupplier Interface 3-41

3.4 The CosNotifyChannelAdmin Module 3-41
3.4.1 The ProxyConsumer Interface 3-50
3.4.2 The ProxySupplier Interface 3-53
3.4.3 The ProxyPushConsumer Interface 3-55
3.4.4 The StructuredProxyPushConsumer Interface . 3-57
3.4.5 The SequenceProxyPushConsumer Interface . 3-57
3.4.6 The ProxyPullSupplier Interface 3-58
3.4.7 The StructuredProxyPullSupplier Interface . . . 3-59
3.4.8 The SequenceProxyPullSupplier Interface 3-60
3.4.9 The ProxyPullConsumer Interface 3-61
3.4.10 The StructuredProxyPullConsumer Interface . 3-63
3.4.11 The SequenceProxyPullConsumer Interface . . 3-65
3.4.12 The ProxyPushSupplier Interface 3-66
3.4.13 The StructuredProxyPushSupplier Interface . . 3-68
3.4.14 The SequenceProxyPushSupplier Interface . . . 3-70
3.4.15 The ConsumerAdmin Interface 3-71
3.4.16 The SupplierAdmin Interface 3-76
3.4.17 The EventChannel Interface 3-79
3.4.18 The EventChannelFactory Interface 3-82

3.5 The CosTypedNotifyComm Module 3-84
3.5.1 The TypedPushConsumer Interface 3-84
3.5.2 The TypedPullSupplier Interface 3-85

3.6 CosTypedNotifyChannelAdmin 3-85
3.6.1 The TypedProxyPushConsumer Interface 3-89
3.6.2 The TypedProxyPullSupplier Interface 3-90
3.6.3 The TypedProxyPullConsumer Interface 3-92
3.6.4 The TypedProxyPushSupplier Interface 3-94

Contents

3.6.5 The TypedConsumerAdmin Interface 3-96
3.6.6 The TypedSupplierAdmin Interface 3-99
3.6.7 The TypedEventChannel Interface 3-102
3.6.8 The TypedEventChannelFactory Interface 3-105

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.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* 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.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.

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

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

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

Acknowledgments

The following companies submitted and/or supported parts of the *Notification Service* specification:

- BEA Systems, Inc.
- Borland International
- Cooperative Research Centre for Distributed Systems Technology (DSTC Pty Ltd).
- Expersoft Corporation
- FUJITSU LIMITED
- GMD Fokus
- International Business Machines Corporation
- International Computers Limited

- Iona Technologies Ltd.
- NEC Corporation
- Nortel Technology
- Oracle Corporation
- TIBCO Software, Inc.

Service Description 1

Contents

This chapter contains the following topics.

Topic	Page
"Overview"	1-1
"Conformance Issues"	1-3

1.1 Overview

This specification describes a CORBA-based Notification Service, a service which extends the existing OMG Event Service, adding to it the following new capabilities:

- The ability to transmit events in the form of a well-defined data structure, in addition to Anys and Typed-events as supported by the existing Event Service.
- The ability for clients to specify exactly which events they are interested in receiving, by attaching filters to each proxy in a channel.
- The ability for the event types required by all consumers of a channel to be discovered by suppliers of that channel, so that suppliers can produce events on demand, or avoid transmitting events in which no consumers have interest.
- The ability for the event types offered by suppliers to an event channel to be discovered by consumers of that channel so that consumers may subscribe to new event types as they become available.
- The ability to configure various quality of service properties on a per-channel, per-proxy, or per-event basis.
- An optional event type repository which, if present, facilitates the formation of filter constraints by end-users, by making information about the structure of events which will flow through the channel readily available.

The Notification Service defined here attempts to preserve all of the semantics specified for the OMG Event Service, allowing for interoperability between basic Event Service clients and Notification Service clients. To recap, the OMG Event Service supports asynchronous exchange of event messages between clients. The Event Service introduces *event channels* which broker event messages, *event suppliers* which supply event messages, and *event consumers* which consume event messages.

The figure below depicts a logical view of the *event channel* defined by the OMG Event Service, showing its IDL-defined interfaces.

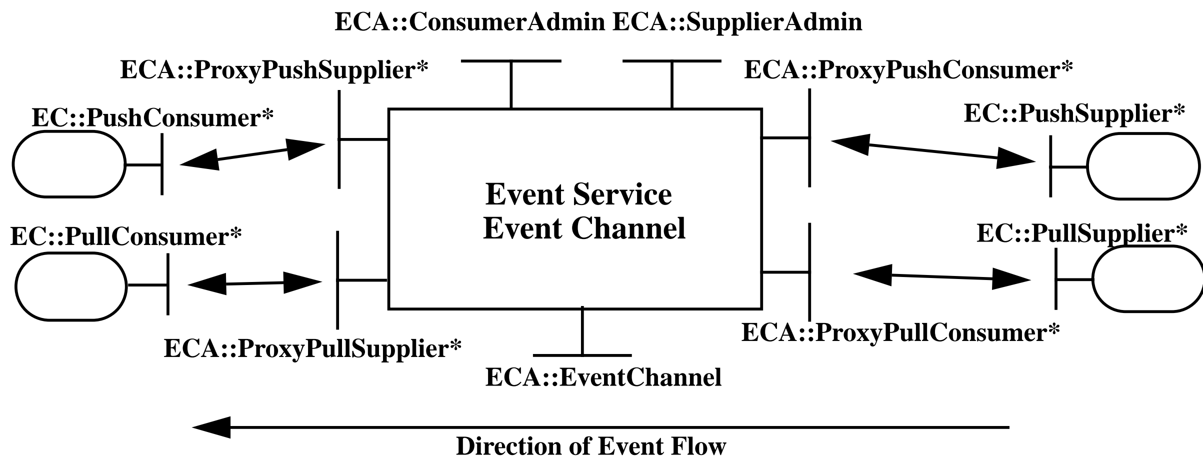

Figure 1-1 Architecture of the untyped OMG Event Channel

The IDL module names of the interfaces defined by the OMG Event Service are abbreviated in the above diagram. ECA stands for **CosEventChannelAdmin**, while EC stands for **CosEventComm**. The "*" next to an interface name denotes the fact that each channel may support one or more of each **Proxy** interface, corresponding to the existence of one or more connected suppliers and/or consumers. Events flow from suppliers to consumers, as depicted by the arrow on the bottom of the figure. Note that this figure depicts the **untyped** event channel defined by the OMG Event Service. A **typed** version also exists that has similar architecture, with additional interfaces defined to handle typed event communication.

Two serious limitations of the event channel defined by the OMG Event Service are that it supports no event filtering capability, and no ability to be configured to support different qualities of service. Thus, the choice of which consumers connected to a channel receive which events, along with the delivery guarantee that is made to each supplier, is hard-wired into the implementation of the channel. Most Event Service implementations deliver all events sent to a particular channel to all consumers connected to that channel on a best-effort basis.

A primary goal of the Notification Service defined here is to enhance the Event Service by introducing the concepts of filtering, and configurability according to various quality of service requirements. Clients of the Notification Service can subscribe to specific events of interest by associating filter objects with the proxies through which the clients communicate with event channels. These filter objects encapsulate

constraints which specify the events the consumer is interested in receiving, enabling the channel to only deliver events to consumers which have expressed interest in receiving them. Furthermore, the Notification Service enables each channel, each connection, and each message to be configured to support the desired quality of service with respect to delivery guarantee, event aging characteristics, and event prioritization.

The Notification Service defined here supports event filtering on three fundamental types of events: **untyped** events contained within a CORBA Any, **typed** events as defined by the OMG Event Service, and **structured** events, which are introduced in this specification. **Structured** events define a well-known data structure which many different types of events can be mapped into in order to support highly optimized event filtering.

1.2 Conformance Issues

1.2.1 Compliance

In order to be conformant with this specification, all of the interfaces must be supported and implemented using the specified semantics, with the exception of the interfaces for typed notification channels, which are optional. In addition, a conforming implementation must support filter objects that support constraints expressed in the default constraint grammar defined in Section 2.4, "The Default Filter Constraint Language," on page 2-23. Lastly, this document defines a set of standard QoS properties, which must at least be understood (although not necessarily implemented) by all conformant implementations.

More precisely,

- A conforming implementation must support all interfaces defined in the **CosNotification**, **CosNotifyFilter**, **CosNotifyComm**, and **CosNotifyChannelAdmin** modules.
- A conforming implementation may also support, in addition to the interfaces enumerated above, all of the interfaces defined in the **CosTypedNotifyChannelAdmin** module.
- A conforming implementation will provide implementations of the **CosNotifyFilter::Filter** and **CosNotifyFilter::MappingFilter** interfaces that support constraints expressed in the default constraint grammar specified in Section 2.4, "The Default Filter Constraint Language," on page 2-23.
- All QoS properties defined in Chapter 2 of this specification must at least be understood by any conforming implementation. However, a conforming implementation may choose to not implement all standard QoS properties and/or QoS property settings. In cases where a client requests a standard QoS property with a setting that is not supported by a conformant implementation, the implementation should raise the CosNotification::UnsupportedQoS exception.

Architectural Features 2

Contents

This chapter contains the following topics.

Topic	Page
"Overview"	2-1
"Structured Events"	2-12
"Event Filtering with Filter Objects"	2-17
"The Default Filter Constraint Language"	2-23
"Quality of Service Administration"	2-34
"Sharing Subscriptions"	2-49
"Filtering Typed Events"	2-52
"The Event Type Repository"	2-55
"Issues with Interoperability"	2-56

2.1 Overview

This section provides a general overview of the service architecture. The main design goal of the Notification Service architecture is to define the service as a direct extension of the existing OMG Event Service, enhancing the latter with important features which are required to satisfy a variety of applications with a broad range of scalability, performance, and quality of service (QoS) requirements. The guiding principles which drove the definition of the Notification Service IDL interfaces were to preserve both backward compatibility with and the programming model of the OMG Event Service. The former principle lead to the specification of IDL modules which have identical structure to corresponding Event Service IDL modules, containing interfaces which inherit directly from those defined in the Event Service. The latter

principle lead to the specification of Notification Service interfaces which are named similarly to corresponding Event Service interfaces, and which define new operations that preserve the semantic behavior of the Event Service operations whose functionality they are intended to embellish.

The Notification Service defined here supports all of the interfaces and functionality supported by the OMG Event Service. In fact, an implementation of the Notification Service defined here can be thought of as subsuming an implementation of the Event Service. The Notification Service, however, also supports new features that are introduced by directly extending the interfaces defined by the Event Service. Both the original Event Service interfaces, and these new extended interfaces specific to Notification, are made available to Notification Service clients in order to preserve backward compatibility.

The general architecture of the Notification Service is depicted in Figure .

Once again the IDL module names of the interfaces defined by the service are abbreviated in the diagram. The following is a key to the abbreviations used:

- EC - CosEventComm
- ECA - CosEventChannelAdmin
- NC - CosNotifyComm
- NCA - CosNotifyChannelAdmin

As in Figure 1-1 on page 1-2, the "*" next to an interface name denotes the fact that there may be multiple object instances supporting this interface in a given channel. Note that in addition to supporting multiple instances of each Proxy interface, each Notification Service event channel may also support multiple instances of the **ConsumerAdmin** and **SupplierAdmin** interfaces defined in the **CosNotifyChannelAdmin** module. The reason for this will be explained shortly. Also note that this figure depicts the generic Notification Service event channel. As with the Event Service, a typed version also exists that has similar architecture, with additional interfaces to handle typed communication. The **typed** Notification Service event channel will also be described shortly.

As depicted in on page 2-3, an instance of the Notification Service event channel (referred to henceforth as the **notification channel**) logically supports all of the interfaces supported by the Event Service event channel. In many cases the service supports two methods for obtaining access to the Event Service version of a particular interface:

1. Since the Notification Service version of a particular interface inherits from the Event Service equivalent of the same interface, an instance of the former can be widened to an instance of the latter. Examples of this are the **EventChannel**, **ConsumerAdmin**, and **SupplierAdmin** interfaces.

2. The factory operations supported by a Notification Service interface through inheritance from the equivalent Event Service interface can be invoked to create a true Event Service version of a particular interface. Examples of Event Service interfaces that can be instantiated in this way are the **ConsumerAdmin**, **SupplierAdmin**, and all **Proxy** interfaces supported by the Event Service.

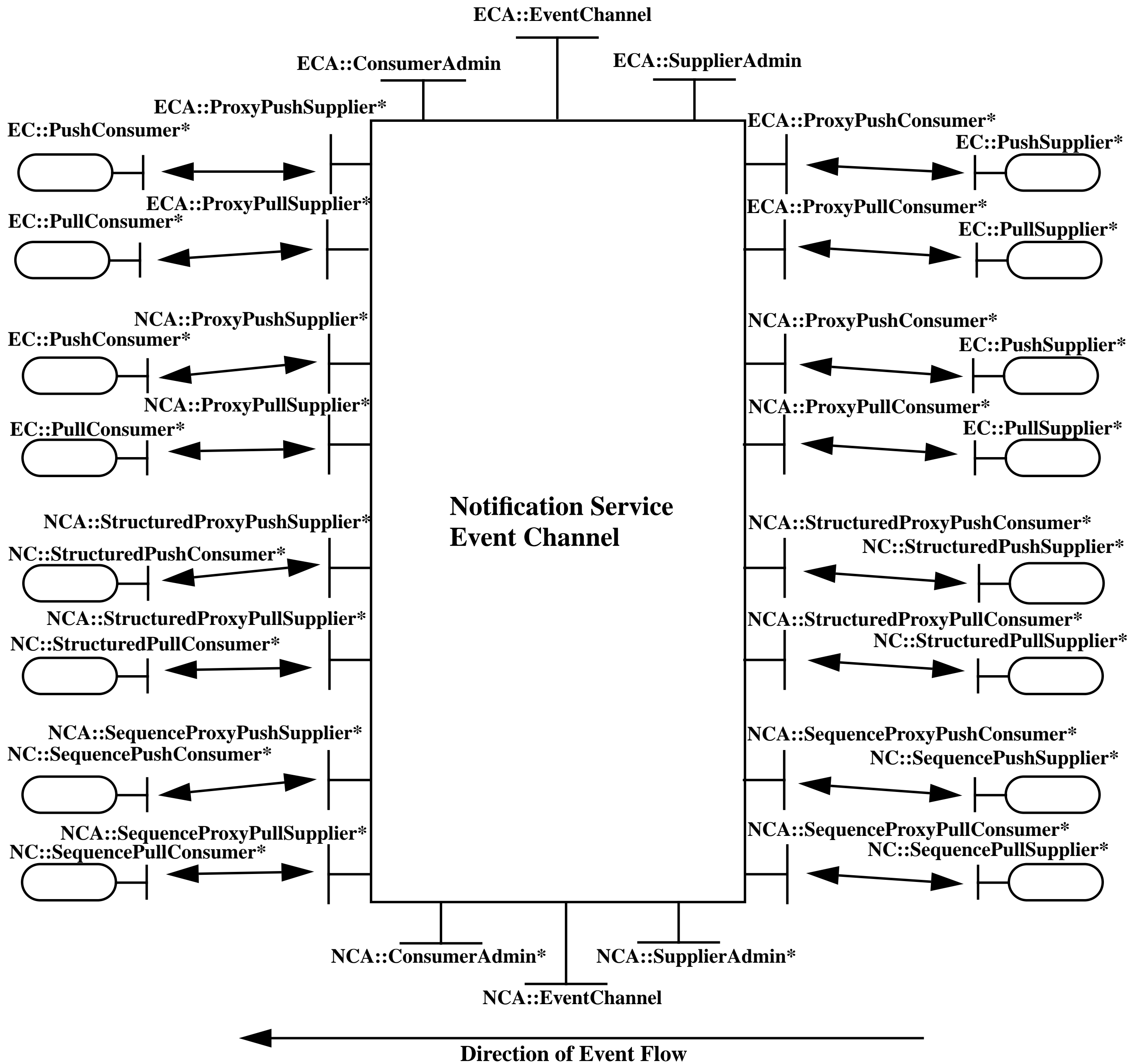

Figure 2-1 General Architecture of the Notification Service

Note that in this specification, the issue of whether or not an instance of an Event Service style interface obtained by method 2) above can be narrowed to an equivalent Notification Service style interface is left as an implementation detail.

Due to interface inheritance, an instance of an object supporting the **NCA::EventChannel** interface (i.e., an instance of a notification channel) can be widened to one supporting the **ECA::EventChannel** interface, and henceforth be treated identically to the Event Service's version of an event channel. The primary reason for this is to support backward compatibility for existing applications that use the Event Service. Furthermore, more fine-grained application migration is supported as true Event Service clients (i.e., consumers and suppliers) can connect to the Notification Service event channel using one of the following three techniques:

1. Using the operations of the inherited **ECA::EventChannel** interface, instantiate the Event Service version of the appropriate **Admin** interface (i.e., **ConsumerAdmin** or **SupplierAdmin**), use it to instantiate an Event Service style **Proxy** interface, then connect to that interface.

2. Obtain the appropriate Notification Service style **Admin** interface. Then, using the operations supported through inheritance of the Event Service version of the analogous **Admin** interface, instantiate an Event Service style **Proxy** interface, and connect to that interface.

3. Use strictly Notification Service style interfaces to instantiate and connect to the **ProxyPushConsumer**, **ProxyPullConsumer**, **ProxyPushSupplier**, or **ProxyPullSupplier** interface defined in the **CosNotifyChannelAdmin** model.

Techniques 1) and 2) described above differ only in at what point (the **EventChannel** or **Admin** interface) the client begins treating the notification channel as a true event channel. The end result of both techniques is identical: a true event service client is connected to a true event service style proxy interface associated with the channel. These techniques enable the client to achieve the identical functionality supported by the Event Service event channel: events in the form of untyped Anys can be supplied to and consumed from the channel.

Technique 3) enables an event service client to take advantage of some of the new functionality supported by the notification channel. The main difference between this and the previous techniques is that it results in an Event Service client being connected to a Notification Service style proxy interface. The Notification Service style proxy interface is capable of filtering events based on end-user provided constraints, and can also be configured to support various qualities of service. Thus, in this case an Event Service client is able to take advantage of the new features supported by Notification.

The previous discussion described how Event Service clients can use the Notification Service. The following subsections will focus on the new features offered by Notification, which are available to clients which have been newly developed to use Notification.

2.1.1 The Notification Service Event Channel Factory

The Notification Service supports a well-defined factory interface, the **EventChannelFactory**, for creating new instances of notification channels. At creation time, the client can specify various QoS and administrative properties that will be supported by the channel. The standard administrative properties that can be set on a channel include the maximum number of events the channel will buffer at any one

time (**MaxQueueLength**), and the maximum number of consumers and suppliers that can connect to the channel (**MaxConsumers** and **MaxSuppliers**). QoS administration is described in detail in Section 2.5, “Quality of Service Administration,” on page 2-34.

Although the **EventChannelFactory** is the only interface in the Notification Service that is explicitly defined to be a factory (i.e., an object that creates other objects), it turns out that the architecture of the service is hierarchical in nature, and all objects defined as part of an event channel are created by some parent object. For instance, consumer and supplier admin instances are created by event channels, and all proxy objects are created by some admin instance.

One important design principle introduced by the Notification Service is that all objects that create other objects assign numeric identifiers to the objects they have created that are unique among all objects they have created. In addition, all objects that create other objects support an operation that returns the list of all unique identifiers they have assigned to objects they have created, and an operation that given a single unique identifier corresponding to an object they have created, can return the object reference of that child object. Additionally, all objects within the channel maintain back references to their parent object (e.g., event channels maintain references to the event channel factory that created them, admin objects maintain references to the event channel that created them, etc.). This design principle significantly enhances the administratibility of a Notification Service event channel, by enabling any client of a channel to discover all objects that comprise the channel, starting from any object within the channel. Note that a Notification Service event channel which contains objects that support pure OMG Event Service style interfaces will not be able to administer these objects in this fashion (since Event Service style objects will not have associated unique identifiers, and will not maintain backreferences to their parent objects).

2.1.2 The Notification Service Event Channel

The Notification Service event channel, also referred to as the “notification channel,” supports the **CosNotifyChannelAdmin::EventChannel** interface. Through interface inheritance, an instance supporting this interface can be treated exactly like an Event Service event channel (as previously described), and can have both QoS and administrative properties assigned to it.

A feature newly introduced by the notification channel is its ability to support multiple instances of objects supporting the **ConsumerAdmin** and **SupplierAdmin** interfaces (referred to in generic terms as “Admin” interfaces). Each Admin interface is essentially a factory that creates the Proxy interfaces to which clients will ultimately connect. The Notification Service also treats each Admin object as the manager of the group of Proxies it has created. Admin objects can themselves have QoS properties and filter objects (explained in detail in Section 2.3, “Event Filtering with Filter Objects,” on page 2-17) associated with them. The QoS properties associated with a given Admin object are assigned to each Proxy object created by the Admin object upon creation of the Proxy, but can subsequently be tailored on a per-proxy basis. On the other hand, the set of filter objects associated with a given Admin are treated as a unit which apply at all times to all Proxy objects which have been created by the Admin.

Additional filter objects can be associated with an individual Proxy, but the set of filter objects associated with an Admin object are automatically associated with all Proxy objects which have been created by the Admin object, and this set can only be modified by invoking operations on the Admin object.

Sharing a set of filter objects among all Proxy objects created by an Admin object provides a powerful mechanism for creating a set of event subscriptions that can be shared by a group of clients. In addition, the filtering of a given event on behalf of a set of clients can be optimized since the same subscription information applies to multiple clients, implying that the filtering of a given event can be performed once for a given set of clients. In summary, supporting multiple Admin objects in a given notification channel enables the logical grouping of the Proxy objects associated with the channel according to common subscription information. This feature is particularly useful with respect to **ConsumerAdmin** objects, since it enables the channel to optimize the servicing of a group of consumers that are interested in receiving the same set of events.

The **CosNotifyChannelAdmin::EventChannel** interface supports the operations for creating new **ConsumerAdmin** and **SupplierAdmin** instances. Each instance is assigned a unique identifier upon creation, which can subsequently be used to obtain the reference of a particular Admin object by invoking an operation on the **EventChannel** interface. Upon creation, each **EventChannel** instance initially supports a single **ConsumerAdmin** and **SupplierAdmin** instance, viewed as the default of each such type of object and assigned the unique identifier value of zero. Note that through inheritance of the **CosEventChannelAdmin::EventChannel** interface, each Notification Service event channel is also capable of creating Event Service style Admin instances, which can subsequently be used to create Event Service style Proxy objects. These event service style Admin instances do not have unique identifiers associated with them, and thus cannot subsequently be obtained by invoking the notification channel operations to obtain an Admin object by unique identifier.

2.1.3 Notification Service Style Admin Objects

As described in the previous section, each notification channel can have associated with it multiple instances of **ConsumerAdmin** and **SupplierAdmin** objects. Both styles of Admin object can have QoS properties and filter objects associated with them. The QoS properties associated with a given Admin object are assigned as the default QoS properties which will be associated with any Proxy object created by that Admin. The properties can be subsequently tailored on a per-Proxy basis if so desired by clients of the service. On the other hand, the filter objects associated with a given Admin object are treated as a unit that are logically associated with every Proxy object that has been created by the Admin. This unit can be modified only by invoking operations on the Admin object itself, and such changes to the set of filters, or to the internal state of the filters themselves, affect every Proxy which was created by the Admin object.

The main idea underlying the support of multiple Admin objects per channel is to optimize the handling of clients with identical requirements. For example, if it is desired to connect multiple consumer applications that are interested in receiving an identical set of events to a notification channel, this would be achieved by creating a

single **ConsumerAdmin** object and associating with it the filters which encapsulate the constraints specifying the desired set of events. Subsequently, each consumer application interested in receiving the specific set of events would connect to the channel using this particular **ConsumerAdmin** object to create its associated Proxy Supplier object. In addition, the same channel could be connected to by a different group of consumer applications interested in receiving a different set of events by creating a new **ConsumerAdmin** object, associating with it a different set of filter objects, and using it to create the new set of Proxy Supplier objects.

Each instance of Notification Service style Admin object is capable of creating and managing a set of Proxy objects. Through inheritance of the analogous Event Service style Admin interface, a Notification Service style Admin object can be used to create Event Service style Proxy objects. In addition, the Admin interfaces defined by the Notification Service support operations to create Notification Service style Proxy objects. Upon creation of such a Proxy object, the Admin object assigns it a unique identifier which can be subsequently used as input to an operation supported by the Admin interface to return the object reference of the Proxy. Note that only Notification Service style Proxy objects will have unique identifiers associated with them, and thus be obtainable through the operations supported by the Admin interfaces which return a Proxy object interface given a unique identifier as input.

The Notification Service introduces separate Proxy interfaces depending on the desired form of message communication. Message communication to and from the channel can be in terms of Anys, Structured Events, or sequences of Structured Events. The exact form of message communication supported by the Proxy object created by a Notification Service style Admin object is controlled by a flag provided as input to the operations on the Admin objects which create new Proxies. The different styles of Proxy object which can be created by invoking operations on a Notification Service style Admin object are explained in the next subsection.

2.1.4 Notification Service Style Proxy Interfaces

Using the operations supported through inheritance of the analogous Event Service interface, a client of the Notification Service can use a Notification Service style Admin object to create a pure Event Service style Proxy object. Such a Proxy supports the identical behavior of the Event Service style Proxy objects: a consumer Proxy is capable of receiving events in the form of Anys, and a supplier Proxy is capable of delivering events in the form of Anys.

Using the newly defined operations of the Notification Service style Admin objects, it's possible to create Notification Service style Proxy objects. As with the Event Service, both push and pull styles of each type of Proxy are supported. In addition, Notification Service style Proxy objects can be further subdivided into three distinct categories: those that send and receive events in the form of Anys, those that send and receive events in the form of Structured Events (described in Section 2.2, "Structured Events," on page 2-12), and those that send and receive events in the form of sequences of Structured Events. The Notification Service also defines new interfaces for clients that send and receive events in the form of Structured Events or Sequences of Structured Events.

The following table summarizes the Notification Service style Proxy interfaces, and the types of clients of an object supporting each interface.

Table 2-1 Notification Service Style Proxy Interfaces

Proxy Interface	**Connected To By**	**Form of Message**
ProxyPushConsumer	CosEventComm::PushSupplier	Any
StructuredProxyPushConsumer	CosNotifyComm::StructuredPushSupplier	Structured Event
SequenceProxyPushConsumer	CosNotifyComm::SequencePushSupplier	Sequence of Structured Event
ProxyPullConsumer	CosEventComm::PullSupplier	Any
StructuredProxyPullConsumer	CosNotifyComm::StructuredPullSupplier	Structured Event
SequenceProxyPullConsumer	CosNotifyComm::SequencePullSupplier	Sequence of Structured Event
ProxyPushSupplier	CosEventComm::PushConsumer	Any
StructuredProxyPushSupplier	CosNotifyComm::StructuredPushConsumer	Structured Event
SequenceProxyPushSupplier	CosNotifyComm::SequencePushConsumer	Sequence of Structured Event
ProxyPullSupplier	CosEventComm::PullConsumer	Any
StructuredProxyPullSupplier	CosNotifyComm::StructuredPullConsumer	Structured Event
SequenceProxyPullSupplier	CosNotifyComm::SequencePullConsumer	Sequence of StructuredEvent

Notification Service style Proxy objects can have two different types of filters associated with them. “Forwarding filters” can be attached to all types of Proxy objects and constrain the events that the Proxy will forward. “Mapping filters” can only be attached to supplier Proxy objects and affect the priority or lifetime properties of each event received by a supplier Proxy. Mapping filters are discussed in more detail in Section 2.3.1, “Mapping Filter Objects,” on page 2-21.

Both forwarding and mapping filters can be associated with a Proxy object either explicitly or implicitly. Explicit association implies that the filters were associated with the Proxy object by invoking an operation directly on the Proxy object to form the association. Alternatively, filters can be associated with an Admin object. A Proxy object also implicitly has associated with it all filter objects that are associated with the Admin object which created it. Note that a Proxy object with no associated filter objects defaults to forwarding all events it receives.

Each Notification Service style proxy object can also have various QoS properties associated with it. The QoS properties which can be associated with a Notification Service style Proxy object, and how they are treated with respect to QoS properties set on a notification channel-wide basis, and potentially those set on a per-message basis, are described in Section 2.5, “Quality of Service Administration,” on page 2-34.

Note that by dividing the Notification Service style Proxy interfaces along the lines of the form of message they are capable of transmitting, and by defining separate client interfaces (in the **CosNotifyComm** module) for clients that deal with each specific form of message, clients of the Notification Service have the freedom to implement consumers and suppliers that deal with events in only the specific format(s) they are interested in sending and receiving them. For instance, to develop a consumer application which only receives events by push-style communication in the form of Structured Events, the developer simply needs to implement the **CosNotifyComm::StructuredPushConsumer** interface, which only supports a push operation which receives events in the form of Structured Events. The Notification Service supports well-defined translations of message format in the case that an event is supplied in a format different than a particular consumer is designed to receive (e.g., an event is supplied as an Any, but the consumer only implements the **StructuredPushConsumer** interface). This translation is summarized in the table below. Note that this translation model naturally extends to the Typed notification channel to which typed event service clients may also connect, and thus translations involving typed events are included in the table as well. Typed Notification is described in Section 2.7, "Filtering Typed Events," on page 2-52.

Table 2-2 Message Translations Performed by the Notification Channel

Form Supplier Sends Events to Channel	Form Consumer Receives Events from Channel	Translation Performed by Channel
Any	Structured Event	Event is packaged into a Structured Event data structure, with the content of the Any assigned to the "remainder_of_body" portion of the structure (see Section 2.2, "Structured Events," on page 2-12). The "type_name" data member of the Structured Event should be set to the value "%ANY", and the "domain_name" member set to the empty string.
Any	Typed Event	The Any must contain a sequence of name-value pairs whose first element must have the name "operation", and corresponding value of type string which nominates the fully scoped operation name to be invoked. The additional elements of the sequence will contain properties with names and values corresponding to the names and value types of the parameters of the typed operation signature. The contents of an Any that do not follow this convention will not result in the event being delivered to any Typed clients.
Structured Event	Any	A new Any is created, with the Structured Event assigned to its value field, and its Typecode set appropriately to indicate a Structured Event data structure.

Table 2-2 Message Translations Performed by the Notification Channel

Form Supplier Sends Events to Channel	**Form Consumer Receives Events from Channel**	**Translation Performed by Channel**
Structured Event	Typed Event	Channel presumes that the filterable data portion of the Structured Event contains a sequence of name/value pairs whose first element has the name “operation”, and corresponding value of type string which nominates the fully scoped operation name to be invoked. The additional elements of the sequence are the name and value of each of the parameters for this operation. If the contents of the Structured Event follow this convention, the typed operation is invoked. Otherwise, the typed client will not receive the event.
Typed Event	Any	A new Any is created containing a data structure that is a sequence of name-value pairs. The name of the first element of this sequence is “operation”, and its value will be a string containing the fully scoped operation name. The remaining elements of the sequence indicate the name of each parameter of the typed operation used to transmit the event to the channel, and for each such name the corresponding value is the value that was passed for that parameter during the invocation by the typed supplier.
Typed Event	Structured Event	A new Structured Event is created, whose filterable data is populated with the contents of the typed event. The first element of the name-value pair sequence that makes up the filterable data has its name set to “operation”, and its value will be a string containing the fully scoped operation name. The remaining elements of the sequence indicate the name of each parameter of the typed operation used to transmit the event to the channel, and for each such name the corresponding value is the value that was passed for that parameter during the invocation by the typed supplier. The “type_name” data member of the Structured Event should be set to the value “%TYPED”, and the “domain_name” member set to the empty string.

In addition to there being separate proxy interfaces for sending events as Anys and Structured Events, there are also proxy interfaces which support transmission of sequences of Structured Events. These proxies should be seen as a shortcut for the use of equivalent proxies which transmit single Structured Events. For example, an invocation of the **push_structured_events** operation supported by a **SequencePushConsumer**, which is passed a sequence of length **n** as input is equivalent to performing **n** calls to the **push_structured_event** operation supported by a **StructuredPushConsumer**. Any untyped events in the queue to be transmitted to a sequence consumer are converted into Structured Events in the same way as

described in Table 2-2 on page 2-9. Likewise, the translation of any Structured Event supplied to a channel within a sequence, and destined for a consumer of untyped events is performed in the same way as with a single Structured Event.

This translation scheme raises the potential for an event to become *wrapped* multiple times as the result of multiple translations, thus deeply embedding the original contents of the event inside multiple such wrappings. Suppose for example, notification channel B is set up to be a consumer of events supplied to notification channel A by connecting a **SequenceProxyPushConsumer** for channel B to a **SequenceProxyPushSupplier** of channel A. Now suppose an event in the form of an Any is supplied to channel A, and is destined for transmission to channel B. Channel A will translate the Any event it received into a Structured Event before delivering it to channel B. Now suppose a consumer of untyped events is connected to channel B, and the wrapped Any event received from channel A is destined for that consumer. Naively, channel B would perform the appropriate translation of the Structured Event into an Any, delivering to the consumer an Any, wrapped within a Structured Event, wrapped within another Any. In addition, all filters containing expressions that address fields in the event will fail to match the wrapped event in either channel.

To avoid this type of situation, the special event type %ANY is defined and assigned to the **type_name** field of the header of any Structured Event which contains an Any as the result of the translation of the Any into a Structured Event. Proxy consumers accepting Structured Events (including those which accept sequences of Structured Events) must examine each event to see if its type is set to this value. In such cases, before performing any filtering or further translations back into an Any, the original Any is extracted from the **remainder_of_body** member of the Structured Event.

Similarly, the wrapping of an Any containing a Structured Event inside another Structured Event must be avoided. Proxy consumers accepting Anys must look at the **TypeCode** for the Any to see if it corresponds to the **TypeCode** for a Structured Event. In this case, the value portion of the Any is extracted into a Structured Event data structure before any filtering or further translations are performed.

One additional aspect of the Notification Service style Proxy interfaces, and the newly defined Notification client interfaces, is that they support a means to share event subscription information between notification channels and their clients. Each type of Notification Service supplier interface (e.g., the supplier clients defined in **CosNotifyComm** and the supplier Proxies defined in **CosNotifyChannelAdmin**) inherits a **NotifySubscribe** interface. This interface supports an operation that allows each notification supplier to be notified when the set of events for which there are currently interested consumers changes. Likewise, each type of Notification Service consumer interface (e.g., the consumer clients defined in **CosNotifyComm** and the consumer Proxies defined in **CosNotifyChannelAdmin**) inherits a **NotifyPublish** interface. This interface supports an operation that allows each notification consumer to be notified when the set of event types which are currently being offered to the channel changes. These mechanisms can be used in concert by an implementation of the Notification Service, transparently to clients of the service, in order to optimize event communication by only transmitting events when necessary. How this can be achieved is described in Section 2.6, “Sharing Subscriptions,” on page 2-49.

2.1.5 Sending Events within a Transaction

The recently adopted CORBA Messaging standard incorporates several changes to the standard OMG Object Transaction Service (OTS). Included among these changes is the deprecation of the **TransactionalObject** interface, which was previously required to be inherited by any interface that supported operations which could be invoked within the context of a transaction.

In certain situations, it may be desirable to transmit one or more events within the context of a transaction. Transactional event transmission can be considered in two distinct cases:

- when a supplier sends one or more events to the channel
- when the channel sends one or more events to consumers

Before the modifications to the OTS described above, it would have been necessary to define special interfaces to support transactional event transmission. In order to support the first case above, it would have been necessary to define a transactional variant of each proxy push-style interface (to support passing the transaction context within the suppliers invocation of the proxy's **push** operation), and a transactional variant of each client pull-style interface (to support passing the transaction context within the proxy consumer's invocation of the client's **pull** or **try_pull** operation). Likewise, in order to support the second case listed above, it would have been necessary to define a transactional variant of each client push-style interface, and a transactional variant of each proxy pull-style interface.

But due to the deprecation of the **TransactionalObject** interface, it is no longer necessary to define special interfaces that inherit from **TransactionalObject** in order to enable a transaction context to be passed within a method invocation. This change has both advantages and disadvantages from the perspective of the Notification Service. On the positive side, the fact that it is not necessary to define specific transactional interfaces simplifies the Notification Service IDL to some degree. On the negative side, however, support of the full Notification Service IDL by an implementation does not itself guarantee that the implementation supports transactional event transmission.

As described in the CORBA Messaging specification, transactionality is viewed as an attribute of an implementation of an interface, which is specified by defining a policy attribute on the POA. In order to support transactional event transmission, an implementation of the Notification Service should support implementations of the various proxy interfaces that are POA objects that support **TransactionPolicy**.

2.2 Structured Events

The OMG Event Service supports two styles of event communication: untyped and typed. Untyped communication involves transmitting all events in the form of Anys. While untyped event communication is generic and easy-to-use, many applications require more strongly typed event messages. To satisfy this latter requirement, the OMG Event Service defines interfaces and conventions for supporting typed event

communication. Unfortunately, many users have found typed event communication as defined by the OMG Event Service difficult to understand, and implementors have found it particularly difficult to deal with.

For these reasons, the Notification Service introduces a new style of event message: the *Structured Event*. Structured Events provide a well-defined data structure into which a wide variety of event types can be mapped. Typically when using the untyped style of event communication supported by the Event Service, clients define a data structure into which they store an event message, then package that data structure into an Any. Structured Events define a standard data structure into which a wide variety of event messages can be stored. New supplier and consumer interfaces are defined by the Notification Service so that Structured Events can be transmitted directly, without needing to be packaged into an Any. Because the structure of Structured Events is known to both Notification Service clients and the notification channel, algorithms that filter and manipulate Structured Events can be optimized. Structured Events provide the equivalent generality and ease-of-use of untyped event communication, while providing more strongly typed event communication.

In a sense, Structured Events can be viewed as a manifestation of typed event style communication, where the typed interfaces defined for sending and receiving this specific type of event message (i.e., the **I** interface as defined by the OMG Event Service for typed event communication) are explicitly defined. Because the interfaces for dealing with Structured Events are explicitly defined, it is straightforward to both use and implement a notification channel that supports this style of event communication. Structured Events thus provide the advantages of typed event communication, without the difficulties inherent in implementing and using the typed style of event communication defined by the OMG Event Service. Figure 2-2 on page 2-14 shows the structure of a Structured Event.

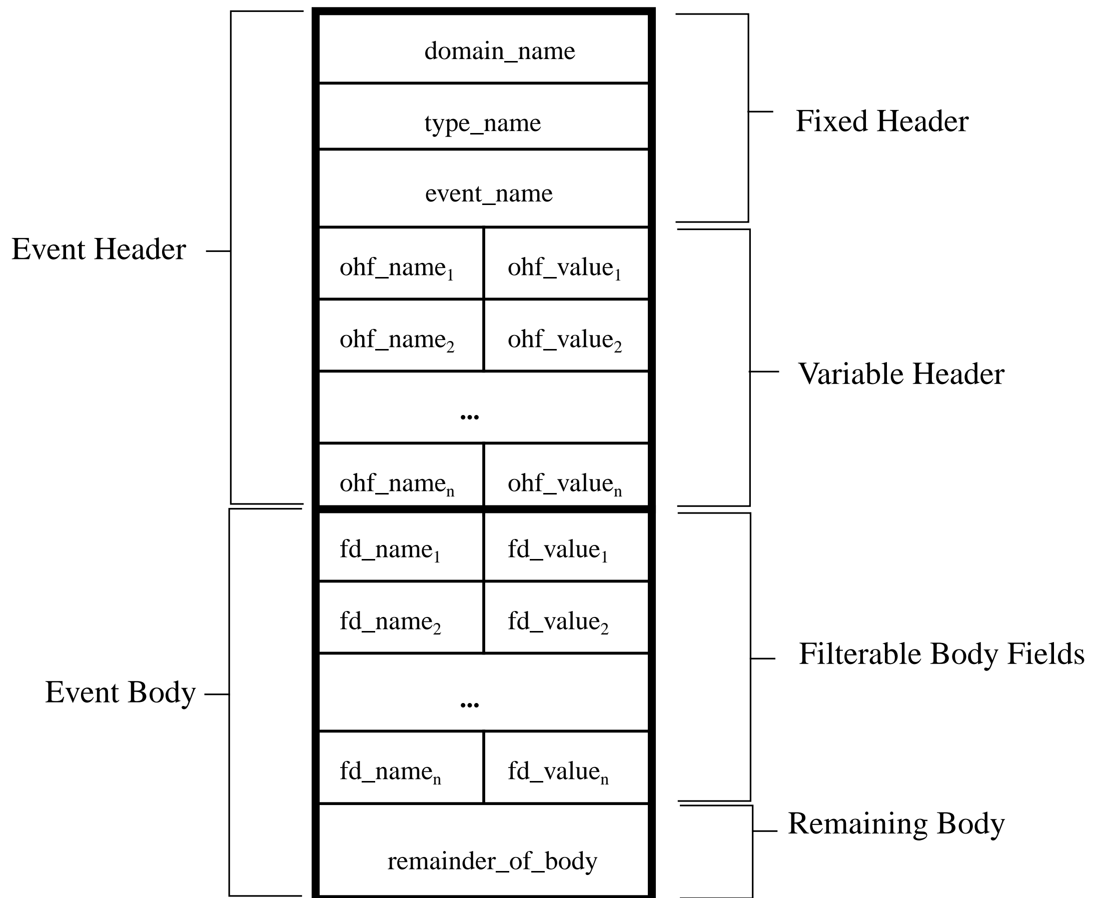

Figure 2-2 The structure of a Structured Event

Figure 2-2 depicts the general format of a structured event. Each event is comprised of two main components: a header and a body. The header can be further decomposed into a fixed portion and a variable portion. The goal of this decomposition is to minimize the size of the header which is required in every Structured Event message, thus enabling lightweight messages where the overhead of supplying additional header fields is viewed as less desirable than any functional benefit achieved by supplying these fields (e.g., additional header fields may contain QoS requirements for the message).

The fixed portion of the event header is comprised of three string fields:

- a **domain_name**, which identifies the particular vertical industry domain in which the event type is defined (e.g., telecommunications, finance, health care, etc.);
- a **type_name**, which categorizes the type of event uniquely within the domain (e.g., **CommunicationsAlarm**, **StockQuote**, **VitalSigns**); and
- an **event_name**, which may uniquely identify the specific instance of event being transmitted.

Note that the combination of the **domain_name** and **type_name** fields could be used as indexes into the event type repository (see Section 2.8, “The Event Type Repository,” on page 2-55), which contains a complete description of the fields of that specific type of event. Thus when Structured Events are used in concert with the event type repository, it is particularly convenient for consumers to receive new types of events, and discover the structure of their contents.

The variable portion of the event header is comprised of a list of zero or more name-value pairs[1] (the “ohf_” prefacing each name-value pair in the figure stands for “optional header field”), where each name is a string, and each value is an Any. While inclusion of these fields is optional and their contents are virtually unbounded, this specification standardizes a set of well-defined optional header field names and defines the data types of their values. These standard optional header fields contain per-message QoS related information. How this information is treated with respect to per-Proxy QoS settings and per-channel QoS settings is described in Section 2.5, “Quality of Service Administration,” on page 2-34. The table below summarizes the standard optional header field names, the data types of their associated values, and a brief description of their meanings. Note also that end-users can define additional proprietary optional header fields. Those in the table below are viewed as standard, however, and every implementation of the Notification Service must be capable of interpreting and handling them with respect to their intended meaning.

1. Note that the phrase “list of name-value pairs” is used frequently throughout this specification to mean an instance of the type CosNotification::PropertySeq. This phrase should not be confused with the common CORBA terms “Named Value”, “Name Value List”, or the data type CORBA::NVList.

Table 2-3 Standard optional header fields

Header field name	**Type of associated value**	**Meaning**
EventReliability	short	This value portion of this header field has two well defined settings: 0 means “best effort”; 1 means “persistent”. If set to 0, event can be treated as non-persistent and lost upon failure of the channel. At least one attempt must be made to transmit the event to each registered consumer, but in the case of a failure to send to any consumer, no further action need be taken. If set to 1, channel should make the event persistent, and attempt to retransmit upon channel recovery from failure. This setting only has meaning when ConnectionReliability is also set to 1, in which the combination essentially means guaranteed delivery.
Priority	short	Indicates the relative priority of the event compared to other events in the channel. Can take on any value between -32,767 and 32,767, with -32,767 being the lowest priority, 32,767 being the highest, and 0 being the default.
StartTime	TimeBase::UtcT	Gives an absolute time (e.g., 12/12/99 at 23:59) after which the channel can deliver the event.
StopTime	TimeBase::UtcT	Gives an absolute time (e.g., 12/12/99 at 23:59) when the channel should discard the event.
Timeout	TimeBase::TimeT	Gives a relative time (e.g., 10 minutes from time received) when the channel should discard the event). The special value *zero* indicates there is no timeout.

Note that the priority and timeout properties may optionally be set within the header of a Structured Event. A proxy receiving a Structured Event may also have a priority and timeout quality of service setting. In this case, if the priority and/or timeout fields are set within the header of the Structured Event, these settings override those set at the proxy level. If one or both of these properties is set on a proxy receiving a Structured Event but not within the header of a Structured Event itself, the setting at the proxy level is used to determine how that proxy treats that event with respect to priority and/or timeout. An important point to note, however, is that no object within the channel ever modifies the contents of a Structured Event.

The second main portion of the structured event is the event body, which is intended to contain the contents of each event instance. The event body is also decomposed into two parts: a filterable portion and the remainder of the body. The filterable portion is intended to contain the most interesting fields of the event, upon which the consumer is most likely to base filtering decisions. Like the optional header fields, the filterable portion of the event body is also defined as a sequence of name-value pairs, with each name being a string and each value an Any (the “fd_” prefacing each name-value pair in the figure stands for “filterable data”).

It is envisioned that different vertical domains will define standard mappings of specific event types into Structured Events. Each such mapping will standardize the name-value pairs that make up the filterable portion of a particular type of event mapped into a Structured Event. Thus while the definition of the filterable data fields contained within the Structured Event data structure may appear to be too generic to provide any real advantage, the advantage of this structure becomes more apparent when viewed in the context of mappings of actual event types into it, since these mappings specify well-defined name-value pairs that go into the filterable portion of the body.

The last portion of the body of a Structured Event is defined as an Any. This portion is intended to provide a convenient place to transmit any event data in addition to that which is viewed as interesting fields upon which consumers are likely to define filters. This portion is particularly suitable to store large blobs of data that are related to the event, such as contents of a file which was the cause of a **CorruptFile** event. Note that although this field is considered separate from the filterable data portion of the event, there is nothing to preclude an end-user from defining a filter based on the contents of this field.

The Structured Event is thus intended to provide a well-defined data structure into which a wide range of specific types of events can be mapped, and upon which optimized filtering and manipulating can be performed. This structure is particularly useful when used in concert with an event type repository which will completely describe the make-up of each type of event mapped into a structured event. As described in Section 2.8, "The Event Type Repository," on page 2-55, end-users can use this meta-data to construct filters which subscribe to new instances of structured events that are dynamically added to the system.

2.3 Event Filtering with Filter Objects

Undoubtedly the most important enhancement of the OMG Event Service introduced by the Notification Service is the enabling of each client to subscribe to the precise set of events it is interested in receiving. This feature is supported in the form of *filter* objects, each of which encapsulates a set of one or more constraints specified in a particular constraint grammar.

Each Admin and Proxy interface defined by the Notification Service inherits the **CosNotifyFilter::FilterAdmin** interface, which supports operations that enable the maintenance of a list of filter objects. Thus, each Admin and Proxy object within a Notification Service event channel can have associated with it one or more filter objects. These filter objects could be co-located in the same server process as a Notification Service event channel, or they can reside in their own address space[2].

There are two types of filter objects defined by the Notification Service: those that affect event forwarding decisions made by Proxy objects, and those that affect the way a Proxy object treats events with respect to certain QoS properties. The former type support the **CosNotifyFilter::Filter** interface and are described here. The latter type support the **CosNotifyFilter::MappingFilter** interface and are described in Section 2.3.1, "Mapping Filter Objects," on page 2-21.

Filter objects that affect the event forwarding decisions made by Proxy objects encapsulate a set of constraints. Each constraint is a data structure comprised of two components:

- A sequence of data structures, each of which indicates an event type.
- A string containing a boolean expression whose syntax conforms to some constraint grammar.

Each element in the sequence of data structures which each indicate an event type is comprised of a string field for the name of the domain within which the event type has meaning (e.g., “Telecom”), and a string field for the name of the specific event type within that domain to which the constraint applies (e.g., “CommunicationsAlarm”). This sequence contains the list of event types to which the subscription encompassed by a particular constraint applies. The second element in the constraint structure contains a boolean expression over the values of the contents of instances of the event types indicated in the first element of the same structure. Note that while there are no limits placed on the number of different constraint grammars supported by an implementation of the Notification Service, every implementation must support an implementation of the **CosNotifyFilter::Filter** interface that supports the grammar described in Section 2.4, “The Default Filter Constraint Language,” on page 2-23.

This two component data structure for the expression of each constraint encapsulated by a filter object is mainly provided for the convenience of both the end-user and the implementor of the Notification Service. From the end-user’s perspective, the structure allows for a short-hand notation for defining constraints which apply to one or more event types. For instance instead of supplying a constraint expression of the following form:

```
"(($domain_name == "Telecom" and $type_name == "Communica-
tionsAlarm")
or ($domain_name == "Transport" and $type_name == "RoadIm-
passable"))
and severity != 4"
```

the same constraint can be expressed as a two element structure as follows:

2.Note that while filter objects can reside in a separate address space from the proxy objects, each time an event is received by a proxy object the “match” operation of the filter is invoked to perform the filtering. Thus, there is a performance penalty paid when using remote filter objects, since each “match” invocation will result in a network communication as opposed to an intraprocess communication in the case of co-located filters.

```
{ [{"Telecom", "CommunicationsAlarm" }, {"Transport",
"RoadImpassable" }], "severity != 4" }
```

The above two constraints have the same meaning: they both subscribe to all events which are of either of the types indicated, and have a *severity* field within the contents of the event not equal to four. Notice that the convenience of this structure for constraint expressions becomes more obvious as the boolean expressions associated with the event types become more complex, and are applicable to more types of event.

From an implementor's perspective, this constraint structure provides facilities for the extracting of event type information from constraint expressions. This information is required in order to share event subscriptions between event channels and their clients as described in Section 2.6, "Sharing Subscriptions," on page 2-49.

Note that the convention is that an empty sequence of event type structures associated with a boolean constraint expression implies that the expression applies to all types of events, as does a single element in the sequence of event type structures in which both fields are the empty string. Also note that an end-user may choose to provide no event types in the sequence and then match on the **type_name** and **domain_name** fields in the constraint expression. However, if event types are specified in the sequence, then only these types will be matched, and any additional types that are specified using constraints may never be matched (since the constraint will only be evaluated if the types in the sequence match). When provided within an element of the sequence of event types contained in the first field of a constraint structure, either the domain or event type field can contain a string with the wildcard ("*") symbol indicating the boolean expression applies to any event whose type matches the indicated pattern. The "*" character may be expanded to zero or more characters, and may appear in any position in the string. As one would expect, a type element whose value is {"*", "*"} indicates that the boolean expression applies to all types of events.

Upon receipt of each event, each Proxy object within a Notification Service event channel invokes an appropriate **match** operation on each of its associated filter objects. A **match** operation accepts as input the contents of the event being filtered against, and returns a boolean result. The result returned will be TRUE if the event satisfies one or more of the constraints encapsulated by the filter object (i.e., OR semantics are applied between the constraints encapsulated by a filter object), and FALSE otherwise.

If the Proxy has multiple filter objects associated with it, it will invoke the **match** operation on each of its associated filter objects until either one returns TRUE, or all have returned FALSE (i.e., OR semantics are also applied between multiple filter objects associated with a given Proxy object). Upon receipt of an event at a given Proxy object, if the **match** operation of all filter objects associated with the Proxy evaluates to FALSE, the Proxy will discard the event. Otherwise, the event will be forwarded (to all proxy suppliers when the filtering is being performed by a proxy consumer, or to the associated consumer when the filtering is being performed by a proxy supplier)[3]. Note that this filtering by proxy objects is performed immediately upon receipt of each event by a proxy. If a given event passes a proxy object's filters and there are currently no other events queued for delivery by that proxy, the event will be forwarded immediately. Othewise, if there are other events waiting to be delivered by the proxy, the current event will be queued by the proxy for eventual delivery.

As previously stated, a set of filter objects can also be associated with each Admin interface within a Notification Service event channel. Recall that each Admin interface is responsible for the management of one or more Proxy objects. The set of filter objects associated with an Admin object thus applies to each Proxy object associated with that Admin. The set of filter objects associated with an Admin object can only be modified by invoking operations on the Admin object itself (and not on the individual Proxy objects managed by the Admin), and any such modifications affect all Proxy objects under the management of that Admin. Filter objects can be added to an individual Proxy object by invoking the **add_filter** operation directly on the Proxy object itself, and filter objects added in this manner affect only the particular Proxy upon which the operation was invoked. The set of filter objects added to an individual Proxy object in this manner can thus be modified by invoking operations directly on the Proxy.

The result of the semantics described in the previous paragraph are that each Proxy object can essentially have two sets of filter objects associated with it: those that are associated with its managing Admin object, and those that were added to it directly. Upon creation of each Admin object, a flag can be set which indicates whether each Proxy object created by the Admin will AND or OR the results of applying these two sets of filter objects when determining whether or not to forward each event (note that within each set, only OR semantics are applied in all cases; this flag only affects the operator used to combine the results of applying each of the two sets of filter objects to each event).

The main advantage of enabling filter objects to be associated with Admin objects is that end-users can define a single set of filters that apply to a group of Notification Service clients. Note that because all Proxy objects associated with a given Admin object essentially share the list of filter objects associated with the Admin, implementations of the Notification Service can optimize the filtering of a given event by a group of Proxies since each member of the group logically applies the same filters to the same event. Thus, the results of the evaluation of a given event against a given filter can be shared by all Proxy objects which are managed by a given Admin object.

A Proxy which has no filters associated with it (either by its Admin object, or through its own FilterAdmin interface) will pass through all events it receives. In the case of Proxy consumers, all events will be passed to the Proxy suppliers on its channel, and in the case of Proxy suppliers, all events will be delivered to its connected consumer.

It's worth noting that the **CosNotifyFilter::Filter** interface supports three styles of **match** operation: **match**, **match_structured**, and **match_typed**. The purpose of all of these operations is the same: take an event as input and evaluate it against the set of constraints encapsulated by the filter object. These operations differ only in the form

3. A minor variation of this algorithm occurs when a Proxy object has some locally defined filter objects, and some which it inherits from its parent Admin object, and the InterProxyGroupOperator flag is set to AND. In this case, the two sets of filter objects are ANDed together, as described shortly.

in which they accept the event as input. The **match** operation accepts an Any as input, and is thus invoked by a Proxy object upon receipt of an untyped event. The **match_structured** operation accepts a Structured Event data structure as input, and is thus invoked by a Proxy object upon receipt of a structured event. The **match_typed** operation is invoked by a Proxy object upon receipt of a typed event. The input parameter to this operation is a sequence of name-value pairs. How a typed event is parsed by the Proxy into the sequence of name-value pairs which is supplied as input to the **match_typed** operation is described in Section 2.7, “Filtering Typed Events,” on page 2-52.

Finally, note that the **CosNotifyFilter::Filter** interface supports an **attach_callback** operation. The purpose of this operation is to associate with each filter object an interface upon which the **subscription_change** operation should be invoked each time the set of constraints associated with the filter object is modified. The reason the filter object supports this feature is so that it can transparently (from the end-users’ perspective) notify event suppliers when the set of events being subscribed to by potential consumers of their events changes. The semantics of the **subscription_change** operation, and the rationale behind it, is explained in detail in Section 2.6, “Sharing Subscriptions,” on page 2-49.

The above discussion describes the semantics of the Notification Service filter objects whose purpose is to encapsulate constraints which affect the event forwarding decisions made by each Proxy object within a Notification Service event channel. The Notification Service also defines another type of filter object for use by consumers, the *mapping filter object*, whose rationale and semantics are described in the following subsection.

2.3.1 Mapping Filter Objects

The Notification Service recognizes two special properties of each event that could influence the delivery policy applied to the event: its *priority* and its expiration time (referred to here as its *lifetime*). While these properties are often populated by the supplier as fields of the event, there are many scenarios in which a consumer’s opinion of the relative importance of the event may differ from that of the supplier. In order to enable consumers to affect the priority and lifetime properties of events, the Notification Service introduces the concept of *mapping filter objects*.

Mapping filter objects support the **CosNotifyFilter::MappingFilter** interface. The specification of this interface looks very similar to that of the interface for regular filter objects. The main difference, however, is that mapping filters also associate a value with each constraint they encapsulate.

Each proxy supplier within a Notification Service event channel can have associated with it a mapping filter object which can affect the priority property of the events it receives, and another mapping filter object which can affect the lifetime property of the events it receives. The value associated with each constraint encapsulated by a mapping filter which affects events’ priority property is of type short, and represents an event priority. The value associated with each constraint encapsulated by a mapping filter which affects events’ lifetime property is of type **TimeBase::TimeT**, and represents a relative event lifetime.

Each mapping filter object can encapsulate one or more constraint-value pairs, and also has a default value associated with it. Upon receipt of an event by a proxy supplier with an associated mapping filter for the priority[4] property, the proxy supplier invokes the appropriate *match* operation on the mapping filter. The mapping filter proceeds to apply its encapsulated constraints in the order of highest to lowest with respect to the value associated with each constraint, until either the event satisfies a constraint or else does not satisfy all constraints. Upon encountering the first constraint which the event satisfies, the operation returns a result of TRUE, and an output parameter set to the value associated with the constraint. If the event satisfies none of the constraints associated with the mapping filter object, the result of the **match** operation will be set to FALSE, and the default value associated with the mapping filter object will be returned as the output parameter. Upon return from the operation, if the output parameter is TRUE, the proxy supplier treats the event with respect to its priority according to the return value, as opposed to a priority setting contained within the event. If the output parameter is FALSE, the proxy supplier will apply the following rules in order to determine the priority that should be associated with the event:

1. If there is a priority property set in the header of the event, that value will be used.
2. If there is no priority property set in the header of the event, but the event has inherited an associated priority by virtue of being processed by a proxy object (either the current proxy supplier or the proxy consumer which first received the event) that has an assoicated priority QoS property, that value will be used.
3. Otherwise, the output parameter returned by the **match** operation, which in this case is the default value of the mapping filter object, will be used.

Proxy suppliers with an associated mapping filter for the lifetime property proceed similarly to invoke the **match** operation on such a mapping filter. The behavior of the **match** operation for a mapping filter related to event lifetime is identical to that for a mapping filter related to priority; the only difference is in the type of the output parameter returned when a constraint is encountered which the event satisfies. In this case, the proxy supplier uses the output parameter as the lifetime property of the event.

Note that the results of applying a mapping filter to an event are used to modify the way in which a proxy supplier treats its copy of the event with respect to priority and lifetime, but not to modify the contents of the event itself. Even if the event contains priority and lifetime fields, these should not be modified as the result of applying a mapping filter to the event.

Notification Service style **ConsumerAdmin** interfaces can also have associated mapping filter objects. The semantics in this case are identical to those with regular filter objects: the mapping filters associated with a **ConsumerAdmin** object are

4.Note that the priority property is used as an example here to explain how mapping filters are applied. Similarly, the lifetime property could have been used in the example. All mapping filter processing rules that apply to priority mapping filters as explained in this paragraph also apply to lifetime mapping filters.

shared by all proxy suppliers being managed by that **ConsumerAdmin** object. Note, however, if a particular proxy supplier has a mapping filter associated with it, this overrides any mapping filter set for the same proxy on the **ConsumerAdmin** that manages that proxy supplier.

Finally, note that this specification uses mapping filters to affect the priority and lifetime properties of events. The **CosNotifyFilter::MappingFilter** interface, however, is generic enough to be applied to any property of an event. Implementations of the Notification Service can thus support as a value-added extension the application of mapping filters to other event properties besides priority and lifetime.

2.4 The Default Filter Constraint Language

This section describes the default filtering constraint language which must be supported by all conformant implementations of the Notification Service. Note that as described in the previous section, filters are supported in the Notification Service as objects which can be associated with Proxy or Admin objects. These filter objects may or may not be colocated with the same server in which the Notification Service event channel resides. Each filter object has associated with it one or more constraints which have meaning in a particular filtering constraint grammar. Implementations of the Notification Service may provide native support for any number of filtering constraint grammars, but each conformant implementation must, at a minimum, support the grammar described in this section. In addition, users of this service may implement their own filter objects external to the Notification Service event channel which may support a proprietary filtering constraint grammar. As long as such a filter supports the standard **match** operations with the appropriate signatures, the Notification Service event channel will be able to use them the same as filters which support the default grammar.

In essence, the default constraint grammar supported by any conformant implementation of the Notification Service is the standard constraint language defined by the OMG Trading Service, along with a few extensions. This section describes the rationale behind the proposed extensions, which essentially make the Trader Constraint Language more appropriate as a filtering constraint language for Notification. This section also provides a detailed specification of the extensions to the Trader Constraint Language defined for Notification. A complete BNF for the default Notification Service filtering constraint language is thus formed by supplementing the BNF defined in the OMG Trading Service specification with the extensions defined here.

2.4.1 Issues with the Trader Constraint Language

The following issues summarize deficiencies and ambiguities in the Trader Constraint Language which, without modification, make it difficult to use as a filtering constraint language for Notification. Included with each item is an indication of how the issue is addressed in Notification.

- The specification is ambiguous as to whether a numeric constant may have a leading plus sign. Section B.2.5 of the OMG Trading Service specification permits this; however, the BNF does not. The Notification Service resolves this ambiguity by explicitly permitting a leading plus sign.
- The grammar defines *<String>* as a sequence of zero or more *<TextChar>*s enclosed in single quotes. At compile-time, if an event type repository is unavailable, it may be impossible to distinguish between a *<String>* of length one and the numeric `char` data type. In these cases, the Notification Service implementation must determine the actual data type from context at run-time.
- The specification does not define operand order for the substring operator. The Notification Service treats the expression "*String1 ~ String2*" to mean "*String1* is contained within *String2*".
- Within a *<String>*, the grammar does not specify how to interpret undefined escape sequences. Alternately, the grammar permits two escape sequences and the meaning of a backslash is ambiguous until the following character is examined. Using the first interpretation, the Notification Service treats a backslash as the start of an escape sequence and removes it when followed by any undefined character sequence.[5]
- The language provides no mechanism for casting a *<Number>* or *<Ident>* to a specific type. This can be troublesome in expressions using mixed data types. For example, if '**$.one**' and '**$.two**' represent integers, the constraint "**2.5 * ($.one / $.two) > 1**" will yield **FALSE** since the division takes place using integer arithmetic (where the result is 0). section 2.4.3 details all arithmetic conversions for the Notification Service and resolves the aforementioned problem.
- The purpose of the grouping operator '**,**' is unclear; it is also not part of the BNF and has no specified operator precedence. Constraints written for the Notification Service must not use the comma operator.
- The specification defines a set of *<preference>* operators; these are not used by the Notification Service.
- The grammar defined by the specification is not inherently context free. Since the Notification grammar must be context free, any *<Ident>* that matches a constraint language keyword must be escaped with a backslash. The *<Ident>* BNF token has been updated to permit a leading backslash.

5.This emulates the behavior of most (if not all) C/C++ compilers.

2.4.2 Trader Constraint Language Extensions for Notification

In order to fully support event filtering on complex data types, several extensions to the Trader Constraint Language are defined. There are two basic types of extensions: those that allow the components of complex data structures to be referenced, and those that are considered *features* of the Notification Service implementation. The complete list of language extensions is as follows:

- The special token '$' is introduced to denote both the current event as well as any run-time variables. The current event, '$', is that on which the constraint expression is evaluated. The form '$<*Ident*>' is used to specify a run-time variable.
- The new symbol <*Component*> denotes a collection of named <*Ident*>s that may be joined with subscript, associative array, or structure member operators (all defined below).
- If <*Component*> refers to a named structure, discriminated **union**, or **CORBA::Any** data structure, then the structure member operator '.' may be used to reference its members.
- If <*Component*> refers to an array or sequence of elements, then the subscript operator '[<*Digits*>]' may be used to reference a specific element in said list (e.g., **array[2]** would reference the third element in the array).
- If <*Component*> refers to a name-value pair list, then the associative array operator '(<*Ident*>)' may be used to reference a specific value in said list (e.g., **nv(priority)**). This syntax is also used for positional notation in discriminated unions as described in Section 2.4.6, "Positional Notation and Intended Applications
- A <*Component*> has implicit members '_type_id' and '_repos_id'. The former identifies the unscoped IDL type name of the component (e.g., **mystruct._typeid == 'mystruct'**) and the latter returns the RepositoryId (e.g., **mystruct._repos_id == 'IDL:module/mystruct:1.0'**).
- If <*Component*> refers to an array or sequence of elements, then the implicit member '_length' refers to the number of elements in the list (e.g., **sequence._length**).
- If <*Component*> refers to a discriminated **union**, then the implicit member '_d' refers to the discriminator (e.g., **union._d**).
- A new boolean operator, 'default', is introduced to provide a means for checking whether a union has a default member that is active (e.g., `default union._d`).
- The 'exist' operator is extended for use on all implicit members of a <*Component*> (e.g., `(exist any._d and any._d == 50) or any == 50`).
- The 'in' operator is extended so that it may operate on a <*Component*>**.**
- The run-time variable '**$curtime**' is reserved; its meaning is current time of day, its data type is that of "**TimeBase::UtcT**" as defined in the OMG Time Service.
- A reserved run-time variable may be escaped by inserting a backslash between the dollar sign and the <*Ident*> (e.g., **$\curtime**).

- Any vendor-defined keywords must be of the form ':<*Ident*>:'. The colons prevent any new conflicts with event-specific **enum**s and also make these extensions easy to locate.

As stated above, a <*Component*> is a collection of named identifiers. Yet, multiple layers of encapsulation may not actually have identifier names associated with them. Fortunately, the constraint author need not be concerned with these unnamed layers. If an event type repository is in use, it will be able to supply the encapsulation information. Alternatively, when the run-time engine is responsible for pulling apart the event structure, it will encounter (and quietly pass over) these unnamed layers.

To make this concept more clear, consider the following event components:

```
Event . memA . Any . struct { int val, cnt; };
Event . memB . Any . Any . int;
Event . char;
Event . methA . ( char key, Any . int types[10] );
```

In the first example, the **struct** is encapsulated in the **CORBA::Any** named **memA**; to reference **cnt**, one would use '**$.memA.cnt**'. In the second example, an **int** is wrapped in an unnamed **CORBA::Any** and then again in **memB** (a named **CORBA::Any). ** Here, to reference the unnamed integer one would write '**$.memB**'. In the third example, a **char** is immediately wrapped in a **CORBA::Any** and sent through the channel; in this case, '**$**' alone represents the data. The last event consists of a method and its arguments; here, '**$.methA.types[3]**' identifies the 4th element in the 2nd argument to method methA.

As stated above, the constraint author need not be concerned about unnamed layers of encapsulation. This implies that it is possible to write a single constraint that will function on structured (typed or untyped) and unstructured events. For example, consider the constraint "**$.header.fixed_header.event_type.type_name == 'CommunicationsAlarm'**"; if the unstructured event included a '**header.fixed_header.event_type.type_name'** member, then both types of events could be filtered by the same proxy using this constraint.

A complete specification of the enhancements to the Trader Constraint Language BNF defined by the Notification Service can be found in Section 2.4.8, "Extensions to Trader Constraint Language BNF," on page 2-32.

2.4.3 Arithmetic Conversions for Mixed Data Types

In general, arithmetic conversions follow the "usual arithmetic conversion" rules set forth by C/C++. However, in the context of the Notification Service, it is not always possible to determine the data types of all operands at compile-time. Therefore, in order to simplify data conversion rules, most arithmetic operations are performed using either `CORBA::Long` or `CORBA::Double`. The result of each operation is then cast back to the data type of the most capacious of the operands, along with it's *weak* or *strong* type attribute (as described below).

The following rules then, govern mathematical operations with mixed data types.

- If either operand is a **CORBA::LongDouble**, the other is converted to **CORBA::LongDouble** and the result is **CORBA::LongDouble.**
- Otherwise, if either operand is a **CORBA::Double**, the other is converted to:
  - **CORBA::Double** and the result is **CORBA::Double.**
  - if either operand is a **CORBA::Float**, both operands are converted to **CORBA::Double**, but the result is **CORBA::Float.**
  - if either operand is a **CORBA::LongLong**, the other is converted to **CORBA::LongLong** and the result is **CORBA::LongLong**.
  - the most strongly-typed of the two operands becomes the result type, and both operands are converted to either **CORBA::Long** or **CORBA::ULong**.
- When:
  - a shorter unsigned type is combined with a larger signed type, the unsigned property does not propagate to the result type.
  - a numeric constant is specified, it is treated as weakly-typed **CORBA::Long** or, in the case of a floating point constant, a weakly-typed **CORBA::Double**.
  - a boolean operand is used in an arithmetic operation, it is treated as weakly-typed **CORBA::Long** with the values TRUE and FALSE corresponding 1 and 0, respectively.

Going back to the example constraint in Section 2.4.1, "Issues with the Trader Constraint Language," on page 2-23:

```
2.5 * ($.one / $.two) > 1
```

In order for this constraint to return TRUE, the parenthesized expression may be cast to floating point by rewriting it as:

```
2.5 * (1.0 * $.one / $.two) > 1
```

For the purpose of describing the operator restrictions, all operands may be classified as one of the following generic types: boolean, enum, numeric, string, or sequence. Numeric operands include boolean and strings of length one (i.e., *char*). Operator restrictions are as follows:

- The substring operator '**~**' may only be applied to string data types.
- The **in** operator may only be applied when the first operand is of a simple type and the second is a sequence of the same type.
- Comparison operations are valid only when both operands are either boolean, numeric, or string.
- Numeric operations are valid only on numeric types.
- For a divide operation, zero is invalid as a denominator.
- A numeric value may not be substituted when a boolean is required.
- Regarding the implicit members of a <*Component*>, '_length' is only valid for arrays or sequences, '_d' may only be used on discriminated unions, and '_type_id' and '_repos_id' are only valid if said information can be obtained.
- The **default** operator may only be applied to a discriminated union. If a discriminated union does not have a default member, this operator returns `FALSE`.

- Only equality and inequality operations (==, !=, >=, <=, >, or <) can be applied to enums.

When first handed a constraint, the Notification Service can only guarantee that it is syntactically correct. It is only when events are filtered, that it becomes possible to check that operands have valid data types. When invalid operands are encountered or when specified identifiers do not exist, the match operation must immediately return FALSE.

The implication of the above rule is that a Notification Service implementation run-time engine must implement short-circuiting of boolean '**and**' and '**or**' operations. Specifically, '**FALSE and <expression>**' must yield FALSE. Similarly, '**TRUE or <expression>**' must yield TRUE. In either case, it is not permissible to evaluate **<expression>**.

As an example, consider the following 4 events and the associated constraint:

```
Event 1: <$.a, 'Hawaii'>, <$.c, 5.0>
Event 2: <$.a, 'H'>, <$.c, 5.0>
Event 3: <$.a, 5>, <$.c, 5.0>
Event 4: <$.a, 5>, <$.b, 5.0>

Constraint: ($.a + 1 > 32) or ($.b == 5) or ($.c > 3)
```

For the first event, the first expression becomes (**'Hawaii' + 1 > 32**). Since it is not possible to add '**1**' to a string data type, the constraint is invalid and the match operation immediately returns FALSE.

In the second event, the first expression becomes (**'H' + 1 > 32**). Since '**H**' is a valid **char** data type, this yields TRUE (for the ASCII character set) and the match operation immediately returns TRUE. Note that here, the fact that '**$.b**' is not part of the event is immaterial due to the defined short-circuit semantics.

For the third event, the first expression yields FALSE and the second expression can not be resolved (since there is no '**$.b**' member in the event). This is an error, so the match operation immediately returns FALSE. Note that, the constraint author could have dealt with the possibility of a missing '**$.b**' by rewriting the constraint as:

```
($.a + 1 > 32) or (exist $.b and $.b == 5) or ($.c > 3)
```

In the fourth event, the first expression again yields FALSE, but this time '**$b**' is defined as a floating point '**5.0**'. Following the arithmetic conversion rules, the constant '**5**' is also cast to floating point and the second expression yields TRUE. Here, the match operation returns TRUE even though the event has no '**$.c**' member.

2.4.4 Support for Name-Value Pairs

The Notification Service makes extensive use of *name-value pair* lists within structured events. These are somewhat difficult to manage using the Trader Constraint Language because each member of the list must be treated as a complex structure (i.e., with both a name and value field), as in:

```
($.header.variable_header[1].name == 'priority' and
 $.header.variable_header[1].value > 1163) or
```

($.header.variable_header[2].name == ‘priority’ and
$.header.variable_header[2].value > 1163)

While the above syntax is correct, it is far more convenient to treat a name-value pair as an associative array such that, when given a name, one expects its value. To accomplish this, we extend the Trader Constraint Language to allow one to identify a component as being that of a name-value pair list. For example, ‘**$.header.variable_header(priority)**’ returns the value of **priority** in the **variable_header** name-value pair list.

2.4.5 A Short-hand Notation for Filtering a Generic Event

Section 2.4.2, “Trader Constraint Language Extensions for Notification,” on page 2-25 shows that it is possible to use a single constraint across both structured and unstructured events. However, for this to work, the layout of the filterable portion of the unstructured event must match that of the structured event. In order to relax these requirements, run-time variables may be employed as a short-hand notation for expressing commonly filtered data.

Specifically, any simple-typed member of **fixed_header** or any property in the name-value pairs **variable_header** and **filterable_data** may be represented as run-time variables. For example, the constraint:

```
$.header.fixed_header.event_type.type_name == 'CommunicationsAlarm' and
$.header.fixed_header.event_name == 'lost_packet' and
$.header.variable_header(priority) < 2
```

can be rewritten using run-time variables as:

```
$type_name == 'CommunicationsAlarm' and
$event_name == 'lost_packet' and $priority < 2
```

The following rules govern translation of a run-time variable, ‘**$variable**’, into a specific event field. If the run-time variable is reserved (e.g., **$curtime**) this translation takes precedence. If the run-time variable is $*domain_name*, $*type_name*, or $*event_name*, these are resolved to

$.header.fixed_header.event_type.domain_name,
$.header.fixed_header.event_type.type_name, or
$.header.fixed_header.event_name, respectively.

Next, the first matching translation is chosen respectively from properties in **$.header.variable_header**, and properties in **$.header.filterable_data**. If no match is found, the translation defaults to either **$.variable**., or in the case of a **CORBA::Any** that encapsulates a single unnamed name-value pair list (Section 2.4.4, “Support for Name-Value Pairs,” on page 2-28), **$(variable)**.

Given these rules, an unstructured event with a `$.priority` member and a structured event using `$.header.variable_header(priority)` can be specified in a generic constraint using the run-time variable '`$priority`'. Alternatively, a constraint can be written specifically for a structured or unstructured event by avoiding the use of run-time variables.

2.4.6 Positional Notation and Intended Applications

CORBA does not require that the names of IDL type members be marshalled into the TypeCode of a **CORBA::Any**. This implies that a filter that matches on named parts of an unstructured event will fail if the **CORBA::Any** was generated by an ORB that does not populate these fields. The population of a TypeCode's RepositoryId is also optional, so one can not depend on looking names up in the Interface Repository either.

To resolve this issue, the Notification Service permits constraints to be written in a purely positional notation which can be used to extract the same data as the traditional name-based filter expressions. For example, the constraint:

```
$.gpa < 80 or $.tests(midterm) > $.tests(final) or
$.monthly_attendance[3] < 10
```

might be rewritten using positional notation as:

```
$.3 < 80 or $1.(midterm) > $.1(final) or $.2[3] < 10
```

Except for discriminated unions, the translation of a constraint using identifiers to one that uses positional notation is idempotent. In the case of `structs` and `enums`, the members are indicated by their position starting from zero. For example, consider the IDL:

```
struct X {
   long   A;
   string B;
   short  C;
};
enum P { Q, R, S };
```

In '`struct X`', member 'A' is denoted by '0', 'B' by '1' and 'C' by '2'. Similarly, in '`enum P`', 'Q' is denoted by '0', 'R' by '1' and 'S' by '2'.

Describing `unions` using positional notation is more complicated because the order of members is not significant, rather, members are indexed by label value. Therefore here, the "positional" notation for `unions` is really an index notation. The grammar defines the *<UnionVal>* literal token to collect all possible discriminator types and uses *<UnionPos>* to disambiguate this special case. For example, consider the IDL:

```
union K switch (short) {
   case 0:
   case 2:  string K;
   case 3:  X L;
   case 5:  long   M;
   default: short  N;
};
```

The member 'M' is denoted as '(5)' and the constraint over an unstructured event comprised of a '**union K**' that read "**$.M < 54**" is translated into positional notation as "**$.(5) < 54**". A constraint involving the 'C' member of the 'L' member of the '**union X**', for example, "**$.L.C < 128**" would be translated as "**$.(3).2 < 128**".

The member 'K' can be denoted using either '(0)' or '(2)', as in "**'putty' ~ $.(2)**". Note that the label is chosen independent of the actual discriminator. Therefore, either of the following expressions will match a **union** with a discriminator value of 2, where the string contained in the **union** is not 'hoob':

```
$._d == 2 and $(0) != 'hoob'
$._d == 2 and $(2) != 'hoob'
```

The last case is that of member 'N', indexed by the default label. This is translated as '()'. For example, the constraint "**$.N == 999**" is translated as "**$.() == 999**".

The semantics of the **exist** operator is also special for discriminated unions. In the case of any other data type, the assertion that a member name exists is sufficient assurance that the value associated with that member may be accessed. For unions, this is only true when the discriminator is set to the corresponding case. Therefore, the expression "**exist $.K**" will return TRUE if and only if the event TypeCode contains the member name information to identify 'K' and the union discriminator has the value 0 or 2. The label value notation is somewhat simpler as the expression "**exist $.(0)**" will return TRUE if and only if the discriminator is set to 0. This implies that the translation of "**exist $.K**" is "**exist $.(0)** or **exist $.(2)**". It also means that the expression "**exist $.(0)**" is equivalent to "**$._d == 0**".

2.4.7 Examples of Notification Service Constraints

This section provides annotated examples of constraints written in the Extended Trader Constraint Language defined by the Notification Service. The following examples intend to show the flexibility of this language.

- Accept all "CommunicationsAlarm" events but no "lost_packet" messages.
  ```
  $type_name == 'CommunicationsAlarm' and not
  ($event_name == 'lost_packet')
  ```
- Accept "CommunicationsAlarm" events with priorities ranging from 1 to 5.
  ```
  $type_name == 'CommunicationsAlarm' and
  $priority >= 1 and $priority <= 5
  ```
- Select "MOVIE" events featuring at least 3 of the Marx Brothers.

```
$type_name == 'MOVIE' and
   (('groucho' in $.starlist) + ('chico' in $.starlist) +
   ('harpo' in $.starlist) + ('zeppo' in $.starlist) +
   ('gummo' in $.starlist)) > 2
```

- Accept only *recent* events (e.g., generated within the last 15 minutes or so).

```
$origination_timestamp.high + 2 < $curtime.high
```

- Accept students that took all 3 tests and had an average score of at least 80%.

```
$.test._length == 3 and
   ($.test[0].score + $.test[1].score + $.test[2].score) / 3
   >= 80
```

- Select processes that exceed a certain usage threshold.

```
$.memsize / 5.5 + $.cputime * 1275.0 + $.filesize * 1.25
   > 500000.0
```

- Accept events with a default union discriminator set to the value 2.

```
default $._d and $.defvalue == 2
```

- Accept events where a threshold has the unscoped type name 'short'.

```
exist $threshold._type_id and $threshold._type_id == 'short'
```

- Accept only Notification Service structured events.

```
$._repos_id == 'IDL:CosNotification/StructuredEvent:1.0'
```

- Accept events with a serviceUser property of the correct standard type.

```
$violation(serviceUser)._repos_id ==
   'IDL:TelecomNotification/ServiceUserType:1.0'
```

- Accept only those events that have a specified security "rights list".

```
exist $.header.variable_header(required_rights)
```

- Accept events whose 'in' enum is set to the value 'HOUSE' or 'CAR'.

```
$.\in == HOUSE or $.\in == CAR
```

2.4.8 Extensions to Trader Constraint Language BNF

This section details Notification Service extensions to the Trader Constraint Language BNF as defined in appendix B of the OMG *Trading Object Service* specification.

The new boolean operator `default` has the same precedence as the `exist` operator. The new structure member operator '.' has the highest precedence.

The lexical token <*factor*> now accepts:

```
|   + <Number>
|   exist $ <Component>
|   $ <Component>
|   default $ <Component>
```

The lexical token <*expr_in*> now accepts:

```
|   <expr_twiddle> in $ <Component>
```

The lexical token <*Ident*> now accepts:

```
                  |    \ <Leader> <FollowSeq>
```

The following additional lexical tokens are also defined:

```
<Component>    :=  /* empty */
               |   . <CompDot>
               |   <CompArray>
               |   <CompAssoc>
               |   <Ident> <CompExt> /* run-time variable */

<CompExt>      :=  /* empty */
               |   . <CompDot>
               |   <CompArray>
               |   <CompAssoc>

<CompDot>      :=  <Ident> <CompExt>
               |   <CompPos>
               |   <UnionPos>
               |   _length
               |   _d
               |   _type_id
               |   _repos_id

<CompArray>    :=  [ <Digits> ] <CompExt>

<CompAssoc>    :=  ( <Ident> ) <CompExt>

<CompPos>      :=  <Digits> <CompExt>

<UnionPos>     :=  ( <UnionVal> ) <CompExt>

<UnionVal>     :=  /* empty */
               |   <Digits>
               |   - <Digits>
               |   + <Digits>
               |   <String>
```

The Notification Service uses the ASCII character set and adopts the same terminal symbols defined in Section B.2.3 of the OMG Trading Service specification. For Notification, the <*Special*> terminal symbol is the set of IDL escape sequences defined in Section 3.2.5 of the CORBA Specification.

A finite state automaton for <*Component*> is shown in Figure 2-3 on page 2-34. Dashed lines represent transitions on any or no input symbol. Also note that, by definition, the <*Ident*> state prohibits identifiers that match constraint language keywords.

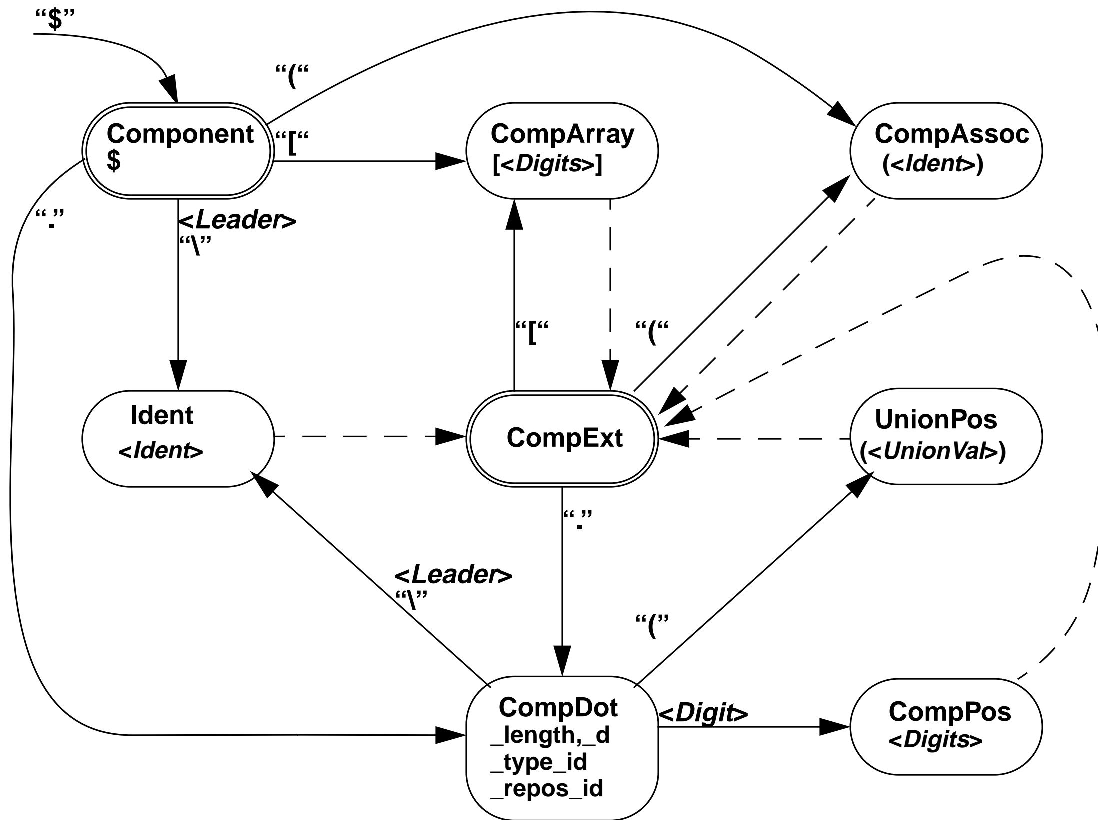

Figure 2-3 The finite automaton for *<Component>*

2.5 *Quality of Service Administration*

The existing OMG Event Service deliberately leaves the issue of Quality of Service as an implementation choice. Section 4.1.6 of the Event Service specification (formal/97-09-13) states:

> *“Note that the interfaces defined in this chapter are incomplete for implementations that support strict notions of atomicity. That is, additional interfaces are needed by an implementation to guarantee that either all consumers receive an event or none of the consumers receive an event; and that all events are received in the same order by all consumers.”*

The Notification Service extends the Event Service, by defining standard interfaces for controlling the QoS characteristics of event delivery.

This section describes each of the components in the Quality of Service (QoS) model and their relationships.

2.5.1 Model Components

The QoS abstract model consists of the following components:

- QoS property representation
- accessor operations for setting and getting QoS at various levels of scope throughout an application:
  - Notification Channel.
  - Supplier/Consumer Group Administration
  - Proxy suppliers and consumers
  - individual event messages
- QoS properties for notification
- negotiating QoS and conflict resolution

2.5.2 QoS Property Representation

A variety of QoS properties, such as reliability and priority, may be expressed to indicate the delivery characteristics of event messages. A particular property may have a range of values that indicate different requirements or delivery characteristics. The precise QoS requirements, at a particular level, can be expressed as a set of properties.

This specification defines a number of QoS properties and their permitted types and value ranges. However, it is clear that whatever choices of properties and their permitted values are, it is not possible to cover all use cases. Therefore a core design principle is to enable implementors to extend the properties understood by a Notification Service implementation, and to facilitate simple evolution of QoS properties. To this end, this specification uses Properties (<String, Any> pairs) to define QoS properties. Note that a special data type, **CosNotification::PropertySeq** is defined in this specification to represent property lists. While it is straightforward to declare and use a structurally equivalent data type wherever properties are called for, it should be noted for purposes of type safetyness that wherever this specification refers to property sequences or name-value pair lists, it explicitly means the **CosNotification::PropertySeq** data type.

The Notification Service defines a number of standard property names, and defines the expected type and value range that should be contained in the associated Any.

Implementations of the Notification Service are expected to understand all properties defined in this specification (however they need not *implement* the full range of qualities of service that these properties are capable of representing). Implementations may also add to the set of properties understood by the service as vendor additions, although doing so may restrict interoperability and portability.

2.5.3 Setting QoS

Programmers can set QoS at various levels of scope by creating a *QoSProperties* sequence and selecting the interface for the particular scope. Accessor operations (**get_qos**, and **set_qos**) are available at the following scope levels:

- notification channel
- admin objects
- individual proxy objects

In addition, for Structured Events, QoS properties can be set in the optional header field on a per-event basis.

These levels of scope form a simple hierarchy, reflecting the ability to override QoS at various levels. QoS set at the notification channel sets the default QoS requirements for message delivery for all groups, proxies, and messages. Setting QoS at the proxy group administration level overrides the notification channel level QoS for all proxies which are a member of the group. Setting QoS at the individual proxy level overrides the group admin or notification channel level settings, and setting QoS within a particular message overrides any other QoS setting (note, however, that per-message priority and lifetime settings can be overridden by use of a mapping filter, as explained in Section 2.3.1, “Mapping Filter Objects,” on page 2-21).

The actual set of QoS properties that should be applied to a component is derived from merging the set of properties passed within the invocation of the operation on the factory that creates the component (for those creation operations that permit this) with the properties currently set on the “parent” component (with respect to the newly created component). In cases where a property is set in both palces, the property value explicitly expressed within the creation operation invocation applies.

In general, QoS properties can be passed to a factory when creating a component. The actual set of properties that should be applied by a component is derived from merging the set of properties passed to the factory with the properties of the component in the higher scope level. In cases when a property is present in both places, the property value explicitly expressed in the lower level applies. Note that this is only a conceptual merge, since changing a QoS property in one scope should be reflected in the lower scope.

It may not make sense to allow all properties to be overridden at all levels. For example, setting reliable delivery at a message level may not make sense if the channel level has only been set to best-effort, as the setting at the channel level may have resulted in the use of an implementation that does not support reliability.

2.5.4 End-to-End QoS

When suppliers and consumers are connected together via a channel, there are three conceptual points where a message may be transmitted: between the supplier and the channel, within the channel, and between the channel and the consumer. In such a supplier/consumer model, where there is no direct communication between the two ends, it is not possible to set QoS in one place that covers the complete path from supplier to consumer. Instead QoS must be set at the individual points that make up the path. This introduces the problem of consistent QoS across a path. For example, both the supplier and the channel may have QoS set to reliable delivery, but a consumer sets the QoS for its individual proxy supplier to best-effort. Without the cooperation of all three parties it is impossible to guarantee a QoS requirement.

Unfortunately it is not possible to solve this problem, due to the nature of the service, and hence it is the user's responsibility to ensure that QoS is consistent across the whole path.

2.5.5 *Notification QoS Properties*

The Properties that have been defined for the Notification Service are defined below.

2.5.5.1 *Reliability*

There are a variety of delivery policies known in distributed systems, such as best-effort, at-least-once, at-most-once, and exactly-once. However most of these only make sense in a point-to-point, request-reply communication model. The Notification Service is by definition a point-to-multipoint delivery mechanism with no explicit reply mechanism.

The Notification Service treats the reliability of specific events, and the reliability of the connections which provide a transport for events between clients of the notification channel and the channel, as separate issues, and thus defines two separate QoS properties to represent them: *EventReliability* and *ConnectionReliability*. Each of these properties can take on one of two possible numeric constant values: *BestEffort* or *Persistent*. The meanings associated with the settings of these properties are inter-related, and are thus defined together below.

EventReliability=BestEffort & ConnectionReliability=BestEffort: No specific delivery guarantees are made. In the presence of failures, the event may or may not be received by each of the consumers, and a given consumer may receive the same event multiple times.

EventReliability=BestEffort & ConnectionReliability=Persistent: The notification channel will maintain all information about its connected clients persistently, implying that connections will not be lost (logically) upon failure of the process within which the notification channel is executing. Any clients which connect to the channel using persistent object references may fail, but unless these object references raise an OBJECT_NOT_EXIST exception, the channel will continue to retry using them. Clients which then re-instantiate objects with these references will (logically) reconnect to their associated proxies. The channel will not, however, store any buffered events persistently. The implication of this combination is that upon restart from a failure of the notification channel server process, the channel will automatically re-establish connections to each of its clients, but will not attempt to retransmit any events that had been buffered at the time the failure occurred.

EventReliability=Persistent & ConnectionRelability=BestEffort: This combination has no meaning and need not be supported by a conformant Notification Service implementation.

EventReliability=Persisent & ConnectionReliability=Persistent: Each event is guaranteed to be delivered to all consumers registered to receive it at the time the event was delivered to the channel, within expiry limits. If the connection between the channel and a consumer is lost for any reason, the channel will persistently store any

events destined for that consumer until either each event times out due to expiry limits, or the consumer once again becomes available and the channel is subsequently able to deliver the events to all registered consumers. In addition, upon restart from a failure the notification channel will automatically re-establish connections to all clients that were connected to it at the time the failure occurred.

Note that the *ConnectionReliability* QoS property can be set at the channel, group admin, and proxy levels. This property has the special characteristic, however, that it may not be set to *Persistent* on an object whose parent object in the hierarchy has this property set to *BestEffort*. For example, it is not possible (or meaningful) to set *ConnectionReliability=Persistent* at the group admin level, if at the channel level *ConnectionReliability* has been set to *BestEffort*.

Note also that the validity of modifications to the reliability quality of service properties depends on the state of the objects within the channel within which the modification is being attempted. In general, reliability settings may not be modified after a “child” object has been created. Such modifications may violate the quality of service validation rules of the notificaiton service.

Specifically,

- modifying the event channel’s *ConnectionReliability* setting through *set_qos* after ConsumerAdmin or SupplierAdmin objects have been created is invalid, and
- modifying the *ConsumerAdmin*’s or SupplierAdmin’s *ConnectionReliability* setting through *set_qos* after child proxy objects have been created is invalid.

One exception to the above rules is the case of an event channel that has not had any new admin or proxy objects associated with it yet, and whose default admin objects have not yet been accessed. Every event channel has one default ConsumerAdmin and one default SupplierAdmin objects associated with it. However, these objects can be viewed as being non-existent prior to the first invocation performed by a client to access them. During that time when the default admins are (at least virtually) non-existent, the *ConnectionReliability* of the channel may be modified, and this will affect the *ConnectionReliability* of the default admins. Once either of the two default admins has been accessed by a client, the current setting of the *ConnectionReliability* of the associated channel essentially becomes permanent, and also applies to both default admin objects.

Also note that the *EventReliability* property can be set per-channel, or per-message to override the per-channel setting. Since all objects within the channel must cooperate to assure an *EventReliability* setting of *Persistent* is satisfied, it makes no sense to set some of them to *Persistent*, and some to *BestEffort*. Thus, *EventReliability* cannot be set on an individual Admin or Proxy basis.

2.5.5.2 Priority

The event service does not define the order in which events are delivered to a consumer. One way to be explicit is to allow delivery to be based on the priority of an event. Priority is represented as a short value, where -32,767 is the lowest priority and 32,767 the highest. The default priority for all events is 0. By default, the notification channel will attempt to deliver messages to consumers in priority order.

It is possible for a consumer to override the priority assigned to a message through the use of mapping filters (see section 2.3.1).

2.5.5.3 Expiry times

It is often desirable to indicate the time range in which an event is valid. If an event is not delivered within a specified time then it should be discarded.

There are two possible properties related to expiry times that can be expressed:

StopTime, a **TimeBase::UtcT** encoded value, states an absolute expiry time (e.g., January 1, 2000), after which the event can be discarded.

Timeout, a **TimeBase::TimeT** encoded value, states a relative expiry time (e.g., 10 minutes from now), after which the event can be discarded. It is possible for a consumer to override the value associated with this property through the use of mapping filters (see Section 2.3.1, "Mapping Filter Objects," on page 2-21). Note that the time value associated with the *Timeout* QoS property is viewed as relative to the time when the channel (i.e., the receiving proxy consumer) first received the event.

Note that *StopTime* can only be used in the manner indicated above on a per-message basis, and thus has an associated UtcT value only when supplied as a QoS property within the header of a Structured Event. At other levels where QoS can be set (i.e., Proxies, Admins, and Channels), a *StopTimeSupported* QoS property is defined which has an associated boolean value, indicating whether or not the setting of *StopTime* on a per-message basis is supported.

2.5.5.4 Earliest Delivery Time

It is often desired that an event be held until at least a specified time, and become eligible for delivery only after that time. *StartTime*, a **TimeBase::UtcT** encoded value, states an absolute earliest delivery time (e.g., January 1, 2000), after which the event can be delivered.

Note that *StartTime* can only be used in the manner indicated above on a per-message basis, and thus has an associated UtcT value only when supplied as a QoS property within the header of a Structured Event. At other levels where QoS can be set (i.e., Proxies, Admins, and Channels), a *StartTimeSupported* QoS property is defined which has an associated boolean value, indicating whether or not the setting of *StartTime* on a per-message basis is supported.

2.5.5.5 Maximum Events Per Consumer

As described in Section 2.5.7, “Notification Channel Administrative Properties,” on page 2-48, an administrative property can be set on the channel to bound the maximum number of events a given channel is allowed to queue at any given point in time. Note, however, that a single badly behaved consumer could result in the channel holding the maximum number of events it is allowed to queue for an extended period of time, preventing further event communication through the channel. Thus, the *MaximumEventsPerConsumer* property helps to avoid this situation by bounding the maximum number of events the channel will queue on behalf of a given consumer. If set only on a per-channel basis, the value of this property applies to all consumers connected to the channel. If set on a per-ConsumerAdmin basis, this property applies to all consumers connected to proxy suppliers created by that ConsumerAdmin. If set on a per-proxy supplier basis, this property applies to the consumer connected to the given proxy supplier. Note that setting this property on a SupplierAdmin or proxy consumer has no meaning. Also note that the default setting of this property is 0, meaning that the proxy imposes no limits on the maximum number of events that may be queued for its consumer.

Order Policy

This QoS property sets the policy used by a given proxy to order the events it has buffered for delivery (either to another proxy or a consumer). Constant values to represent the following settings are defined:

AnyOrder - Any ordering policy is permitted.

FifoOrder - Events should be delivered in the order of their arrival.

PriorityOrder - Events should be buffered in priority order, such that higher priority events will be delivered before lower priority events.

DeadlineOrder - Events should be buffered in the order of shortest expiry deadline first, such that events that are destined to timeout soonest should be delivered first.

Note that this property has no meaning if set on a per-message basis.

Discard Policy

This QoS property enables a user of the Notification Service to specify in what order the channel or a proxy supplier should begin discarding events in the case of an internal buffer overflow. This property applies on a per-channel basis only if it is set on a channel that has the **RejectNewEvents** admin property (defined in Section 2.5.7, “Notification Channel Administrative Properties,” on page 2-48) set to FALSE. If set on such a channel, the chosen discard policy will be applied whenever a supplier attempts to send a new event to the channel, and the total number of events already queued within the channel is equal to the **MaxQueueLength** administrative property (defined in Section 2.5.7, “Notification Channel Administrative Properties,” on page 2-48). If set on a per-ConsumerAdmin basis, the chosen discard policy will be applied whenever the number of events queued on behalf of one of the consumers connected to one of the proxy suppliers created by the ConsumerAdmin exceeds the **MaxEventsPerConsumer** setting for that consumer. If set on a per-proxy supplier

basis, the chosen discard policy will be applied whenever the number of events queued on behalf of the consumer connected to the proxy supplier exceeds the **MaxEventsPerConsumer** setting for that proxy supplier. Note that in these latter two cases, an event will only be "discarded" with respect to its scheduled delivery to the consumer(s) on whose behalf the policy is being applied. In other words, if the event targeted for discarding is scheduled for delivery to any consumer(s) on whose behalf the discard policy was not invoked, the event remains queued for those consumers.

Constant values to represent the following settings are defined:

AnyOrder - Any event may be discarded on overflow. This is the default setting for this property.

FifoOrder - The first event received will be the first discarded.

LifoOrder - The last event received will be the first discarded.

PriorityOrder - Events should be discarded in priority order, such that lower priority events will be discarded before higher priority events.

DeadlineOrder - Events should be discarded in the order of shortest expiry deadline first.

Note that this property has no meaning if set on a per-message basis.

Maximum Batch Size

This QoS property has meaning in the case of consumers that register to receive sequences of Structured Events. For any such consumer, this property indicates the maximum number of events that will be delivered within each sequence. The data type associated with this property is *long*. The default setting for this property is 1, whereas an attempt to set it to 0 will result in the UnsupportedQoS (BAD_VALUE) exception being raised. Note that this property does not apply to Any or Structured Event style proxy objects. It only applies to Sequence style proxies, and the Admins and Channels that create them (so that the Sequence style proxies can derived a default value from these higher level objects).

Pacing Interval

This QoS property also has meaning in the case of consumers that register to receive sequences of Structured Events. For any such consumer, this property defines the maximum period of time the channel will collect individual events into a sequence before delivering the sequence to the consumer. If the number of events received within a given *PacingInterval* equals or exceeds *MaximumBatchSize*, the consumer will receive a sequence of events whose length equals *MaximumBatchSize*. Otherwise, the consumer will receive however many events arrived at the proxy supplier during the *PacingInterval,* unless no events have arrived during the *PacingInterval* in which case the sequence-style proxy supplier will wait for at least one event to arrive before forwarding the sequence to its consumer. The data type of the value associated with this property is **TimeBase::TimeT**. The default setting for this property is 0, meaning that the object upon which it is set will never forward a sequence of events whose

length is less than *MaximumBatchSize*. Note that this property does not apply to Any or Structured Event style proxy objects. It only applies to Sequence style proxies, and the Admins and Channels that create them (so that the Sequence style proxies can derive a default value from these higher level objects).

Note that setting certain QoS properties at a particular level is meaningless. For example, it makes no sense to allow **ConnectionReliability** to be specified on a per-message basis. The table below summarizes which QoS properties can be set at each level (an 'X' in a cell indicates that setting the property indicated by the first column in the row may be supported at the level indicated by the column heading).

Table 2-4 Levels at Which Setting Each QoS Property is Supported

Property	**Per-Message**	**Per-Proxy**	**Per-Admin**	**Per-Channel**
EventReliability	**X**			**X**
ConnectionReliability		**X**	**X**	**X**
Priority	**X**	**X**	**X**	**X**
StartTime	**X**			
StopTime	**X**			
Timeout	**X**	**X**	**X**	**X**
StartTimeSupported		**X**	**X**	**X**
StopTimeSupported		**X**	**X**	**X**
MaxEventsPerConsumer[1]		**X**	**X**	**X**
OrderPolicy		**X**	**X**	**X**
DiscardPolicy[1]		**X**	**X**	**X**
MaximumBatchSize[2]		**X**	**X**	**X**
PacingInterval[2]		**X**	**X**	**X**

1. Note that setting this property on a per-SupplierAdmin or per-proxy consumer basis has no meaning.
2. At the proxy-level, this property only applies to Sequence-style proxies.

2.5.6 Negotiating QoS and Conflict Resolution

QoS is intended to be both broad-ranging and extensible. Not all implementations will support all possible Qualities of Service. Version updates and vendor-private extensions will also mean that some QoS properties, or property values, will be unsupported by some implementations. Therefore, a Notification Service client may be unable to obtain exactly its desired QoS, and may need to negotiate the QoS. The Notification Service provides several mechanisms related to QoS negotiation:

1. The **set_qos** operation establishes QoS properties on its target object (notification channel, proxy group admin, or individual proxy).
2. QoS properties can be inserted directly into the header of a structured event. Such properties apply only to that particular event.

3. The **get_qos** operation returns the current QoS properties for its target object (notification channel, proxy group admin, or individual proxy). This includes properties initialized from higher-level objects, and properties which were never explicitly set but have default values.
4. The **validate_qos** operation checks a potential QoS request to see if it would be supported, without actually changing the QoS settings. This operation is available for notification channels, proxy group admin objects, and individual proxies. If the request can be supported, this operation returns additional optional QoS properties, which could be added (if desired) to the given request.
5. The **validate_event_qos** operation is similar to **validate_qos**, but applies to QoS properties which are to be set in the header of a structured event. The operation is available only for proxy producers and consumers.
6. The UnsupportedQoS user exception is raised by certain operations, to indicate that a QoS input parameter has an invalid or unsupported QoS. This exception attempts to minimize negotiation effort, by returning a list of the offending properties and their supported ranges (if they are supported at all).
7. The BAD_QOS system exception can be raised, to indicate that a QoS property in the header of a structured event is invalid or unsupported.

The following sections will explain the use of these mechanisms, and provide some examples.

2.5.6.1 Use of set_qos

This is the principal way to set QoS in the Notification Service. QoS can be set at any of three levels: a notification channel, a proxy group administration object, or an individual proxy. If any of the requested QoS properties cannot be supported, this operation raises the *UnsupportedQoS* exception, and the target object is unchanged. To assist in negotiation, this exception provides feedback on how to fix the QoS request (Section 2.5.6.6, “UnsupportedQoS Exception,” on page 2-45).

set_qos applies its input argument as a series of *incremental changes* to any existing QoS of the target object. Values of existing QoS properties can be changed, and new QoS properties can be added. Any existing QoS properties not mentioned in the input to **set_qos** are unmodified.

When QoS is set on a notification channel, it changes only the channel, and not any existing proxy group administration objects which are subordinate to the channel. Likewise, when QoS is set on a proxy group administration object, existing proxies which are subordinate to that object are not changed. Such changes affect only the initial QoS of subordinate objects created *after* the change.

2.5.6.2 QoS in a Structured Event Header

For Structured Events, QoS can also be set by inserting QoS properties directly into the event header, without using the **set_qos** operation. The setting applies just to a particular event sent to the channel within a push or pull operation, and is not

remembered for future events. If the requested QoS cannot be supported, the *BAD_QOS* system exception (Section 2.5.6.7, “BAD_QOS System Exception,” on page 2-46) is raised. Like the use of **set_qos**, QoS properties in the event do not replace the QoS specified for the proxy; they incrementally change it.

Care should be taken when setting QoS in an event header, because the BAD_QOS exception may not provide details of any errors. Clients should verify such QoS requests in advance, by means of **validate_event_qos** or some other means.

2.5.6.3 *Use of get_qos()*

The **get_qos** operation can be used to determine the current QoS properties in effect for a notification channel, a proxy group admin object, or an individual proxy. It returns all properties and their values, including those initialized from higher levels, and those which were never explicitly set but have default values.

2.5.6.4 *Use of validate_qos*

The **validate_qos** operation has two uses:

1. It checks a QoS request to see if it could be supported in a **set_qos** operation, without actually changing the current QoS in the target object.
2. If the supplied QoS is supported, it returns additional QoS properties which could be optionally added as well. This may help a client interested in the range of supported QoS in a given situation.

If the requested QoS cannot be supported, this operation raises the UnsupportedQoS exception, which provides feedback in how to fix the problem (Section 2.5.6.6, “UnsupportedQoS Exception,” on page 2-45).

If the request *can* be supported, then **validate_qos** checks whether any other QoS properties could be specified as part of the same request. The operation returns these additional properties, with a supported range of values for each one. Each additional property and value range is strictly optional—the client can choose *any or all* of them to add to an actual **set_qos** request. For each chosen property, the client can select any value between the “low_val” and “high_val,” inclusive.

If a client has a rough idea of the desired QoS, the client should invoke **validate_qos** against a property list of QoS it definitely requires. Then, the client can examine the returned “available QoS,” to see other properties which can be optionally added to the QoS request.

However, the client should not rely on **validate_qos** to return every available QoS property. Only *strictly optional* properties are returned. Since any subset of them must be chooseable, all returned properties must be independent of one another. If two properties are interdependent—if support for one depends on the value of another—then neither of them may be returned by **validate_qos**. See Section 2.5.6.8, “Examples of validate_qos and validate_event_qos,” on page 2-46 for an example.

2.5.6.5 *Use of validate_event_qos*

The **validate_event_qos** operation has two uses:

1. It checks a QoS request to see if it could be supported in the header of a structured event, without actually sending the event.
2. If the supplied QoS is supported, it returns additional QoS properties which could be optionally added to the structured event as well. This may help a client interested in the range of supported QoS in a given situation.

If the requested QoS cannot be supported, this operation raises the UnsupportedQoS exception, which provides feedback in how to fix the problem (Section 2.5.6.6, “UnsupportedQoS Exception,” on page 2-45).

If the request *can* be supported, then **validate_event_qos** checks whether any other QoS properties could be specified as part of the same request. The operation returns these additional properties, with a supported range of values for each one. Each additional property and value range is strictly optional—the client can choose *any or all* of them to include in an actual structured event. For each chosen property, the client can select any value between the “low_val” and “high_val”, inclusive.

If a client has a rough idea of the desired QoS, the client should invoke **validate_event_qos** against a property list of QoS it definitely requires. Then, the client can examine the returned “available QoS,” to see other properties which can be optionally added to the QoS request.

However, the client should not rely on **validate_event_qos** to return every available QoS property. Only *strictly optional* properties are returned. Since any subset of them must be chooseable, all returned properties must be independent of one another. If two properties are interdependent—if support for one depends on the value of another—then neither of them may be returned by **validate_event_qos**. See Section 2.5.6.8, “Examples of validate_qos and validate_event_qos,” on page 2-46 for an example.

Use of **validate_event_qos** is particularly important, because it can avoid BAD_QOS system exceptions caused by inserting an unsupportable QoS request into an actual structured event. It may be difficult to recover from this system exception.

2.5.6.6 *UnsupportedQoS Exception*

Certain operations raise the UnsupportedQoS exception, when supplied with a QoS property list which cannot be supported. The exception returns a sequence of QoS properties and value ranges. Only properties from the request which were in error are

returned. Each returned property is accompanied by an error code, which identifies the problem with that property. The meanings of the possible error codes are described in the following table.

Table 2-5 Meanings Of UnsupportedQoS Error Codes

Error Code	**Meaning**
UNSUPPORTED_PROPERTY	This property is not supported by this implementation for this type of target object.
UNAVAILABLE_PROPERTY	This property cannot be set (to any value) in the current context.[1]
UNSUPPORTED_VALUE	The value requested for this property is not supported by this implementation for this type of target object. A range of values which *would* be supported is returned.
UNAVAILABLE_VALUE	The value requested for this property is not supported in the current context.[1] A range of values which *would* be supported is returned.
BAD_PROPERTY	This property name is unrecognized. The implementation knows nothing about it.
BAD_TYPE	The type supplied for the value of this property is incorrect.
BAD_VALUE	An illegal value is supplied for this property. A range of values which *would* be supported is returned.

1. “Current context” means in the context of other QoS properties.

The returned property value range is meaningful only for the UNSUPPORTED_VALUE, UNAVAILABLE_VALUE, and BAD_VALUE error codes; otherwise it should be ignored.

2.5.6.7 *BAD_QOS System Exception*

While the user exception UnsupportedQoS is the appropriate exception to raise for operations on the objects that comprise a notification channel that involve QoS property modifications, an exception must also be raised during the transmission of a Structured Event from a supplier to the channel whenever the QoS properties indicated in the header of such an event cannot be satisfied by the channel. For this situation, we propose the addition of the BAD_QOS system exception to the CORBA standard. This exception, which should be useful for other OMG standards (e.g., Messaging) and may even make sense for the ORB itself to raise in certain situations, is described in more detail in Section 2.10.1, “A New Standard Exception,” on page 2-56.

2.5.6.8 *Examples of validate_qos and validate_event_qos*

Note – The QoS property names in this section are purely for example, and may not represent the required set of QoS properties supported at any level.

Example 1: Setting QoS in a structured event.

Suppose there are exactly two QoS properties supported for structured event headers: *Timeout* and *StopTime*. These are independent of one another: all combinations are allowed. If we invoke **validate_event_qos** with input "Timeout=50", the request will be accepted (no UnsupportedQoS exception), and the operation might return:

"Additional QoS" returned by validate_event_qos(Timeout=50)		
Property	**Low_Val**	**High_Val**
StopTime	(Dinosaur Era)	(Armageddon)

This indicates that the client can add a StopTime (with any possible value) to the request if desired. Similarly, if we invoke **validate_event_qos** with input "StopTime=January31,1999", the request will be accepted, and the operation might return:

"Additional QoS" returned by validate_event_qos(StopTime=...)		
Property	**Low_Val**	**High_Val**
Timeout	0	99999

However, if we invoke **validate_event_qos** with input "EventReliability=Persistent", the operation will raise an UnsupportedQoS exception, and no "Additional QoS" will be returned.

Example 2: Two interdependent QoS properties.

Suppose there are only three possible QoS properties supported by some proxy. The properties are *ConnectionReliability*, *EventReliability*, and *Timeout*. **ConnectionReliability** and **EventReliability** each have two possible values, "BestEffort" and "Persistent", but they are not independent—the combination

<*ConnectionReliability*=BestEffort, *EventReliability*=Persistent>

is not supported. *Timeout* can have any positive time value, independent of the other two properties.

Suppose the current QoS setting for the proxy is

Property	Value
ConnectionReliability	Persistent
EventReliability	Persistent
Timeout	100 sec.

If we invoke **validate_qos** with input “Timeout=50”, the QoS will be accepted (no UnsupportedQoS exception), and the operation will return the following:

“Additional QoS” returned by validate_qos(Timeout=50)		
Property	**Low_Val**	**High_Val**
EventReliability	BestEffort	BestEffort

The only “additional QoS” returned is a change to **EventReliability**. **ConnectionReliability** is not returned, because it can’t be changed unless **EventReliability** is *also* changed (recall the unsupported combination of these two). All properties returned as “Additional QoS” must be optional and independent of each other.

On the other hand, if we invoke **validate_qos** with input “EventReliability=BestEffort,” the QoS will be again be accepted, but more options will be returned as “additional QoS”:

“Additional QoS” returned by validate_qos(EventReliability=BestEffort)		
Property	**Low_Val**	**High_Val**
ConnectionReliability	BestEffort	BestEffort
Timeout	0	99999

Why are more “additional QoS” options returned here? Once we change **EventReliability** to BestEffort, any value for **ConnectionReliability** will be supported. And Timeout will always be returned if the request did not already include it, since it is totally independent of the others. If the client wants to add to his QoS request, he can chose any **ConnectionReliability** value in the indicated range, or any Timeout value in the indicated range, or both (or neither).

2.5.7 Notification Channel Administrative Properties

The notification channel also supports the configuration of certain administrative properties. The following administrative properties, each of which has an associated value of type *long*, can be set on a notification channel:

- MaxQueueLength - The maximum number of events that will be queued by the channel before the channel begins discarding events (according to the Discard Policy QoS parameter, which is defined in Section 2.5.5, “Notification QoS Properties,” on page 2-37) or rejecting new events (depending on the setting of the RejectNewEvents admin property described below) upon receipt of each new event.
- MaxConsumers - The maximum number of consumers that can be connected to the channel at any given time
- MaxSuppliers - The maximum number of suppliers that can be connected to the channel at any given time

For all of these properties, the default value is zero, which means that no limit applies to that property.

In addition, the notification channel supports the **RejectNewEvents** administrative property. This value associated with this property is of type *Boolean*, where TRUE and FALSE have the following meanings:

- TRUE: When the total number of undelivered events within the channel is equal to **MaxQueueLength**, each pull-style proxy consumer will stop attempting to perform *pull* invocations on its supplier until the total number of undelivered events within the channel is decreased. In addition, attempts to push new events to the channel by push-style suppliers will result in the IMPL_LIMIT system exception being raised.
- FALSE: When the total number of undelivered events within the channel is equal to **MaxQueueLength**, attempts to pull new events to the channel by a pull-style proxy consumer, or to push new events to the channel by a push-style supplier will result in one of the currently queued undelivered events being discarded by the channel to make room for the new event. The discarded event will be chosen based on the setting of the DiscardPolicy QoS property.

2.6 Sharing Subscriptions

2.6.1 Sharing Subscriptions Between Channels and Clients

The flow of events through a Notification channel depends on the events supplied to the channel and the subscriptions from event consumers which match them (or cause them to be discarded). In order to convey end-to-end the knowledge of what is required from suppliers, and what might be produced by them, we introduce two complementary operations, **offer_change** and **subscription_change**. These operations are available on interfaces supported by channels and, due to the symmetry of design, also on the interfaces supported by the clients of channels.

2.6.2 Offer

The **offer_change** operation, provided by the **NotifyPublish** interface is supported by all Proxy Consumer interfaces and the **SupplierAdmin** interface, and may be supported by consumers of events. It has two paramaters: one for event types that are newly offered, and one for event types no longer offered. This operation is used by suppliers of events to indicate to the channel the new event types that they will supply, and to indicate event types that they will no longer supply. Channels will then aggregate the offers from all their suppliers. If a new or removed offer by a supplier changes the aggregate list of event types offered to the channel, the channel will in turn invoke the same operation on its consumers, informing those consumers of new event types available to them, or events types no longer offered.

Consumers can use offer information to consult the Event Type Repository to discover what property names and types the event type contains, and thus write well-formed subscription expressions for these types.

Consumers may also discover the current set of event types that a channel has been offered by its suppliers by calling the **obtain_offered_types** operation on their **ProxySupplier** interface.

Consumers which are only interested in a fixed set of events may choose to supply a nil object reference to the channel at connect time if they pull events from the channel. They may also return a NO_IMPLEMENT exception from the **offer_change** operation if they must support the interface which allows the channel to push events to them.

2.6.3 Subscription Change

The **subscription_change** operation is provided by the **NotifySubscribe** interface, which is supported by all Proxy Supplier interfaces and the **ConsumerAdmin** interface, and may be supported by suppliers of events. It is a means of relaying subscription information, in the form of required event types, back to the source of events. It has two parameters: one to specify event types that are required, and one to specify event types that are no longer required. Event Channels will aggregate the event types that their consumers require. If a new event type required by a consumer (or consumer group represented by a **ConsumerAdmin**) changes the aggregate list then the channel will inform its suppliers by their calling **subscription_change** operation indicating that a new type is required. Likewise, if change from a consumer removes an event type from the aggregate list of event types in the channel it will call **subscription_change** on it suppliers indicating that the type is no longer required.

Suppliers may also discover the current set of event types that consumers of a channel require by calling the **obtain_subscription_types** operation on the **ProxyConsumer** interface.

Suppliers that are not interested in the event types currently subscribed to will not invoke **obtain_subscription_types**, and will raise the NO_IMPLEMENT exception in their implementation of the **subscription_change** operation.

2.6.4 Notifications on Demand

One consequence of suppliers being informed of the event types that clients require is that they know which notifications are being consumed and which are not. This knowledge can be used by end-suppliers to influence which notifications they will produce. For instance, an operating system process watcher connected to a channel as a supplier is informed that only notifications of certain process watching types are being consumed. It might therefore choose to produce notifications only about those kinds of processes rather than producing notifications for all processes. Another example is where suppliers only generate notifications while there is an interested party. For instance, a consumer indicates an interest in CPU load statistics by subscribing to a particular event type. Intelligent suppliers would then begin to produce these statistics only for as long as a consumer was interested in the information. When no consumers are interested, the channel's aggregate list of event types will change, and the supplier will be notified via the **subscription_change** operation that these

notifications are no longer being consumed. The supplier could then stop generating its statistics until the relevant event type was received in **subscription_change** in the parameter which indicates added types.

In these examples the anonymity of event suppliers and consumers is maintained while enabling communication about notification requirements between them.

2.6.5 Obligations on Filter Objects

Filter objects support the operations **attach_callback** and **detach_callback**, which are used by Proxy Suppliers and ConsumerAdmins to provide and remove object references to their **NotifySubscribe** interfaces. Filter objects must call the references that are currently attached when the set of event types that their constraints require changes. This means that they must use the **EventTypeSeq** associated with their constraints as a parameter to an invocation of **subscription_change**. Implementations of Filters may choose whether to convey the event types for a constraint as the added parameter to **subscription_change** when the constraint is added and then as the removed parameter to **subscription_change** when that constraint is removed, or whether to maintain an aggregate of the event types that all its constraints require, and only call **subscription_change** when this changes.

Note that a Proxy or Admin object that is evaluating the suitability of an event that is not of one of the types that the filter has indicated that it requires may assume that the filter does not require this event and never call a match operation at that filter. See Section 2.6.6, “Special Event Types,” on page 2-51 about special event types that force matching.

A consumer connected to a proxy supplier may use the **subscription_change** operation instead of a filter object if it requires all events of one or more event types. However, once it adds a filter to the proxy, it must then interact with the filter only, and allow the filter to invoke **subscription_change**. An event channel will create an aggregate list of all the event types required by its consumers, which it will return as the result of **obtain_subscription_types**. It will also communicate any changes to the list to its suppliers by invoking their **subscription_change** operation.

2.6.6 Special Event Types

If a constraint expression potentially applies to any event type, then the special event type “%ALL” can be specified in a constraint’s event type sequence (with a domain name of “*” or the empty string). When the filter calls back **subscription_change** with this type it indicates to the channel that the filter wants to match on all event types.

Alternatively the event type and domain specified by any combination of empty strings or the “*” string is treated as the equivalent to “%ALL”, and upon receiving a requirement for this type the channel will send “%ALL” to its suppliers by calling their **subscription_change** operations.

The event type “%TYPED” is given by channels to the Structured Event representation of events that are supplied by a typed event supplier using an operation other then the push and pull operations (i.e., untyped) specified in this document. The domain field of these events must be set to the empty string. This event type allows a Structured Event supplier to create an event that is equivalent to the invocation of a typed operation using the mapping given in Table 2-2 on page 2-9. It also allows Structured Event consumers and event service style consumers to write filter constraints that match events which were delivered to the channel by a typed supplier.

Note – the leader characters ‘%’ and ‘*’ are not legal for type names stored in the Event Type Repository, and are chosen because they do not clash with any pre-existing event type naming schemes known to the authors.

2.7 Filtering Typed Events

The Notification Service defines a *typed* version of the notification channel that is analogous to the typed event channel defined in the OMG Event Service. The typed notification channel extends the architecture of the notification channel depicted in on page 2-3 by adding to it typed versions of the EventChannel, Admin, and Proxy interfaces. Essentially, a typed notification channel can be connected to by traditional untyped event service clients, notification service clients (which supply untyped events, Structured Events, or sequences of Structured Events), and typed event service clients (as defined by the OMG Event Service). The particular value-add of the typed notification channel is that it enables typed event service clients to realize the advantages of event filtering and configurable quality of service.

The typed notification service interfaces are defined in a separate IDL module, the **CosTypedNotifyChannelAdmin** module, which defines interfaces that are analogous to the untyped notification channel interfaces. The module defines a **TypedEventChannelFactory** interface which supports an operation for creating new typed notification channel instances. This operation accepts the same input parameters as the operation supported by the factory interface for an untyped notification channel:

- a list of initial QoS settings for the channel
- a list of initial administrative settings for the channel

A typed notification channel supports the **TypedEventChannel** interface defined in the **CosTypedNotifyChannelAdmin** module. This interface inherits from both the **CosNotifyChannelAdmin::EventChannel** interface and the **CosTypedEventChannelAdmin::TypedEventChannel** interface. The former inheritance enables a typed notification channel to support untyped notification channel Admin interfaces, which can in turn create untyped notification channel style Proxy interfaces. Essentially, this enables untyped notification channel clients to connect to a typed notification channel, if so desired. The latter inheritance enables backward compatibility to the typed event channel as defined by the OMG Event Service, analogous to the way the notification channel supports backward compatibility to the untyped event channel. The operations supported through inheritance from the

CosTypedEventChannelAdmin::TypedEventChannel interface can be used to create the Admin interfaces defined for the OMG Event Service version of the typed event channel.

In similar fashion as the notification channel, the typed notification channel supports operations that can be used to create typed notification service style Admin objects. These objects have unique identifiers associated with them, whereas OMG Event Service style typed Admin objects created by invoking the inherited operations do not.

The typed notification service style Admin interfaces inherit from both their untyped notification service and typed event service counterparts. The former inheritance enables instances supporting one of these interfaces to create untyped notification service style proxy objects, which can in turn be connected to by untyped notification service style clients. The latter inheritance enables creation of the OMG Event Service style typed proxy objects, which can be connected to by OMG Event Service style typed clients. Such clients can pass typed events through the channel, but do not realize the benefits of filtering or QoS configurability.

The typed notification service style Admin interfaces also support operations which can be used to create typed notification service Proxy objects. Such objects can be connected to by clients which send and receive typed events as defined by the OMG Event Service, and also support filtering and QoS configurability. Like their untyped counterparts, typed notification service style proxies are assigned unique identifiers upon creation, and can be administered in the same fashion. Exactly the same as the operations which create typed event service style proxies, the operations supported by the typed notification service style Admin interfaces accept a string input parameter which indicates either the typed interface it should use (to receive events in the case of a **TypedProxyPullConsumer** or to supply events in the case of a **TypedProxyPushSupplier**) or the typed interface it must support (to receive events in the case of a **TypedProxyPushConsumer** or to supply events in the case of a **TypedProxyPullSupplier**).

The authors of the OMG Event Service realized that there is no difference in the interfaces supported by a pull style consumer of typed events and a pull style consumer of untyped events, and likewise between a push style supplier of typed events and a push style supplier of untyped events. For this reason, they chose not to define new Proxy interfaces for connections between the channel and either typed push consumers (which connect to **ProxyPushSuppliers**) or typed pull suppliers (which connect to **ProxyPullConsumers**). While the same model could have been followed when defining the typed version of the notification channel's proxy interfaces, it was believed that this would have lead to more confusion for the end user than if special proxy interfaces for all styles of clients were defined. For this reason, the typed notification channel explicitly defines **TypedProxyPullConsumer** and **TypedProxyPushSupplier** interfaces, as well as **TypedProxyPushConsumer** and **TypedProxyPullSupplier**. Each Proxy interface defined for the typed notification channel has the following properties:

- It inherits from the appropriate base Proxy interface defined in the **CosNotifyChannelAdmin** module, which enables it to support filtering and QoS configurability.

- It inherits from the appropriate consumer or supplier interface defined by the OMG Event Service to enable a traditional OMG Event Service typed event channel client to connect to it.
- It supports an explicit "connect" operation to be invoked by its client in order to establish the connection.

It is believed that this model of explicitly defining all four styles of Proxy interface for the typed notification channel, each of which supports an explicit "connect" operation, will result in a typed channel that is more straightforward to use than the typed channel defined by the OMG Event Service, without significantly altering the programming model of the latter.

Clients of the typed notification channel specify the interface type that they wish to use for communication to the channel by supplying a "key" string parameter to the "obtain" operation supported by the typed Admin objects. In the case where the channel is the active participant (i.e., when interacting with pull model suppliers and push model consumers), the channel expects that the proxy object reference supplied to the "connect" operation may be narrowed to the interface type nominated by the "key." In the cases where the client of the channel is the active party, i.e. push suppliers and pull consumers, the client will be able to narrow the proxy returned from the "obtain" operation to the interface type that was supplied as the "key" to that operation.

Note that the clients of the typed notification channel support identical interfaces to those of clients of the typed event channel defined by the OMG Event Service, implying that the same rules apply to the operations supported by those clients' interfaces as those defined for clients of the OMG Event Service typed event channel. In the push model typed events will thus be transmitted to the typed notification channel using a strongly typed interface <*I*> which supports operations which take only input parameters and have a void return type. The equivalent interface to <*I*> for the pull model will be called Pull<*I*>. The Pull<*I*> interface must support two operations for every operation <*op*> in interface <*I*>. These operations are called pull_<*op*> which has return type void and try_<*op*> which returns a boolean. Their parameters are identical to those in <*op*> except that they are all out parameters rather than in paramters.

The base interface name <*I*> is equivalent to the name of an event type domain, and the base operation name <*op*> is the event type in that domain. Parameters to the supported operation(s) of <*I*> and Pull<*I*> form the contents of the typed event. Each filter object supports a **match_typed** operation which is used to perform filtering on typed events. This operation accepts as input a sequence of name-value pairs. Upon receipt of a typed event, the notification channel will disassemble the event into a name-value pair sequence, where each name is the name of an input parameter to the operation on the typed interface which was invoked to transmit the event to the channel, and the value is the value associated with the parameter. The first element of such a sequence will always have its name set to "event_type," and its associated value set to an event type structure containing the strings which are the name of the typed interface, and the name of the operation in that interface.

An example of an interaction using typed notifications which uses both push and pull models is as follows. The IDL interface Coffee is defined as

```
interface Coffee {
    void drinking_coffee(in string name, in long minutes);
    void cancel_coffee(in string name);
};
```

A typed push consumer for coffee notifications would need to provide a “key” interface name, “Coffee,” to the “obtain” operation on its typed consumer admin operation, and then when calling the “connect” operation on the returned proxy it would provide an object reference of a type that multiply inherits from the **PushConsumer** and **Coffee** interfaces, so that the channel can narrow to the coffee interface and begin invoking **drinking_coffee** and **cancel_coffee** operations.

A typed pull consumer for coffee notifications would supply the same key, “Coffee,” to its “obtain” operation, and then narrow the Proxy interface it receives as a result to the interface type:

```
interface PullCoffee {
    void drinking_coffee(out string name, out long minutes);
    boolean try_drinking_coffee(out string name, out long minutes);
    void cancel_coffee(out string name);
    boolean try_cancel_coffee(out string name);
};
```

After calling the “connect” operation on the proxy, to which it provides an object reference of type **PullConsumer** to allow the channel to inform it of disconnection, it can begin calling the operations of the **PullCoffee** interface.

2.8 The Event Type Repository

This specification defines an Event Type Repository as a value-added, optional feature which can be provided along with implementations of the Notification Service. The Event Type Repository is treated as optional since it is not required in order for an implementation of the Notification Service to operate correctly. An implementation of the Notification Service which provides an Event Type Repository may or may not use the Event Type Repository to perform run-time checking.

An Event Type Repository can provide significant advantages to end-users of the Notification Service. Such a repository would be populated with the meta-data which describes the structure of all known event types which may be supplied to an instance of a notification channel supported by the implementation. End-users can use this information to construct meaningful constraints which subscribe their applications to the specific types of events that will be supplied within a given installation of the service. In addition, an implementation of the Notification Service may choose to use the information in the Event Type Repository to perform type checking of the event properties referenced within constraints to ensure they are used appropriately in mathematical or boolean expressions.

The standard schema for the Event Type Repository is provided in Appendix A of this specification. As defined there, each event type in the repository is characterized by a name and a set of properties. New event types can be defined in terms of existing event types by either *importing* the properties of one or more pre-existing types, or by

inheriting a pre-existing type, or some combination of these. The new type's full name may be generated from a combination of its local name and the names of its base types, according to the naming scheme of the type's domain. The full name of any event type must be unique within its domain. The default domain is named by the empty string, and its types have a flat name space, that is their local name is the same as their full name, and each type's name must be unique.

Note that this scheme integrates naturally with the event naming scheme used by Structured Events. The fixed portion of each Structured Event includes a *domain_name* and a **type_name** field. The **domain_name** names a specific vertical industry (e.g., telecommunications, finance, health care, etc.) within which a given **type_name** has meaning. These fields are accepted as the parameters to the query functions of the Repository, and in combination they act as a key to uniquely identify any type in the repository. The properties of the particular event type defined in the Event Type Repository would then define the specific name-value pairs that would be present in an instance of that type of event. Thus when Structured Events are used in concert with the Event Type Repository, it is particularly convenient for consumers to learn of new types of events, and to discover the structure of their contents.

The schema of the Event Type Repository is defined in Appendix A using the Meta-Object Definition Language of the Meta-Object Facility (MOF) joint submission being developed by DSTC, Unisys, and other submitters.

2.9 Issues with Interoperability

It's important to note that this specification guarantees interoperability of only those implementations that comply with the following requirements:

- Support all standard optional header fields summarized in Table 2-3 on page 2-16.
- Support filter constraints expressed in the default constraint grammar described in Section 2.4, "The Default Filter Constraint Language," on page 2-23.

These requirements do not mean that implementations of the Notification Service cannot support user or vendor specific event header field names, constraint grammars, or event types. The implication here is that the use of such user or vendor specific capabilities is outside of the scope of this specification, and therefore interoperability of such features between different implementations is not guaranteed.

Modules and Interfaces 3

Contents

This chapter contains the following topics.

Topic	Page
"The CosNotification Module"	3-2
"The CosNotifyFilter Module"	3-9
"The CosNotifyComm Module"	3-28
"The CosNotifyChannelAdmin Module"	3-41
"The CosTypedNotifyComm Module"	3-84
"CosTypedNotifyChannelAdmin"	3-85

This section describes the semantic behavior of the interfaces which make up the Notification Service. Each IDL module is presented, along with a brief description of the purpose of the module. For each interface in the module, a brief description of its purpose is provided, along with an explanation of the semantics of each of its operations and attributes.

The Notification Service is defined in terms of the following IDL modules:

CosNotification - Defines the Structured Event data type, quality of service and administrative properties, and interfaces which are used to administer these properties.

CosNotifyFilter - Defines the interfaces for filters supported by the Notification Service.

CosNotifyComm - Defines supplier and consumer interfaces for basic notification communication.

CosNotifyChannelAdmin - Defines proxy, admin and channel interfaces for notification channels.

CosTypedNotifyChannelAdmin - Defines proxy, admin and channel interfaces for typed notification channels.

Each of these modules is defined in its own subsection as follows.

3.1 The CosNotification Module

The **CosNotification** module defines the Structured Event data type, along with a data type used for transmitting sequences of Structured Events. In addition, this module provides constant declarations for each of the standard quality of service (QoS) and administrative properties supported by all Notification Service implementations. Some properties also have associated constant declarations which indicate their possible settings. Finally, administrative interfaces are defined for managing sets of QoS and administrative properties.

```
module CosNotification {

    typedef string Istring;
    typedef Istring PropertyName;
    typedef any PropertyValue;


    struct Property {
        PropertyName name;
        PropertyValue value;
    };
    typedef sequence<Property> PropertySeq;


    // The following are the same, but serve different purposes.
    typedef PropertySeq OptionalHeaderFields;
    typedef PropertySeq FilterableEventBody;
    typedef PropertySeq QoSProperties;
    typedef PropertySeq AdminProperties;


    struct EventType {
        string domain_name;
        string type_name;
    };
    typedef sequence<EventType> EventTypeSeq;


    struct PropertyRange {
        PropertyValue low_val;
        PropertyValue high_val;
    };


    struct NamedPropertyRange {
```

```
        PropertyName name;
        PropertyRange range;
};
typedef sequence<NamedPropertyRange> NamedPropertyRangeSeq;


enum QoSError_code {
        UNSUPPORTED_PROPERTY,
        UNAVAILABLE_PROPERTY,
        UNSUPPORTED_VALUE,
        UNAVAILABLE_VALUE,
        BAD_PROPERTY,
        BAD_TYPE,
        BAD_VALUE
};


struct PropertyError {
        QoSError_code code;
        PropertyName name;
        PropertyRange available_range;
};
typedef sequence<PropertyError> PropertyErrorSeq;


exception UnsupportedQoS { PropertyErrorSeq qos_err; };
exception UnsupportedAdmin { PropertyErrorSeq admin_err; };


// Define the Structured Event structure
struct FixedEventHeader {
        EventType event_type;
        string event_name;
};


struct EventHeader {
        FixedEventHeader fixed_header;
        OptionalHeaderFields variable_header;
};


struct StructuredEvent {
        EventHeader header;
        FilterableEventBody filterable_data;
        any remainder_of_body;
}; // StructuredEvent
typedef sequence<StructuredEvent> EventBatch;


// The following constant declarations define the standard

// QoS property names and the associated values each property
// can take on. The name/value pairs for each standard property
// are grouped, beginning with a string constant defined for the
```

```
// property name, followed by the values the property can take on.

const string EventReliability = "EventReliability";
const short BestEffort = 0;
const short Persistent = 1;

const string ConnectionReliability = "ConnectionReliability";
// Can take on the same values as EventReliability

const string Priority = "Priority";
const short LowestPriority = -32767;
const short HighestPriority = 32767;
const short DefaultPriority = 0;

const string StartTime = "StartTime";
// StartTime takes a value of type TimeBase::UtcT.

const string StopTime = "StopTime";
// StopTime takes a value of type TimeBase::UtcT.

const string Timeout = "Timeout";
// Timeout takes on a value of type TimeBase::TimeT

const string OrderPolicy = "OrderPolicy";
const short AnyOrder = 0;
const short FifoOrder = 1;
const short PriorityOrder = 2;
const short DeadlineOrder = 3;

const string DiscardPolicy = "DiscardPolicy";
// DiscardPolicy takes on the same values as OrderPolicy, plus
const short LifoOrder = 4;

const string MaximumBatchSize = "MaximumBatchSize";
// MaximumBatchSize takes on a value of type long

const string PacingInterval = "PacingInterval";
// PacingInterval takes on a value of type TimeBase::TimeT

const string StartTimeSupported = "StartTimeSupported";
// StartTimeSupported takes on a boolean value

const string StopTimeSupported = "StopTimeSupported";
```

```
// StopTimeSupported takes on a boolean value

const string MaxEventsPerConsumer = “MaxEventsPerConsumer”;
// MaxEventsPerConsumer takes on a value of type long

interface QoSAdmin {

    QoSProperties get_qos();

    void set_qos ( in QoSProperties qos)
        raises ( UnsupportedQoS );

    void validate_qos (
        in QoSProperties required_qos,
        out NamedPropertyRangeSeq available_qos )
    raises ( UnsupportedQoS );

}; // QosAdmin

// Admin properties are defined in similar manner as QoS
// properties. The only difference is that these properties
// are related to channel administration policies, as opposed
// message quality of service

const string MaxQueueLength = “MaxQueueLength”;
// MaxQueueLength takes on a value of type long

const string MaxConsumers = “MaxConsumers”;
// MaxConsumers takes on a value of type long

const string MaxSuppliers = “MaxSuppliers”;
// MaxSuppliers takes on a value of type long

const string RejectNewEvents = “RejectNewEvents”;
// RejectNewEvents takes on a value of type Boolean

interface AdminPropertiesAdmin {

    AdminProperties get_admin();

    void set_admin (in AdminProperties admin)
```

```
raises ( UnsupportedAdmin);

}; // AdminPropertiesAdmin

}; // CosNotification
```

3.1.1 The StructuredEvent Data Structure

The **StructuredEvent** data structure defines the fields which comprise a Structured Event. The following subsections briefly describe each of these fields. A detailed description of Structured Events is provided in Section 2.2, “Structured Events,” on page 2-12.

3.1.1.1 Fixed Header

The following fields make up the fixed portion of the header of every Structured Event.

domain_name

The **domain_name** field contains a string which identifies the vertical industry domain (e.g., telecommunications, healthcare, finance, etc.) within which the type of event which characterizes a given Structured Event is defined. The definition of this field enables each vertical domain to define their own set of event types without worrying about name collisions with those defined by other vertical domains.

type_name

The **type_name** field contains a string which identifies the type of event contained within a given Structured Event. This name should be unique among all event types defined within a given vertical domain, which is identified by the **domain_name** field.

event_name

The **event_name** field contains a string which names a specific instance of Structured Event. This name is not interpreted by any component of the Notification Service, and thus the semantics associated with it can be defined by end-users of the service. This field can be used, for instance, to associate names with individual Structured Events which can be used to uniquely identify an instance of a particular type of Structured Event within a given installation of the Notification Service.

3.1.1.2 Variable Header

The remainder of the header of a Structured Event is contained within the **variable_header** field. The data type of this field is a sequence of name-value pairs, where each name is a string and each value a **CORBA::Any**. While this field can essentially contain any name-value pairs which users of the service deem to be useful

to provide in the header of a Structured Event, standard names and associated value types are defined in Table 2-3 on page 2-16 of this document. The standard variable header fields defined there provide QoS related information about the current Structured Event that should override other QoS settings within the channel when objects within the channel process the current Structured Event.

3.1.1.3 Body of a Structured Event

The body of a Structured Event is intended to contain the contents of an instance of a Structured Event being published by a Notification Service supplier. It's contents are broken down into two fields: the **filterable_data** and the **remainder_of_body**. The purpose of each of these fields is defined below.

filterable_data

The **filterable_data** portion of the body of a Structured Event is a sequence of name-value pairs, where name is of type **string** and the value is a **CORBA::Any**. The main purpose of this portion of the event body is to provide a convenient structure into which event body fields upon which filtering is likely to be performed can be placed. It is anticipated that mappings of standard event types to the Structured Event will be defined such that standard event body field names correspond to values of well-known data types. Examples of such mappings for common event types used within the Telecommunications industry are provided in Section 1.2, "Conformance Issues," on page 1-3 of this document. In addition, end users can define their own name-value pairs which comprise the filterable portion of any proprietary event types.

remainder_of_body

The **remainder_of_body** portion of the event body is intended to hold event data upon which filtering is not likely to be performed. From a logical point of view, the "interesting" fields of the event data should be placed into the **filterable_data** portion, and the "rest" of the event placed here. Obviously it is not possible to predict what portion of the event will be interesting (or not) to all consumers. The division of the event body within the structured event in this fashion merely provides a hint to consumers. It is still possible to perform filtering on the contents of the **remainder_of_body** portion of the event body, however this will require decomposing the Any data structure which contains this portion into actual typed data elements, using the typecode contained within the Any. Thus filtering on this portion of the event body is likely to be less efficient than filtering on the **filterable_data** portion.

3.1.2 The EventBatch Data Type

The **CosNotification** module defines the **EventBatch** data type as a sequence of Structured Events. The **CosNotifyComm** module defined in Section 3.3, "The CosNotifyComm Module," on page 3-28 defines interfaces which support the transmission and receipt of sequences of Structured Events within a single operation. Such a sequence of Structured Events transmitted as a unit is referred to as an *event batch*, and is of the **EventBatch** data type.

3.1.3 QoS and Administrative Constant Declarations

The **CosNotification** module declares several constants related to QoS properties of event transmission, and administrative properties of notification channels. The meanings of each property related to QoS and its associated values is described in detail in Section 2.5.5, "Notification QoS Properties," on page 2-37 of this document. The meanings of each property related to channel administration and its associated values is described in Section 2.5.7, "Notification Channel Administrative Properties," on page 2-48.

3.1.4 The QoSAdmin Interface

The **QoSAdmin** interface defines operations which enable clients to get and set the values of QoS properties. It also defines an operation that can verify whether or not a set of requested QoS property settings can be satisfied, along with returning information about the range of possible settings for additional QoS properties. **QoSAdmin** is intended to be an abstract interface which is inherited by the Proxy, Admin, and Event Channel interfaces defined in the **CosNotifyChannelAdmin** and **CosTypedNotifyChannelAmin** modules. The semantics of the operations supported by this interface are defined below.

3.1.4.1 get_qos

The **get_qos** operation takes no input parameters, and returns a sequence of name-value pairs which encapsulates the current quality of service settings for the target object (which could be an Event Channel, Admin, or Proxy object).

3.1.4.2 set_qos

The **set_qos** operation takes as an input parameter a sequence of name-value pairs which encapsulates quality of service property settings that a client is requesting that the target object (which could be an Event Channel, Admin, or Proxy object) support as its default quality of service. If the implementation of the target object is not capable of supporting any of the requested quality of service settings, or if any of the requested settings would be in conflict with a QoS property defined at a higher level of the object hierarchy with respect to QoS (see Section 2.5.6, "Negotiating QoS and Conflict Resolution," on page 2-42), the UnsupportedQoS exception is raised. This exception contains as data a sequence of data structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes which might be returned are described in Table 2-5 on page 2-46.

3.1.4.3 validate_qos

The **validate_qos** operation accepts as input a sequence of QoS property name-value pairs which specify a set of QoS settings that a client would like to know if the target object is capable of supporting. If the any of the requested settings could not be

satisfied by the target object, the operation raises the UnsupportedQoS exception. This exception contains as data a sequence of data structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes which might be returned are described in Table 2-5 on page 2-46.

If all requested QoS property value settings could be satisfied by the target object, the operation returns successfully (without actually setting the QoS properties on the target object) with an output parameter that contains a sequence of **PropertyRange** data structures. Each element in this sequence includes the name of an additional QoS property supported by the target object which could have been included on the input list and resulted in a successful return from the operation., along with the range of values that would have been acceptable for each such property.

3.1.5 The AdminPropertiesAdmin Interface

The **AdminPropertiesAdmin** interface defines operations which enable clients to get and set the values of administrative properties. This interface is intended to be an abstract interface which is inherited by the Event Channel interfaces defined in the **CosNotifyChannelAdmin** and **CosTypedNotifyChannelAmin** modules. The semantics of the operations supported by this interface are defined below.

3.1.5.1 get_admin

The **get_admin** operation takes no input parameters, and returns a sequence of name-value pairs which encapsulates the current administrative settings for the target channel.

3.1.5.2 set_admin

The **set_admin** operation takes as an input parameter a sequence of name-value pairs which encapsulates administrative property settings that a client is requesting that the target channel support. If the implementation of the target object is not capable of supporting any of the requested administrative property settings, the UnsupportedAdmin exception is raised. This exception has associated with it a list of name-value pairs of which each name identifies an administrative property whose requested setting could not be satisfied, and each associated value the closest setting for that property which could be satisfied.

3.2 The CosNotifyFilter Module

The **CosNotifyFilter** module defines the interfaces supported by the filter objects used by the Notification Service. Two different types of filter objects are defined here. The first type support the **Filter** interface and encapsulate the constraints which will be used by a proxy object associated with a notification channel in order to make decisions about which events to forward, and which to discard. The second type support the **MappingFilter** interface and encapsulate constraints along with associated

values, whereby the constraints determine whether a proxy object will alter the way it treats each event with respect to a particular property of the event, and the value specifies the property value the proxy would apply to each event which satisfies the associated constraint. In addition to the two types of filter object interface defined in this module, the **CosNotifyFilter** module also defines the **FilterFactory** interface which supports the operations required to create each type of filter object, and the **FilterAdmin** interface which supports operations which enable an interface (Proxy or Admin) which inherits it to manage a list of **Filter** instances.

```
module CosNotifyFilter {

    typedef long ConstraintID;

    struct ConstraintExp {
        CosNotification::EventTypeSeq event_types;
        string constraint_expr;
    };

    typedef sequence<ConstraintID> ConstraintIDSeq;
    typedef sequence<ConstraintExp> ConstraintExpSeq;

    struct ConstraintInfo {
        ConstraintExp constraint_expression;
        ConstraintID constraint_id;
    };

    typedef sequence<ConstraintInfo> ConstraintInfoSeq;

    struct MappingConstraintPair {
        ConstraintExp constraint_expression;
        any result_to_set;
    };

    typedef sequence<MappingConstraintPair> MappingConstraintPairSeq;

    struct MappingConstraintInfo {
        ConstraintExp constraint_expression;
        ConstraintID constraint_id;
        any value;
    };

    typedef sequence<MappingConstraintInfo> MappingConstraintInfoSeq;

    typedef long CallbackID;
    typedef sequence<CallbackID> CallbackIDSeq;
```

```
exception UnsupportedFilterableData {};
exception InvalidGrammar {};
exception InvalidConstraint {ConstraintExp constr;};
exception DuplicateConstraintID {ConstraintID id;};

exception ConstraintNotFound {ConstraintID id;};
exception CallbackNotFound {};

exception InvalidValue {ConstraintExp constr; any value;};

interface Filter {

    readonly attribute string constraint_grammar;

    ConstraintInfoSeq add_constraints (
            in ConstraintExpSeq constraint_list)
        raises (InvalidConstraint);

    void modify_constraints (
        in ConstraintIDSeq del_list,
        in ConstraintInfoSeq modify_list)
    raises (InvalidConstraint, ConstraintNotFound);

ConstraintInfoSeq get_constraints(
        in ConstraintIDSeq id_list)
    raises (ConstraintNotFound);

ConstraintInfoSeq get_all_constraints();

void remove_all_constraints();

void destroy();

boolean match ( in any filterable_data )
    raises (UnsupportedFilterableData);

boolean match_structured (
        in CosNotification::StructuredEvent filterable_data )
    raises (UnsupportedFilterableData);
```

```
        boolean match_typed (
                in CosNotification::PropertySeq filterable_data )
            raises (UnsupportedFilterableData);

        CallbackID attach_callback (
                in CosNotifyComm::NotifySubscribe callback);

        void detach_callback ( in CallbackID callback)
            raises ( CallbackNotFound );

        CallbackIDSeq get_callbacks();

    }; // Filter

    interface MappingFilter {

        readonly attribute string constraint_grammar;

        readonly attribute CORBA::TypeCode value_type;

        readonly attribute any default_value;

        MappingConstraintInfoSeq add_mapping_constraints (
                in MappingConstraintPairSeq pair_list)
            raises (InvalidConstraint, InvalidValue);

        void modify_mapping_constraints (
                in ConstraintIDSeq del_list,
                in MappingConstraintInfoSeq modify_list)
            raises (InvalidConstraint, InvalidValue,
                        ConstraintNotFound);

        MappingConstraintInfoSeq get_mapping_constraints (
                in ConstraintIDSeq id_list)
            raises (ConstraintNotFound);

        MappingConstraintInfoSeq get_all_mapping_constraints();

        void remove_all_mapping_constraints();

        void destroy();
```

```
      boolean match ( in any filterable_data,
                out any result_to_set )
          raises (UnsupportedFilterableData);

      boolean match_structured (
                in CosNotification::StructuredEvent filterable_data,
                out any result_to_set)
          raises (UnsupportedFilterableData);

      boolean match_typed (
                in CosNotification::PropertySeq filterable_data,
                out any result_to_set)
          raises (UnsupportedFilterableData);

}; // MappingFilter

interface FilterFactory {

      Filter create_filter (
                in string constraint_grammar)
          raises (InvalidGrammar);

      MappingFilter create_mapping_filter (
                in string constraint_grammar,
                in any default_value)
          raises(InvalidGrammar);

}; // FilterFactory

typedef long FilterID;
typedef sequence<FilterID> FilterIDSeq;

exception FilterNotFound {};

interface FilterAdmin {

      FilterID add_filter ( in Filter new_filter );

      void remove_filter ( in FilterID filter )
          raises ( FilterNotFound );
```

```
            Filter get_filter ( in FilterID filter )
                raises ( FilterNotFound );

            FilterIDSeq get_all_filters();

            void remove_all_filters();

        }; // FilterAdmin

    }; // CosNotifyFilter
```

3.2.1 The Filter Interface

The **Filter** interface defines the behaviors supported by objects which encapsulate constraints used by the proxy objects associated with an event channel in order to determine which events they receive will be forwarded, and which will be discarded. Each object supporting the **Filter** interface can encapsulate a sequence of any number of constraints. Each event received by a proxy object which has one or more objects supporting the **Filter** interface associated with it must satisfy at least one of the constraints associated with one of its associated **Filter** objects in order to be forwarded (either to another proxy object or to the consumer, depending on the type of proxy the filter is associated with), otherwise it will be discarded.

Each constraint encapsulated by a filter object is a structure comprised of two main components. The first component is a sequence of data structures, each of which indicates an event type comprised of a domain and a type name. The second component is a boolean expression over the properties of an event, expressed in some constraint grammar (more on this below). For a given constraint, the sequence of event type structures in the first component nominates a set of event types to which the constraint expression in the second component applies. Each element of the sequence can contain strings which will be matched for equality against the **domain_name** and **type_name** fields of each event being evaluated by the filter object, or it could contain strings with wildcard symbols (*), indicating a pattern match should be performed against the type contained in each event, rather than a comparison for equality when determining if the boolean expression should be applied to the event, or the event should simply be discarded without even attempting to apply the boolean expression. Note that an empty sequence included as the first component of a constraint implies that the associated expression applies to all types of events, as does a sequence comprised of a single element whose domain and type name are both set to either the empty string or else the wildcard symbol alone contained in quotes.

The constraint expressions associated with a particular object supporting the **Filter** interface are expressed as strings which obey the syntax of a particular constraint grammar (i.e., a BNF). Every conformant implementation of this service must support constraint expressions expressed in the default constraint grammar described in Section 2.4, “The Default Filter Constraint Language,” on page 2-23. In addition,

implementations may support other constraint grammars, and/or users of this service may implement their own filter objects which allow constraints to be expressed in terms of an alternative constraint grammar. As long as such user-defined filter objects support the **Filter** interface, they can be attached to Proxy or Admin objects in the same fashion as the default **Filter** objects supported by the implementation of the service are, and the channel should be able to use them to filter events in the same fashion.

The **Filter** interface supports the operations required to manage the constraints associated with an object instance which supports the interface, along with a readonly attribute which identifies the particular constraint grammar in which the constraints encapsulated by this object have meaning. In addition, the **Filter** interface supports three variants of the **match** operation which can be invoked by an associated proxy object upon receipt of an event (the specific variant selected depends upon whether the event is received in the form of an Any, a Structured Event, or a Typed Event), to determine if the event should be forwarded or discarded, based on whether or not the event satisfies at least one criteria encapsulated by the filter object.

The **Filter** interface also supports operations which enable a client to associate with the target filter object any number of “callbacks” which are notified each time there is a change to the list of event types which the constraints encapsulated by the filter object could potentially cause proxies to which the filter is attached to receive. Operations are also defined to support administration of this callback list by unique identifier.

The operations supported by the **Filter** interface are described in more detail within the following subsections.

3.2.1.1 constraint_grammar

The **constraint_grammar** attribute is a readonly attribute which identifies the particular grammar within which the constraint expressions encapsulated by the target filter object have meaning. The value of this attribute is set upon creation of a filter object instance, based on the input provided to the factory creation operation for the filter instance.

The dependency of a filter object on its constraints being expressed within a particular constraint grammar manifests itself within the implementation of the **match** operations described below, which must be able to parse the constraints to determine whether or not a particular event satisfies one of them.

Every conformant implementation of the Notification Service must support an implementation of the **Filter** interface which supports the default constraint grammar described in Section 2.4, “The Default Filter Constraint Language,” on page 2-23. **[Reviewer please clarify the following:** The value which the constraint_grammar attribute is set to in the case the target filter object supports this default grammar will be “EXTENDED_TCL”.**]** In addition, implementations and/or end users may provide additional implementations of the **Filter** interface which support different constraint grammars, and thus would set the **constraint_grammar** attribute to a different value upon creation of such a filter object.

3.2.1.2 add_contraints

The **add_constraints** operation is invoked by a client in order to associate one or more new constraints with the target filter object. The operation accepts as input a sequence of constraint data structures, each element of which consists of a sequence of event type structures (described in Section 3.2.1, "The Filter Interface," on page 3-14) and a constraint expressed within the constraint grammar supported by the target object. Upon processing each constraint, the target object associates a numeric identifier with the constraint that is unique among all constraints it encapsulates. If any of the constraints in the input sequence is not a valid expression within the supported constraint grammar, the InvalidConstraint exception is raised. This exception contains as data the specific constraint expression that was determined to be invalid. Upon successful processing of all input constraint expressions, the **add_constraints** operation returns a sequence in which each element will be a structure including one of the input constraint expressions, along with the unique identifier assigned to it by the target filter object.

Note that the semantics of the **add_constraints** operation are such that its side-effects are performed atomically upon the target filter object. Once **add_constraints** is invoked by a client, the target filter object is temporarily disabled from usage by any proxy object it may be associated with. The operation is then carried out, either successfully adding all of the input constraints to the target object or none of them (in the case one of the input expressions was invalid). Upon completion of the operation, the target filter object is effectively re-enabled and can once again be used by associated filter objects in order to make event forwarding decisions.

3.2.1.3 modify_constraints

The **modify_constraints** operation is invoked by a client in order to modify the constraints associated with the target filter object. This operation can be used both to remove constraints currently associated with the target filter object, and to modify the constraint expressions of constraints which have previously been added to the target filter object.

The operation accepts two input parameters. The first input parameter is a sequence of numeric values which are each intended to be the unique identifier associated with one of the constraints currently encapsulated by the target filter object. If all input values supplied within a particular invocation of this operation are valid, then the specific constraints identified by the values contained in the first input parameter will be deleted from the list of those encapsulated by the target filter object.

The second input parameter to this operation is a sequence of structures, each of which contains a constraint structure and a numeric value. The numeric value contained by each element of the sequence is intended to be the unique identifier associated with one of the constraints currently encapsulated by the target filter object. If all input values supplied within a particular invocation of this operation are valid, then the constraint expression associated with the already encapsulated constraint identified by the numeric value contained within each element of the input sequence will be modified to the new constraint expression that is contained within the same sequence element.

If any of the numeric values supplied within either of the two input sequences does not correspond to the unique identifier associated with some constraint currently encapsulated by the target filter object, the ConstraintNotFound exception is raised. This exception contains as data the specific identifier which was supplied as input but did not correspond to the identifier of some constraint encapsulated by the target object. If any of the constraint expressions supplied within an element of the second input sequence is not a valid expression in terms of the constraint grammar supported by the target object, the InvalidConstraint exception is raised. This exception contains as data the specific constraint that was determined to be invalid.

Note that the semantics of the **modify_constraints** operation are such that its side-effects are performed atomically upon the target filter object. Once **modify_constraints** is invoked by a client, the target filter object is temporarily disabled from usage by any proxy object it may be associated with. The operation is then carried out, either successfully deleting all of the constraints identified in the first input sequence and modifying those associated with constraints identified in the second input sequence, or performing no side effects to the target object (in the case one of the inputs was invalid). Upon completion of the operation, the target filter object is effectively re-enabled and can once again be used by associated filter objects in order to make event forwarding decisions.

3.2.1.4 get_constraints

The **get_constraints** operation is invoked to return a sequence of a subset of the constraints associated with the target filter object. The operation accepts as input a sequence of numeric values which should correspond to the unique identifiers of constraints encapsulated by the target object. If one of the input values does not correspond to the identifier of some encapsulated constraint, the ConstraintNotFound exception is raised, containing as data the numeric value that did not correspond to some constraint. Upon successful completion, this operation returns a sequence of data structures, each of which contains one of the input identifiers along with its associated constraint.

3.2.1.5 get_all_constraints

The **get_all_constraints** operation returns all of the constraints currently encapsulated by the target filter object. The return value of this operation is a sequence of structures, each of which contains one of the constraints encapsulated by the target object along with its associated unique identifier.

3.2.1.6 remove_all_constraints

The **remove_all_constraints** operation is invoked to remove all of the constraints currently encapsulated by the target filter object. Upon completion, the target filter object will still exist but have no constraints associated with it.

3.2.1.7 *destroy*

The **destroy** operation destroys the target filter object, invalidating its object reference.

3.2.1.8 *match*

The **match** operation evaluates the filter constraints associated with the target filter object against an instance of an event supplied to the channel in the form of a **CORBA::Any**. The operation accepts as input a **CORBA::Any** which contains an event to be evaluated, and returns a boolean value which will be TRUE in cases where the input event satisfies one of the filter constraints, and FALSE otherwise. The act of determining whether or not a given event passes a given filter constraint is specific to the type of grammar in which the filter constraint is specified. Thus, this operation will need to be re-implemented for each supported grammar.

If the input parameter contains data that the **match** operation is not designed to handle, the UnsupportedFilterableData exception will be raised. An example of this would be if the filterable data contained a field whose name corresponds to a standard event field that has a numeric value, but the actual value associated with this field name within the event is a string.

3.2.1.9 *match_structured*

The **match_structured** operation evaluates the filter constraints associated with the target filter object against an instance of an event supplied to the channel in the form of a Structured Event. The operation accepts as input a data structure of type **CosNotification::StructuredEvent** which contains an event to be evaluated, and returns a boolean value which will be TRUE in cases where the input event satisfies one of the filter constraints, and FALSE otherwise. The act of determining whether or not a given event passes a given filter constraint is specific to the type of grammar in which the filter constraint is specified. Thus, this operation will need to be re-implemented for each supported grammar.

If the input parameter contains data that the *match* operation is not designed to handle, the UnsupportedFilterableData exception will be raised. An example of this would be if the filterable data contained a field whose name corresponds to a standard event field that has a numeric value, but the actual value associated with this field name within the event is a string.

3.2.1.10 *match_typed*

The **match** operation evaluates the filter constraints associated with the target filter object against an instance of an event supplied to the channel in the form of a typed event. The operation accepts as input a sequence of name-value pairs which contains the contents of the event to be evaluated (how a typed event is converted to a sequence of name-value pairs by the channel is described in Section 2.7, “Filtering Typed Events,” on page 2-52), and returns a boolean value which will be TRUE in cases where the input event satisfies one of the filter constraints, and FALSE otherwise. The

act of determining whether or not a given event passes a given filter constraint is specific to the type of grammar in which the filter constraint is specified. Thus, this operation will need to be re-implemented for each supported grammar.

If the input parameter contains data that the **match** operation is not designed to handle, the UnsupportedFilterableData exception will be raised. An example of this would be if the filterable data contained a field whose name corresponds to a standard event field that has a numeric value, but the actual value associated with this field name within the event is a string.

3.2.1.11 attach_callback

The **attach_callback** operation accepts as input the reference to an object supporting the **CosNotifyComm::NotifySubscribe** interface, and returns a numeric value assigned to this callback that is unique to all such callbacks currently associated with the target object. This operation is invoked to associate with the target filter object an object supporting the **CosNotifyComm::NotifySubscribe** interface. This interface is inherited by all supplier interfaces (either those that are clients of a notification channel, or those that are proxy objects within a notification channel) defined by the Notification Service, and supports a **subscription_change** operation. After this operation has been successfully invoked on a filter object, each time the set of constraints associated with the target filter object is modified (either by an invocation of its **add_constraints** or its **modify_constraints** operations), the filter object will invoke the **subscription_change** object of all its associated callback objects in order to inform suppliers to which the target filter object is attached of the change in the set of event types to which clients of the filter object subscribe. This enables suppliers to make intelligent decisions about which types of events it should actually produce, and which it can suppress the production of. This mechanism is described in more detail in Section 2.6, “Sharing Subscriptions,” on page 2-49.

3.2.1.12 detach_callback

The **detach_callback** operation accepts as input a numeric value which should be one of the unique identifiers associated with one of the callback objects attached to the target filter object. If the input value does not correspond to the unique identifier of a callback object currently attached to the target filter object, the CallbackNotFound exception is raised. Otherwise, the callback object to which the input value corresponds is removed from the list of those associated with the target filter object, so that subsequent changes to the event type subscription list encapsulated by the target filter object will not be propagated to the callback object which is being detached.

3.2.1.13 get_callbacks

The **get_callbacks** operation accepts no input parameters and returns the sequence of all unique identifiers associated with callback objects attached to the target filter object.

3.2.2 The MappingFilter Interface

The **MappingFilter** interface defines the behaviors of objects which encapsulate a sequence of constraint-value pairs, where each constraint is a structure of the same type as that described in Section 3.2.1, "The Filter Interface," on page 3-14, and each value represents a possible setting of a particular property of an event. Note that *setting of a particular property* is not intended to imply that any contents of the event will be altered as a result of applying a mapping filter, but rather the way a proxy treats the event with respect to a particular property (i.e., priority or lifetime) could change. Upon receiving each event, a proxy object with an associated object supporting the **MappingFilter** interface will invoke the appropriate **match** operation variant (depending upon whether the event is received in the form of an untyped event, a Structured Event, or a typed event) on the mapping filter object in order to determine how it should modify a particular property value associated with the event to one of the values associated with one of the constraints encapsulated by the mapping filter. Internally, the mapping filter object applies the constraints it encapsulates to the event in order to determine whether or not the event's property should be modified to one of the values associated with a constraint, or else the default value associated with the mapping filter.

Each instance of an object supporting the **MappingFilter** interface is typically associated with a specific event property. For instance, in this specification **MappingFilter** object instances are used to affect the properties of priority and lifetime for events received by a proxy supplier object. Each event received by a proxy object, which has an object supporting the **MappingFilter** interface associated with it must satisfy at least one of the constraints associated with the **MappingFilter** object in order to have its property value modified, otherwise the property will remain unchanged. A specific instance supporting the **MappingFilter** interface typically applies its encapsulated constraints in an order which begins with the best possible property setting (e.g., the highest priority or the longest lifetime), and ends with the worst possible property setting. As soon as a matching constraint is encountered, the associated value is returned as an output parameter and the proxy which invoked the operation proceeds to modify the property of the event to the new value.

The constraint expressions associated with a particular object supporting the **MappingFilter** interface are expressed as strings which obey the syntax of a particular constraint grammar (i.e., a BNF). Every conformant implementation of this service must support constraint expressions expressed in the default constraint grammar described in Section 2.4, "The Default Filter Constraint Language," on page 2-23. In addition, implementations may support other constraint grammars, and/or users of this service may implement their own filter objects which allow constraints to be expressed in terms of an alternative constraint grammar. As long as such user-defined filter objects support the **MappingFilter** interface, they can be attached to proxy objects in the same fashion as the default **MappingFilter** objects supported by the implementation of the service are, and the channel should be able to use them to potentially affect the properties of events in the same fashion.

The **MappingFilter** interface supports the operations required to manage the constraint-value pairs associated with an object instance which supports the interface. In addition, the **MappingFilter** interface supports a readonly attribute which identifies

the particular constraint grammar in which the constraints encapsulated by this object have meaning. The **MappingFilter** interface also supports a readonly attribute which identifies the typecode associated with the datatype of the specific property value it is intended to affect, and another readonly attribute which holds the default value which will be returned as the result of a match operation in cases when the event in question is found to satisfy none of the constraints encapsulated by the mapping filter. Lastly, the **MappingFilter** interface supports three variants of the operation which will be invoked by an associated proxy object upon receipt of an event, to determine how the property of the event which the target mapping filter object was designed to affect should be modified.

The operations supported by the **MappingFilter** object are described in more detail within the following subsections.

3.2.2.1 *constraint_grammar*

The **constraint_grammar** attribute is a readonly attribute which identifies the particular grammar within which the constraint expressions encapsulated by the target filter object have meaning. The value of this attribute is set upon creation of a mapping filter object instance, based on the input provided to the factory creation operation for the mapping filter instance.

The dependency of a filter object on its constraints being expressed within a particular constraint grammar manifests itself within the implementation of the **match** operations described below, which must be able to parse the constraints to determine whether or not a particular event satisfies one of them.

Every conformant implementation of the Notification Service must support an implementation of the **MappingFilter** object which supports the default constraint grammar described in Section 2.4, "The Default Filter Constraint Language," on page 2-23. The value which the **constraint_grammar** attribute is set to in case the target filter object supports this default grammar will be "EXTENDED_TCL." In addition, implementations and/or end users may provide additional implementations of the **MappingFilter** interface that support different constraint grammars, and thus would set the **constraint_grammar** attribute to a different value upon creation of such a filter object.

3.2.2.2 *value_type*

The **value_type** attribute is a readonly attribute which identifies the datatype of the property value which the target mapping filter object was designed to affect. Note that the factory creation operation for mapping filter objects accepts as an input parameter the **default_value** to associate with the mapping filter instance. This **default_value** is a **CORBA::Any**. Upon creation of a mapping filter, the Typecode associated with the **default_value** is extracted from the **CORBA::Any**, and its value is assigned to this attribute. The **value_type** attribute thus serves mainly as a convenience for clients attempting to examine the state of a mapping filter object.

3.2.2.3 default_value

The **default_value** attribute is a readonly attribute that will be the output parameter returned as the result of any **match** operation during which the input event is found to satisfy none of the constraints encapsulated by the mapping filter. within which the constraints encapsulated by the target filter object have meaning. The value of this attribute is set upon creation of a mapping filter object instance, based on the input provided to the factory creation operation for the mapping filter instance.

3.2.2.4 add_mapping_contraints

The **add_mapping_constraints** operation is invoked by a client in order to associate specific mapping constraints with the target filter object. Note that a mapping constraint is comprised of a constraint structure paired with an associated value. The operation accepts as input one parameter that is a sequence of constraint-value pairs. Each constraint in this sequence must be expressed within the constraint grammar supported by the target object, and each associated value must be of the data type indicated by the **value_type** attribute of the target object.

Upon processing each element in the input sequence, the target object associates a numeric identifier with this constraint-value pair that is unique among all those that it encapsulates. If any of the constraint expressions in the input sequence is not a valid expression within the supported constraint grammar, the InvalidConstraint exception is raised. This exception contains as data the specific constraint that was determined to be invalid. If any of the values supplied in the input sequence is not of the same datatype as that indicated by the **value_type** attribute associated with the target object, the InvalidValue exception is raised. This exception contains as data both the invalid value and its corresponding constraint in the first input sequence. Upon successful processing of all input constraints, the **add_mapping_constraints** operation returns a sequence in which each element will be a structure including one of the input constraint expressions, its corresponding value, and the unique identifier assigned to this constraint-value pair by the target filter object.

Note that the semantics of the **add_mapping_constraints** operation are such that its side-effects are performed atomically upon the target filter object. Once **add_mapping_constraints** is invoked by a client, the target filter object is temporarily disabled from usage by any proxy object it may be associated with. The operation is then carried out, either successfully adding all of the input constraint-value pairs to the target object or none of them (in case one of the input expressions or values was invalid). Upon completion of the operation, the target filter object is effectively re-enabled and can once again be used by associated filter objects in order to make event property mapping decisions.

3.2.2.5 modify_mapping_constraints

The **modify_mapping_constraints** operation is invoked by a client in order to modify the constraint-value pairs associated with the target filter object. This operation can be used both to remove constraint-value pairs currently associated with the target filter object, and to modify the constraints and/or values of constraint-value pairs which have previously been added to the target filter object.

The operation accepts two input paramaters. The first input parameter is a sequence of numeric values which are each intended to be the unique identifier associated with one of the constraint-value pairs currently encapsulated by the target filter object. If all input values supplied within a particular invocation of this operation are valid, then the specific constraint-value pairs identified by the values contained in the first input parameter will be deleted from the list of those encapsulated by the target filter object.

The second input parameter to this operation is a sequence of structures, each of which contains a constraint structure, an Any value, and a numeric identifier. The numeric identifier contained by each element of the sequence is intended to be the unique identifier associated with one of the constraint-value pairs currently encapsulated by the target filter object. If all input values supplied within a particular invocation of this operation are valid, then the constraint associated with the already encapsulated constraint-value pair identified by the numeric identifier contained within each element of the input sequence will be modified to the new constraint that is contained within the same sequence element. Likewise, the data value associated with the already encapsulated constraint-value pair identified by the numeric identifier contained within each element of the input sequence will be modified to the new data value that is contained in the same element of the sequence.

If any of the numeric identifiers supplied within either of the two input sequences does not correspond to the unique identifier associated with some constraint-value pairs currently encapsulated by the target filter object, the ConstraintNotFound exception is raised. This exception contains as data the specific identifier which was supplied as input but did not correspond to the identifier of some constraint-value pair encapsulated by the target object. If any of the constraint expressions supplied within an element of the second input sequence is not a valid expression in terms of the constraint grammar supported by the target object, the InvalidConstraint exception is raised. This exception contains as data the specific constraint that was determined to be invalid. If any of the values supplied in the second input sequence is not of the same datatype as that indicated by the **value_type** attribute associated with the target object, the InvalidValue exception is raised. This exception contains as data both the invalid value and its corresponding constraint expression.

Note that the semantics of the **modify_mapping_constraints** operation are such that its side-effects are performed atomically upon the target filter object. Once **modify_mapping_constraints** is invoked by a client, the target filter object is temporarily disabled from usage by any proxy object it may be associated with. The operation is then carried out, either successfully deleting all of the constraint-value pairs identified in the first input sequence and modifying the constraints and values associated with constraints identified in the second input sequence, or performing no side effects to the target object (in the case one of the inputs was invalid). Upon

completion of the operation, the target filter object is effectively re-enabled and can once again be used by associated filter objects in order to make event property mapping decisions.

3.2.2.6 *get_mapping_constraints*

The **get_mapping_constraints** operation is invoked to return a sequence of a subset of the constraint-value pairs associated with the target filter object. The operation accepts as input a sequence of numeric values which should correspond to the unique identifiers of constraint-value pairs encapsulated by the target object. If one of the input values does not correspond to the identifier of some encapsulated constraint-value pair, the ConstraintNotFound exception is raised, containing as data the numeric value that did not correspond to some such pair. Upon successful completion, this operation returns a sequence of data structures, each of which contains one of the input identifiers along with its associated constraint structure and constraint value.

3.2.2.7 *get_all_mapping_constraints*

The **get_all_mapping_constraints** operation returns all of the constraint-value pairs currently encapsulated by the target filter object. The return value of this operation is a sequence of structures, each of which contains one of the constraints encapsulated by the target object along with its associated value and unique identifier.

3.2.2.8 *remove_all_mapping_constraints*

The **remove_all_mapping_constraints** operation is invoked to remove all of the constraint-value pairs currently encapsulated by the target filter object. Upon completion, the target filter object will still exist but have no constraint-value pairs associated with it.

3.2.2.9 *destroy*

The **destroy** operation destroys the target filter object, invalidating its object reference.

3.2.2.10 *match*

The **match** operation is invoked on an object supporting the **MappingFilter** interface in order to determine how some property value of a particular event supplied to the channel in the form of a **CORBA::Any** should be modified. The operation accepts an Any as input, which contains the event being evaluated. Upon invocation, the target mapping filter object begins applying the constraints it encapsulates in order according to each constraints associated value, starting with the constraint with the “best” associated value for the specific property the mapping filter object was designed to affect (e.g., the highest priority, the longest lifetime, etc.), and ending with the constraint with the “worst” associated value. Upon encountering a constraint which the input filterable data matches, the operation sets the output parameter contained in its

signature to the value associated with the constraint, and sets the return value of the operation to TRUE. If the input filterable data satisfies none of the constraints encapsulated by the target mapping filter object, the return value of the operation is set to FALSE, and the output parameter is set to the value of the **default_value** attribute associated with the target mapping filter object. The act of determining whether or not a given filterable event data passes a given filter constraint is specific to the type of grammar in which the filter constraint is specified. Thus, this operation will need to be re-implemented for each supported grammar.

If the input parameter contains data that the **match** operation is not designed to handle, the UnsupportedFilterableData exception will be raised. An example of this would be if the filterable data contained a field whose name corresponds to a standard event field that has a numeric value, but the actual value associated with this field name within the event is a string.

3.2.2.11 match_structured

The **match_structured** operation is invoked on an object supporting the **MappingFilter** interface in order to determine how some property value of a particular event supplied to the channel in the form of a Structured Event should be modified. The operation accepts a **CosNotification::StructuredEvent** as input, which contains the event being evaluated. Upon invocation, the target mapping filter object begins applying the constraints it encapsulates in order according to each constraints associated value, starting with the constraint with the “best” associated value for the specific property the mapping filter object was designed to affect (e.g., the highest priority, the longest lifetime, etc.), and ending with the constraint with the “worst” associated value. Upon encountering a constraint which the input filterable data matches, the operation sets the output parameter contained in its signature to the value associated with the constraint, and sets the return value of the operation to TRUE. If the input filterable data satisfies none of the constraints encapsulated by the target mapping filter object, the return value of the operation is set to FALSE, and the output parameter is set to the value of the **default_value** attribute associated with the target mapping filter object. The act of determining whether or not a given filterable event data passes a given filter constraint is specific to the type of grammar in which the filter constraint is specified. Thus, this operation will need to be re-implemented for each supported grammar.

If the input parameter contains data that the **match** operation is not designed to handle, the UnsupportedFilterableData exception will be raised. An example of this would be if the filterable data contained a field whose name corresponds to a standard event field that has a numeric value, but the actual value associated with this field name within the event is a string.

3.2.2.12 match_typed

The **match_typed** operation is invoked on an object supporting the **MappingFilter** interface in order to determine how some property value of a particular event supplied to the channel in the form of a typed event should be modified. The operation accepts as input a sequence of name-value pairs which contains the contents of the event to be

evaluated (how a typed event is converted to a sequence of name-value pairs by the channel is described in Section 2.7, "Filtering Typed Events," on page 2-52). Upon invocation, the target mapping filter object begins applying the constraints it encapsulates in order according to each constraints associated value, starting with the constraint with the "best" associated value for the specific property the mapping filter object was designed to affect (e.g., the highest priority, the longest lifetime, etc.), and ending with the constraint with the "worst" associated value. Upon encountering a constraint which the input filterable data matches, the operation sets the output parameter contained in its signature to the value associated with the constraint, and sets the return value of the operation to TRUE. If the input filterable data satisfies none of the constraints encapsulated by the target mapping filter object, the return value of the operation is set to FALSE, and the output parameter is set to the value of the **default_value** attribute associated with the target mapping filter object. The act of determining whether or not a given filterable event data passes a given filter constraint is specific to the type of grammar in which the filter constraint is specified. Thus, this operation will need to be re-implemented for each supported grammar.

If the input parameter contains data that the **match** operation is not designed to handle, the UnsupportedFilterableData exception will be raised. An example of this would be if the filterable data contained a field whose name corresponds to a standard event field that has a numeric value, but the actual value associated with this field name within the event is a string.

3.2.3 The FilterFactory Interface

The **FilterFactory** interface defines operations for creating filter objects.

3.2.3.1 create_filter

The **create_filter** operation is responsible for creating a new forwarding filter object. It takes as input a string parameter which identifies the grammar in which constraints associated with this filter will have meaning. If the client invoking this operation supplies as input the name of a grammar that is not supported by any forwarding filter implementation this factory is capable of creating, the InvalidGrammar exception is raised. Otherwise, the operation returns the reference to an object supporting the **Filter** interface, which can subsequently be configured to support constraints in the appropriate grammar.

3.2.3.2 create_mapping_filter

The **create_mapping_filter** operation is responsible for creating a new mapping filter object. It takes as input a string parameter which identifies the grammar in which constraints associated with this filter will have meaning, and an Any which will be set as the **default_value** of the newly created mapping filter. If the client invoking this operation supplies as input the name of a grammar that is not supported by any mapping filter implementation this factory is capable of creating, the InvalidGrammar exception is raised. Otherwise, the operation returns the reference

to an object supporting the **MappingFilter** interface, which can subsequently be configured to support constraints in the appropriate grammar, along with their associated mapping values.

3.2.4 The FilterAdmin Interface

The **FilterAdmin** interface defines operations that enable an object supporting this interface to manage a list of filter objects, each of which supports the **Filter** interface. This interface is intended to be an abstract interface which is inherited by all of the **Proxy** and **Admin** interfaces defined by the Notification Service. The difference in the semantics between a list of filter objects that is associated with an **Admin** object, and a list that is associated with a **Proxy** object, is described in Section 2.1.2, "The Notification Service Event Channel," on page 2-5.

3.2.4.1 add_filter

The **add_filter** operation accepts as input the reference to an object supporting the **Filter** interface. The affect of this operation is that the input filter object is appended to the list of filter objects associated with the target object upon which the operation was invoked. The operation associates with the newly added filter object a numeric identifier that is unique among all filter objects currently associated with the target, and returns that value as the result of the operation.

3.2.4.2 remove_filter

The **remove_filter** operation accepts as input a numeric value that is intended to be the unique identifier of a filter object that is currently associated with the target object. If identifier supplied does correspond to a filter object currently associated with the target object, then the corresponding filter object will be removed from the list of filters associated with the target object. Otherwise, the FilterNotFound exception will be raised.

3.2.4.3 get_filter

The **get_filter** operation accepts as input a numeric identifier that is intended to correspond to one of the filter objects currently associated with the target object. If this is the case, the object reference of the corresponding filter object is returned. Otherwise, the FilterNotFound exception is raised.

3.2.4.4 get_all_filters

The **get_all_filters** operation accepts no input parameters, and returns the list of unique identifiers which correspond to all of the filters currently associated with the target object.

3.2.4.5 remove_all_filters

The **remove_all_filters** operation accepts no input parameters, and removes all filter objects from the list of those currently associated with the target object.

3.3 The CosNotifyComm Module

The **CosNotifyComm** module defines the interfaces which support Notification Service clients that communicate using Anys, Structured Events, or sequences of Structured Events. In addition, this module defines the interfaces which enable event suppliers to be informed when the types of events being subscribed to by their associated consumers change, and event consumers to be informed whenever there is a change in the types of events being produced by their suppliers (this model is described in detail in Section 2.6, “Sharing Subscriptions,” on page 2-49).

```
module CosNotifyComm {

    exception InvalidEventType { CosNotification::EventType type; };


    interface NotifyPublish {


        void offer_change (
                in CosNotification::EventTypeSeq added,
                in CosNotification::EventTypeSeq removed )
            raises ( InvalidEventType );


    }; // NotifyPublish


    interface NotifySubscribe {


        void subscription_change(

            in CosNotification::EventTypeSeq added,
            in CosNotification::EventTypeSeq removed )
        raises ( InvalidEventType );


    }; // NotifySubscribe


    interface PushConsumer :
        NotifyPublish,
        CosEventComm::PushConsumer {
    }; // PushConsumer


    interface PullConsumer :
        NotifyPublish,
```

```
        CosEventComm::PullConsumer {
}; // PullConsumer

interface PullSupplier :
        NotifySubscribe,
        CosEventComm::PullSupplier {
}; // PullSupplier

interface PushSupplier :
        NotifySubscribe,
        CosEventComm::PushSupplier {
};

interface StructuredPushConsumer : NotifyPublish {

        void push_structured_event(
                        in CosNotification::StructuredEvent notification)
                raises(CosEventComm::Disconnected);

        void disconnect_structured_push_consumer();

}; // StructuredPushConsumer

interface StructuredPullConsumer : NotifyPublish {
        void disconnect_structured_pull_consumer();
}; // StructuredPullConsumer

interface StructuredPullSupplier : NotifySubscribe {

        CosNotification::StructuredEvent pull_structured_event()
                raises(CosEventComm::Disconnected);
        CosNotification::StructuredEvent try_pull_structured_event(
                        out boolean has_event)
                raises(CosEventComm::Disconnected);

        void disconnect_structured_pull_supplier();

}; // StructuredPullSupplier

interface StructuredPushSupplier : NotifySubscribe {
        void disconnect_structured_push_supplier();
}; // StructuredPushSupplier
```

```
interface SequencePushConsumer : NotifyPublish {

    void push_structured_events(
            in CosNotification::EventBatch notifications)
        raises(CosEventComm::Disconnected);

    void disconnect_sequence_push_consumer();

}; // SequencePushConsumer

interface SequencePullConsumer : NotifyPublish {
    void disconnect_sequence_pull_consumer();
}; // SequencePullConsumer

interface SequencePullSupplier : NotifySubscribe {

    CosNotification::EventBatch pull_structured_events(
            in long max_number )
        raises(CosEventComm::Disconnected);

    CosNotification::EventBatch try_pull_structured_events(
            in long max_number,
            out boolean has_event)
        raises(CosEventComm::Disconnected);

    void disconnect_sequence_pull_supplier();

}; // SequencePullSupplier

interface SequencePushSupplier : NotifySubscribe {
    void disconnect_sequence_push_supplier();
}; // SequencePushSupplier

}; // CosNotifyComm
```

3.3.1 The NotifyPublish Interface

The **NotifyPublish** interface supports an operation which allows a supplier of Notifications to announce, or publish, the names of the types of events it will be supplying, It is intended to be an abstract interface which is inherited by all Notification Service consumer interfaces, and enables suppliers to inform consumers supporting this interface of the types of events they intend to supply.

3.3.1.1 offer_change

The **offer_change** operation takes as input two sequences of event type names: the first specifying those event types which the client of the operation (an event supplier) is informing the target consumer object that it is adding to the list of event types it plans to supply, and the second specifying those event types which the client no longer plans to supply. This operation raises the InvalidEventType exception if one of the event type names supplied in either input parameter is syntactically invalid. In this case, the invalid name is returned in the **type** field of the exception.

Note that each event type name is comprised of two components: the name of the domain in which the event type has meaning, and the name of the actual event type. Also note that either component of a type name may specify a complete domain/event type name, a domain/event type name containing the wildcard '*' character, or the special event type name "%ALL" described in Section 2.6.6, "Special Event Types," on page 2-51.

3.3.2 The NotifySubscribe Interface

The **NotifySubscribe** interface supports an operation which allows a consumer of notifications to inform suppliers of notifications of the types of notifications it wishes to receive. It is intended to be an abstract interface which is inherited by all Notification Service supplier interfaces. In essence, its main purpose is to enable notification consumers to inform suppliers of the types of notifications that are of interest to them, ultimately enabling the suppliers to avoid supplying notifications that are not of interest to any consumer.

3.3.2.1 subscription_change

The **subscription_change** operation takes as input two sequences of event type names: the first specifying those event types which the associated Consumer wants to add to its subscription list, and the second specifying those event types which the associated consumer wants to remove from its subscription list. This operation raises the InvalidEventType exception if one of the event type names supplied in either input parameter is syntactically invalid. If this case, the invalid name is returned in the *type* field of the exception.

Note that each event type name is comprised of two components: the name of the domain in which the event type has meaning, and the name of the actual event type. Also note that either component of a type name may specify a complete domain/event type name, a domain/event type name containing the wildcard '*' character, or the special event type name "%ALL" described in Section 2.6.6, "Special Event Types," on page 2-51.

3.3.3 The PushConsumer Interface

The **PushConsumer** interface supports the functionality required by notification service consumers that receive events as Anys using push-style communication. This interface defines no new attributes or operations directly. Instead, it multiply inherits

the **PushConsumer** interface defined in the **CosEventComm** module of the OMG Event Service, and the **NotifyPublish** interface described above. This enables push-style consumers of Any events to also receive **offer_change** messages from the channel, allowing it to learn about changes to the types of events being offered to the channel by suppliers.

3.3.4 The PullConsumer Interface

The **PullConsumer** interface supports the functionality required by notification service consumers that receive events as Anys using pull-style communication. This interface defines no new attributes or operations directly. Instead, it multiply inherits the **PullConsumer** interface defined in the **CosEventComm** module of the OMG Event Service, and the **NotifyPublish** interface described above. This enables pull-style consumers of Any events to also receive **offer_change** messages from the channel, allowing it to learn about changes to the types of events being offered to the channel by suppliers.

3.3.5 The PullSupplier Interface

The **PullSupplier** interface supports the functionality required by notification service suppliers that transmit events as Anys using pull-style communication. This interface defines no new attributes or operations directly. Instead, it multiply inherits the **PullSupplier** interface defined in the **CosEventComm** module of the OMG Event Service, and the **NotifySubscribe** interface described above. This enables pull-style suppliers of Any events to also receive **subscription_change** messages from the channel, allowing it to learn about changes to the types of events being subscribed to by consumers connected to the channel.

3.3.6 The PushSupplier Interface

The **PushSupplier** interface supports the functionality required by notification service suppliers that transmit events as Anys using push-style communication. This interface defines no new attributes or operations directly. Instead, it multiply inherits the **PushSupplier** interface defined in the **CosEventComm** module of the OMG Event Service, and the **NotifySubscribe** interface described above. This enables push-style suppliers of Any events to also receive **subscription_change** messages from the channel, allowing it to learn about changes to the types of events being subscribed to by consumers connected to the channel.

3.3.7 The StructuredPushConsumer Interface

The **StructuredPushConsumer** interface supports an operation which enables consumers to receive Structured Events by the push model. It also defines an operation which can be invoked to disconnect the push consumer from its associated supplier. In addition, the **StructuredPushConsumer** interface inherits the **NotifyPublish**

interface described above, enabling a notification supplier to inform an instance supporting this interface whenever there is a change to the types of events it intends to produce.

Note that an object supporting the **StructuredPushConsumer** interface can receive all events which were supplied to its associated channel, including events supplied in a form other than a Structured Event. How events supplied to the channel in other forms are internally mapped into a Structured Event for delivery to a **StructuredPushConsumer** is summarized in Table 2-2 on page 2-9.

3.3.7.1 push_structured_event

The **push_structured_event** operation takes as input a parameter of type **StructuredEvent** as defined in the **CosNotification** module. Upon invocation, this parameter will contain an instance of a Structured Event being delivered to the consumer by the supplier to which it is connected. If this operation is invoked upon a **StructuredPushConsumer** instance that is not currently connected to the supplier of the event, the Disconnected exception will be raised. Note that the condition that a proxy supplier believes it is actively connected to a consumer, while the consumer believes it is disconnected is an invalid state. Thus, if the invocation of **push_structured_event** upon a **StructuredPushConsumer** instance by a **StructuredProxyPushSupplier** instance results in the Disconnected exception being raised, the **StructuredProxyPushSupplier** will invoke its own **disconnect_structured_push_supplier** operation, resulting in the destruction of that **StructuredProxyPushSupplier** instance.

In reality there are two types of objects that will support the **StructuredPushConsumer** interface: an object representing an application which receives Structured Events, and a **StructuredProxyPushConsumer** (defined in the **CosNotifyChannelAdmin** module) associated with an event channel which receives structured events from a supplier on behalf of the channel. For the first type of object, the implementation of the **push_structured_event** operation is application specific, and is intended to be supplied by application developers. For the second type of object, the behavior of the operation is tightly linked to the implementation of the event channel. Basically, it is responsible for applying any filters that have been registered by with the **StructuredProxyPushConsumer**, then either discarding the event or forwarding it to each proxy supplier within the channel depending on whether or not the event passed the filter.

3.3.7.2 disconnect_structured_push_consumer

The **disconnect_structured_push_consumer** operation is invoked to terminate a connection between the target **StructuredPushConsumer**, and its associated supplier. This operation takes no input parameters and returns no values. The result of this operation is that the target **StructuredPushConsumer** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.8 The StructuredPullConsumer Interface

The **StructuredPullConsumer** interface supports the behavior of objects that receive Structured Events using pull-style communication. It defines an operation which can be invoked to disconnect the pull consumer from its associated supplier. In addition, the **StructuredPullConsumer** interface inherits the **NotifyPublish** interface described above, enabling a notification supplier to inform an instance supporting this interface whenever there is a change to the types of events it intends to produce.

Note that an object supporting the **StructuredPullConsumer** interface can receive all events which were supplied to its associated channel, including events supplied in a form other than a Structured Event. How events supplied to the channel in other forms are internally mapped into a Structured Event for delivery to a **StructuredPullConsumer** is summarized in Table 2-2 on page 2-9.

3.3.8.1 disconnect_structured_pull_consumer

The **disconnect_structured_pull_consumer** operation is invoked to terminate a connection between the target **StructuredPullConsumer**, and its associated supplier. This operation takes no input parameters and returns no values. The result of this operation is that the target **StructuredPullConsumer** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.9 The StructuredPullSupplier Interface

The **StructuredPullSupplier** interface supports operations which enable suppliers to transmit Structured Events by the pull model. It also defines an operation which can be invoked to disconnect the pull supplier from its associated consumer. In addition, the **StructuredPullSupplier** interface inherits the **NotifySubscribe** interface described above, enabling a notification consumer to inform an instance supporting this interface whenever there is a change to the types of events it is interested in receiving.

Note that an object supporting the **StructuredPullSupplier** interface can transmit events which can potentially be received by any consumer connected to the channel, including those which consume events in a form other than a Structured Event. How events supplied to the channel in the form of a Structured Event are internally mapped into different forms for delivery to consumers which receive events in a form other than the Structured Event is summarized in Table 2-2 on page 2-9.

3.3.9.1 pull_structured_event

The **pull_structured_event** operation takes no input parameters, and returns a value of type **Structured Event** as defined in the **CosNotification** module. Upon invocation, the operation will block until an event is available for transmission, at which time it will return an instance of a Structured Event, which contains the event being delivered to its connected consumer. If invoked upon a **StructuredPullSupplier** that is not currently connected to the consumer of the event, the Disconnected exception will be raised. Note that the condition that a proxy consumer believes it is actively connected to a supplier, while the supplier believes it is

disconnected is an invalid state. Thus, if the invocation of **pull_structured_event** upon a **StructuredPullSupplier** instance by a **StructuredProxyPullConsumer** instance results in the Disconnected exception being raised, the **StructuredProxyPullConsumer** will invoke its own **disconnect_structured_pull_consumer** operation, resulting in the destruction of that **StructuredProxyPullConsumer** instance.

In reality there are two types of objects that will support the **StructuredPullSupplier** interface: an object representing an application which transmits Structured Events, and a **StructuredProxyPullSupplier** (defined in the **CosNotifyChannelAdmin** module) associated with an event channel which transmits events to a pull style consumer on behalf of the channel. For the first type of object, the implementation of the **pull_structured_event** operation is application specific, and is intended to be supplied by application developers. The application specific implementation of this operation should construct a structured event, and return it within a **StructuredEvent** data structure. For the second type of object, the behavior of the operation is tightly linked to the implementation of the event channel. Basically, it is responsible for forwarding a structured event, within a **StructuredEvent** data structure, as the return value to the consumer it is connected to upon the availability of an event which passes the filter(s) associated with the **StructuredProxyPullSupplier**. Note that the operation will block until such an event is available to return.

3.3.9.2 try_pull_structured_event

The **try_pull_structured_event** operation takes no input parameters, and returns a value of type **StructuredEvent** as defined in the **CosNotification** module. It also returns an output parameter of type boolean which indicates whether or not the return value actually contains an event. Upon invocation, the operation will return an instance of a Structured Event which contains the event being delivered to its connected consumer, if such an event is available for delivery at the time the operation was invoked. If an event is available for delivery and thus returned as the result, the output parameter of the operation will be set to TRUE. If no event is available to return upon invocation, the operation will return immediately with the value of the output parameter set to FALSE. In this case, the return value will not contain a valid event. If invoked upon a **StructuredPullSupplier** that is not currently connected to the consumer of the event, the Disconnected exception will be raised. Note that the condition that a proxy consumer believes it is actively connected to a supplier, while the supplier believes it is disconnected is an invalid state. Thus, if the invocation of **try_pull_structured_event** upon a **StructuredPullSupplier** instance by a **StructuredProxyPullConsumer** instance results in the Disconnected exception being raised, the **StructuredProxyPullConsumer** will invoke its own **disconnect_structured_pull_consumer** operation, resulting in the destruction of that **StructuredProxyPullConsumer** instance.

In reality there are two types of objects that will support the **StructuredPullSupplier** interface: an object representing an application which transmits Structured Events, and a **StructuredProxyPullSupplier** (defined within the **CosNotifyChannelAdmin** module) associated with an event channel which transmits events to a PullConsumer on behalf of the channel. For the first type of object, the implementation of the

try_pull_structured_event operation is application specific, and is intended to be supplied by application developers. If an event is available to be returned upon invocation of this operation, the application specific implementation of this operation should construct a Structured Event, and return it within a **StructuredEvent** data structure along with setting the value of the output parameter to TRUE. Otherwise, the operation should return immediately after setting the value of the output parameter to FALSE. For the second type of object, the behavior of the operation is tightly linked to the implementation of the event channel. Basically, if an event is available to be returned upon invocation of this operation, it is responsible for forwarding it, within a **StructuredEvent** data structure, as the return value to the consumer it is connected to, in addition to setting the output parameter to FALSE. If no event is available to return to the consumer upon invocation of this operation, it will immediately return with the output parameter to set to FALSE, and the return value not containing a valid event.

3.3.9.3 disconnect_structured_pull_supplier

The **disconnect_structured_pull_supplier** operation is invoked to terminate a connection between the target **StructuredPullSupplier**, and its associated consumer. This operation takes no input parameters and returns no values. The result of this operation is that the target **StructuredPullSupplier** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.10 The StructuredPushSupplier Interface

The **StructuredPushSupplier** interface supports the behavior of objects that transmit Structured Events using push-style communication. It defines an operation which can be invoked to disconnect the push supplier from its associated consumer. In addition, the **StructuredPushSupplier** interface inherits the **NotifySubscribe** interface described above, enabling a notification consumer to inform an instance supporting this interface whenever there is a change to the types of events it is interested in receiving.

Note that an object supporting the **StructuredPushSupplier** interface can transmit events which can potentially be received by any consumer connected to the channel, including those which consume events in a form other than a Structured Event. How events supplied to the channel in the form of a Structured Event are internally mapped into different forms for delivery to consumers which receive events in a form other than the Structured Event is summarized in Table 2-2 on page 2-9.

3.3.10.1 disconnect_structured_push_supplier

The **disconnect_structured_push_supplier** operation is invoked to terminate a connection between the target **StructuredPushSupplier**, and its associated consumer. This operation takes no input parameters and returns no values. The result of this operation is that the target **StructuredPushSupplier** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.11 The SequencePushConsumer Interface

The **SequencePushConsumer** interface supports an operation which enables consumers to receive sequences Structured Events by the push model. It also defines an operation which can be invoked to disconnect the push consumer from its associated supplier. In addition, the **SequencePushConsumer** interface inherits the **NotifyPublish** interface described above, enabling a notification supplier to inform an instance supporting this interface whenever there is a change to the types of events it intends to produce.

Note that an object supporting the **SequencePushConsumer** interface can receive all events which were supplied to its associated channel, including events supplied in a form other than a sequence of Structured Events. How events supplied to the channel in other forms are internally mapped into a sequence of Structured Events for delivery to a **SequencePushConsumer** is summarized in Table 2-2 on page 2-9.

3.3.11.1 push_structured_events

The **push_structured_events** operation takes as input a parameter of type **EventBatch** as defined in the **CosNotification** module. This data type is the same as a sequence of Structured Events. Upon invocation, this parameter will contain a sequence of Structured Events being delivered to the consumer by the supplier to which it is connected. If this operation is invoked upon a **SequencePushConsumer** instance that is not currently connected to the supplier of the event, the Disconnected exception will be raised. Note that the condition that a proxy supplier believes it is actively connected to a consumer, while the consumer believes it is disconnected is an invalid state. Thus, if the invocation of **push_structured_events** upon a **SequencePushConsumer** instance by a **SequenceProxyPushSupplier** instance results in the Disconnected exception being raised, the **SequenceProxyPushSupplier** will invoke its own **disconnect_sequence_push_supplier** operation, resulting in the destruction of that **SequenceProxyPushSupplier** instance.

Note that the maximum number of events that will be transmitted within a single invocation of this operation, along with the amount of time a supplier of a sequence of Structured Events will accumulate individual events into the sequence before invoking this operation, are controlled by QoS property settings as described in Section 2.5.5, “Notification QoS Properties,” on page 2-37.

In reality there are two types of objects that will support the **SequencePushConsumer** interface: an object representing an application which receives sequences of Structured Events, and a **SequenceProxyPushConsumer** (defined in the **CosNotifyChannelAdmin** module) associated with an event channel, which receives sequences of Structured Events from a supplier on behalf of the channel. For the first type of object, the implementation of the **push_structured_events** operation is application-specific, and is intended to be supplied by application developers. For the second type of object, the behavior of the operation is tightly linked to the implementation of the event channel. Basically, it is responsible for applying any filters that have been registered by with the

SequenceProxyPushConsumer to each event in each sequence it receives, then either discarding each event or forwarding it to each proxy supplier within the channel depending on whether or not the event passed the filter.

3.3.11.2 disconnect_sequence_push_consumer

The **disconnect_sequence_push_consumer** operation is invoked to terminate a connection between the target **SequencePushConsumer**, and its associated supplier. This operation takes no input parameters and returns no values. The result of this operation is that the target **SequencePushConsumer** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.12 The SequencePullConsumer Interface

The **SequencePullConsumer** interface supports the behavior of objects that receive sequences of Structured Events using pull-style communication. It defines an operation which can be invoked to disconnect the pull consumer from its associated supplier. In addition, the **SequencePullConsumer** interface inherits the **NotifyPublish** interface described above, enabling a notification supplier to inform an instance supporting this interface whenever there is a change to the types of events it intends to produce.

Note that an object supporting the **SequencePullConsumer** interface can receive all events which were supplied to its associated channel, including events supplied in a form other than a sequence of Structured Events. How events supplied to the channel in other forms are internally mapped into a sequence of Structured Events for delivery to a **SequencePullConsumer** is summarized in Table 2-2 on page 2-9.

3.3.12.1 disconnect_sequence_pull_consumer

The **disconnect_sequence_pull_consumer** operation is invoked to terminate a connection between the target **SequencePullConsumer**, and its associated supplier. This operation takes no input parameters and returns no values. The result of this operation is that the target **SequencePullConsumer** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.13 The SequencePullSupplier Interface

The **SequencePullSupplier** interface supports operations that enable suppliers to transmit sequences of Structured Events by the pull model. It also defines an operation that can be invoked to disconnect the pull supplier from its associated consumer. In addition, the **SequencePullSupplier** interface inherits the **NotifySubscribe** interface described above, enabling a notification consumer to inform an instance supporting this interface whenever there is a change to the types of events it is interested in receiving.

Note that an object supporting the **SequencePullSupplier** interface can transmit events that can potentially be received by any consumer connected to the channel, including those that consume events in a form other than a sequence of Structured Events. How events supplied to the channel in the form of a sequence of Structured Events are internally mapped into different forms for delivery to consumers which receive events in a form other than a sequence of Structured Events is summarized in Table 2-2 on page 2-9.

3.3.13.1 pull_structured_events

The **pull_structured_events** operation takes as an input parameter a numeric value, and returns a value of type **EventBatch** as defined in the **CosNotification** module. This data type is the same as a sequence of Structured Events. Upon invocation, the operation will block until a sequence of Structured Events is available for transmission, at which time it will return the sequence containing events being delivered to its connected consumer. If invoked upon a **SequencePullSupplier** that is not currently connected to the consumer of the event, the Disconnected exception will be raised. Note that the condition that a proxy consumer believes it is actively connected to a supplier, while the supplier believes it is disconnected is an invalid state. Thus, if the invocation of **pull_structured_events** upon a **SequencePullSupplier** instance by a **SequenceProxyPullConsumer** instance results in the Disconnected exception being raised, the **SequenceProxyPullConsumer** will invoke its own **disconnect_sequence_pull_consumer** operation, resulting in the destruction of that **SequenceProxyPullConsumer** instance.

Note that the maximum length of the sequence returned will never exceed the value of the input parameter. In addition, when this operation is invoked upon a **SequenceProxyPullSupplier**, the amount of time the supplier will accumulate events into the sequence before transmitting it is controlled by the PacingInterval QoS property described in Section 2.5.5, "Notification QoS Properties," on page 2-37. In this case, the proxy will never return a sequence of less than **MaximumBatchSize** events until at least **PacingInterval** amount of time has elapsed after the request was received by the proxy. However if no events arrived at the proxy during a particular **PacingInterval**, the request will block until at least one event arrives at the proxy. Also note that **MaximumBatchSize** places an upper boundary on the total number of events the proxy will return within an invocation. If the input parameter indicates more than **MaximumBatchSize** events are being requested, the request will be treated as though the input parameter was equivalent to **MaximumBatchSize**.

In reality there are two types of objects that will support the **SequencePullSupplier** interface: an object representing an application which transmits sequences of Structured Events, and a **SequenceProxyPullSupplier** (defined in the **CosNotifyChannelAdmin** module) associated with an event channel which transmits events to a pull style consumer on behalf of the channel. For the first type of object, the implementation of the **pull_structured_events** operation is application specific, and is intended to be supplied by application developers. The application specific implementation of this operation should construct a sequence of Structured Events, and return it within a **EventBatch** data structure. For the second type of object, the behavior of the operation is tightly linked to the implementation of the event channel.

Basically, it is responsible for forwarding a sequence of Structured Events, within an **EventBatch** data structure, as the return value to the consumer it is connected to upon the availability of events which pass the filter(s) associated with the **SequenceProxyPullSupplier**.

3.3.13.2 *try_pull_structured_events*

The **try_pull_structured_events** operation takes as an input parameter a numeric value, and returns a value of type **EventBatch** as defined in the **CosNotification** module. This data type is the same as a sequence of Structured Events. The operation also returns an output parameter of type boolean which indicates whether or not the return value actually contains a sequence of events. Upon invocation, the operation will return a sequence of a Structured Events which contains events being delivered to its connected consumer, if such a sequence is available for delivery at the time the operation was invoked. If an event sequence is available for delivery and thus returned as the result, the output parameter of the operation will be set to TRUE. If no event sequence is available to return upon invocation, the operation will return immediately with the value of the output parameter set to FALSE. In this case, the return value will not contain a valid event sequence. If invoked upon a **SequencePullSupplier** that is not currently connected to the consumer of the event, the Disconnected exception will be raised. Note that the condition that a proxy consumer believes it is actively connected to a supplier, while the supplier believes it is disconnected is an invalid state. Thus, if the invocation of **try_pull_structured_events** upon a **SequencePullSupplier** instance by a **SequenceProxyPullConsumer** instance results in the Disconnected exception being raised, the **SequenceProxyPullConsumer** will invoke its own **disconnect_sequence_pull_consumer** operation, resulting in the destruction of that **SequenceProxyPullConsumer** instance.

Note that the maximum length of the sequence returned will never exceed the value of the input parameter. Also note that **MaximumBatchSize** places an upper boundary on the total number of events the proxy will return within an invocation. If the input parameter indicates more than **MaximumBatchSize** events are being requested, the request will be treated as though the input parameter was equivalent to **MaximumBatchSize**.

In reality there are two types of objects that will support the **SequencePullSupplier** interface: an object representing an application which transmits sequences of Structured Events, and a **SequenceProxyPullSupplier** (defined within the **CosNotifyChannelAdmin** module) associated with an event channel which transmits events to a **PullConsumer** on behalf of the channel. For the first type of object, the implementation of the **try_pull_structured_events** operation is application-specific, and is intended to be supplied by application developers. If an event sequence is available to be returned upon invocation of this operation, the application-specific implementation of this operation should construct an **EventBatch** instance, and return it along with setting the value of the output parameter to TRUE. Otherwise, the operation should return immediately after setting the value of the output parameter to FALSE. For the second type of object, the behavior of the operation is tightly linked to the implementation of the event channel. Basically, if an event sequence is available to

be returned upon invocation of this operation, it is responsible for forwarding it, within an **EventBatch** data structure, as the return value to the consumer it is connected to, in addition to setting the output parameter to FALSE. If no event sequence is available to return to the consumer upon invocation of this operation, it will immediately return with the output parameter to set to FALSE, and the return value not containing a valid event.

3.3.13.3 disconnect_sequence_pull_supplier

The **disconnect_sequence_pull_supplier** operation is invoked to terminate a connection between the target **SequencePullSupplier**, and its associated consumer. This operation takes no input parameters and returns no values. The result of this operation is that the target **SequencePullSupplier** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.3.14 The SequencePushSupplier Interface

The **SequencePushSupplier** interface supports the behavior of objects that transmit sequences of Structured Events using push-style communication. It defines an operation that can be invoked to disconnect the push supplier from its associated consumer. In addition, the **SequencePushSupplier** interface inherits the **NotifySubscribe** interface described above, enabling a notification consumer to inform an instance supporting this interface whenever there is a change to the types of events it is interested in receiving.

Note that an object supporting the **SequencePushSupplier** interface can transmit events that can potentially be received by any consumer connected to the channel, including those that consume events in a form other than a sequence of Structured Events. How events supplied to the channel in the form of a sequence of Structured Events are internally mapped into different forms for delivery to consumers that receive events in a form other than a sequence of Structured Events is summarized in Table 2-2 on page 2-9.

3.3.14.1 disconnect_sequence_push_supplier

The **disconnect_sequence_push_supplier** operation is invoked to terminate a connection between the target **SequencePushSupplier**, and its associated consumer. This operation takes no input parameters and returns no values. The result of this operation is that the target **SequencePushSupplier** will release all resources it had allocated to support the connection, and dispose its own object reference.

3.4 The CosNotifyChannelAdmin Module

The **CosNotifyChannelAdmin** module defines the interfaces necessary to create, configure, and administer instances of a Notification Service event channel. It defines the different types of proxy interfaces that support connections from the various types of clients that are supported, the **Admin** interfaces, the **EventChannel** interface, and a factory interface for instantiating new channels.

```
module CosNotifyChannelAdmin {

    exception ConnectionAlreadyActive {};
    exception ConnectionAlreadyInactive {};
    exception NotConnected {};


    // Forward declarations
    interface ConsumerAdmin;
    interface SupplierAdmin;
    interface EventChannel;
    interface EventChannelFactory;


    enum ProxyType {
        PUSH_ANY,
        PULL_ANY,
        PUSH_STRUCTURED,
        PULL_STRUCTURED,
        PUSH_SEQUENCE,
        PULL_SEQUENCE,
        PUSH_TYPED,
        PULL_TYPED
    };


    enum ObtainInfoMode {
        ALL_NOW_UPDATES_OFF,
        ALL_NOW_UPDATES_ON,
        NONE_NOW_UPDATES_OFF,
        NONE_NOW_UPDATES_ON
    };


    interface ProxyConsumer :
            CosNotification::QoSAdmin,
            CosNotifyFilter::FilterAdmin {


        readonly attribute ProxyType MyType;
        readonly attribute SupplierAdmin MyAdmin;


        CosNotification::EventTypeSeq obtain_subscription_types(
                in ObtainInfoMode mode );


        void validate_event_qos (
                in CosNotification::QoSProperties required_qos,
                out CosNotification::NamedPropertyRangeSeq available_qos)
            raises (CosNotification::UnsupportedQoS);


    }; // ProxyConsumer
```

```
interface ProxySupplier :
        CosNotification::QoSAdmin,
        CosNotifyFilter::FilterAdmin {

    readonly attribute ProxyType MyType;
    readonly attribute ConsumerAdmin MyAdmin;

    attribute CosNotifyFilter::MappingFilter priority_filter;
    attribute CosNotifyFilter::MappingFilter lifetime_filter;

    CosNotification::EventTypeSeq obtain_offered_types(
            in ObtainInfoMode mode );

    void validate_event_qos (
             in CosNotification::QoSProperties required_qos,
             out CosNotification::NamedPropertyRangeSeq available_qos)
        raises (CosNotification::UnsupportedQoS);

}; // ProxySupplier

interface ProxyPushConsumer :
        ProxyConsumer,
        CosNotifyComm::PushConsumer {

void connect_any_push_supplier (
        in CosEventComm::PushSupplier push_supplier)
    raises(CosEventChannelAdmin::AlreadyConnected);

}; // ProxyPushConsumer

interface StructuredProxyPushConsumer :
        ProxyConsumer,
        CosNotifyComm::StructuredPushConsumer {

    void connect_structured_push_supplier (
            in CosNotifyComm::StructuredPushSupplier push_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected);

}; // StructuredProxyPushConsumer

interface SequenceProxyPushConsumer :
        ProxyConsumer,
```

```
		CosNotifyComm::SequencePushConsumer {

	void connect_sequence_push_supplier (
			in CosNotifyComm::SequencePushSupplier push_supplier)
		raises(CosEventChannelAdmin::AlreadyConnected);

}; // SequenceProxyPushConsumer

interface ProxyPullSupplier :
		ProxySupplier,
		CosNotifyComm::PullSupplier {

	void connect_any_pull_consumer (
			in CosEventComm::PullConsumer pull_consumer)
		raises(CosEventChannelAdmin::AlreadyConnected);

}; // ProxyPullSupplier

interface StructuredProxyPullSupplier :
		ProxySupplier,
		CosNotifyComm::StructuredPullSupplier {

void connect_structured_pull_consumer (
			in CosNotifyComm::StructuredPullConsumer pull_consumer)
		raises(CosEventChannelAdmin::AlreadyConnected);

}; // StructuredProxyPullSupplier

interface SequenceProxyPullSupplier :
		ProxySupplier,
		CosNotifyComm::SequencePullSupplier {

	void connect_sequence_pull_consumer (
			in CosNotifyComm::SequencePullConsumer pull_consumer)
		raises(CosEventChannelAdmin::AlreadyConnected);

}; // SequenceProxyPullSupplier

interface ProxyPullConsumer :
		ProxyConsumer,
		CosNotifyComm::PullConsumer {
```

```
    void connect_any_pull_supplier (
            in CosEventComm::PullSupplier pull_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive, NotConnected);

    void resume_connection()
        raises(ConnectionAlreadyActive, NotConnected);

}; // ProxyPullConsumer

interface StructuredProxyPullConsumer :
            ProxyConsumer,
            CosNotifyComm::StructuredPullConsumer {

    void connect_structured_pull_supplier (
            in CosNotifyComm::StructuredPullSupplier pull_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive, NotConnected);

    void resume_connection()
        raises(ConnectionAlreadyActive, NotConnected);

}; // StructuredProxyPullConsumer

interface SequenceProxyPullConsumer :
            ProxyConsumer,
            CosNotifyComm::SequencePullConsumer {

    void connect_sequence_pull_supplier (
            in CosNotifyComm::SequencePullSupplier pull_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive, NotConnected);

    void resume_connection()
```

```
        raises(ConnectionAlreadyActive, NotConnected);

}; // SequenceProxyPullConsumer

interface ProxyPushSupplier :
        ProxySupplier,
        CosNotifyComm::PushSupplier {

    void connect_any_push_consumer (
            in CosEventComm::PushConsumer push_consumer)
        raises(CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive, NotConnected);

    void resume_connection()
        raises(ConnectionAlreadyActive, NotConnected);

}; // ProxyPushSupplier

interface StructuredProxyPushSupplier :
            ProxySupplier,
            CosNotifyComm::StructuredPushSupplier {

    void connect_structured_push_consumer (
            in CosNotifyComm::StructuredPushConsumer push_consumer)
        raises(CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive, NotConnected);

    void resume_connection()
        raises(ConnectionAlreadyActive, NotConnected);

}; // StructuredProxyPushSupplier

interface SequenceProxyPushSupplier :
        ProxySupplier,
        CosNotifyComm::SequencePushSupplier {
```

```
        void connect_sequence_push_consumer (
                in CosNotifyComm::SequencePushConsumer push_consumer)
            raises(CosEventChannelAdmin::AlreadyConnected,
                    CosEventChannelAdmin::TypeError );

        void suspend_connection()
            raises(ConnectionAlreadyInactive, NotConnected);

        void resume_connection()
            raises(ConnectionAlreadyActive, NotConnected);

    }; // SequenceProxyPushSupplier

    typedef long ProxyID;
    typedef sequence <ProxyID> ProxyIDSeq;

    enum ClientType {
        ANY_EVENT,
        STRUCTURED_EVENT,
        SEQUENCE_EVENT
    };

    enum InterFilterGroupOperator { AND_OP, OR_OP };

    typedef long AdminID;
    typedef sequence<AdminID> AdminIDSeq;

    exception AdminNotFound {};
    exception ProxyNotFound {};

    struct AdminLimit {
        CosNotification::PropertyName name;
        CosNotification::PropertyValue value;
    };

    exception AdminLimitExceeded { AdminLimit admin_property_err; };

    interface ConsumerAdmin :
            CosNotification::QoSAdmin,
            CosNotifyComm::NotifySubscribe,
            CosNotifyFilter::FilterAdmin,
            CosEventChannelAdmin::ConsumerAdmin {
```

```
        readonly attribute AdminID MyID;
        readonly attribute EventChannel MyChannel;

        readonly attribute InterFilterGroupOperator MyOperator;

        attribute CosNotifyFilter::MappingFilter priority_filter;
        attribute CosNotifyFilter::MappingFilter lifetime_filter;

        readonly attribute ProxyIDSeq pull_suppliers;
        readonly attribute ProxyIDSeq push_suppliers;

        ProxySupplier get_proxy_supplier (
                in ProxyID proxy_id )
            raises ( ProxyNotFound );

        ProxySupplier obtain_notification_pull_supplier (
                in ClientType ctype,
                out ProxyID proxy_id)
            raises ( AdminLimitExceeded );

        ProxySupplier obtain_notification_push_supplier (
                in ClientType ctype,
                out ProxyID proxy_id)
            raises ( AdminLimitExceeded );

        void destroy();

    }; // ConsumerAdmin

    interface SupplierAdmin :
            CosNotification::QoSAdmin,
            CosNotifyComm::NotifyPublish,
            CosNotifyFilter::FilterAdmin,
            CosEventChannelAdmin::SupplierAdmin {

        readonly attribute AdminID MyID;
        readonly attribute EventChannel MyChannel;

        readonly attribute InterFilterGroupOperator MyOperator;

        readonly attribute ProxyIDSeq pull_consumers;
        readonly attribute ProxyIDSeq push_consumers;
```

```
        ProxyConsumer get_proxy_consumer (
                in ProxyID proxy_id )
            raises ( ProxyNotFound );

        ProxyConsumer obtain_notification_pull_consumer (
                in ClientType ctype,
                out ProxyID proxy_id)
            raises ( AdminLimitExceeded );

        ProxyConsumer obtain_notification_push_consumer (
                in ClientType ctype,
                out ProxyID proxy_id)
            raises ( AdminLimitExceeded );

        void destroy();

    }; // SupplierAdmin

    interface EventChannel :
        CosNotification::QoSAdmin,
        CosNotification::AdminPropertiesAdmin,
        CosEventChannelAdmin::EventChannel {

        readonly attribute EventChannelFactory MyFactory;

        readonly attribute ConsumerAdmin default_consumer_admin;
        readonly attribute SupplierAdmin default_supplier_admin;

        readonly attribute CosNotifyFilter::FilterFactory
                                default_filter_factory;

        ConsumerAdmin new_for_consumers(
                in InterFilterGroupOperator op,
                out AdminID id );

        SupplierAdmin new_for_suppliers(
                in InterFilterGroupOperator op,
                out AdminID id );

        ConsumerAdmin get_consumeradmin ( in AdminID id )
            raises (AdminNotFound);
```

```
        SupplierAdmin get_supplieradmin ( in AdminID id )
            raises (AdminNotFound);

        AdminIDSeq get_all_consumeradmins();
        AdminIDSeq get_all_supplieradmins();

    }; // EventChannel

    typedef long ChannelID;
    typedef sequence<ChannelID> ChannelIDSeq;

    exception ChannelNotFound {};

    interface EventChannelFactory {

        EventChannel create_channel (
                in CosNotification::QoSProperties initial_qos,
                in CosNotification::AdminProperties initial_admin,
                out ChannelID id)
            raises(CosNotification::UnsupportedQoS,
                    CosNotification::UnsupportedAdmin );

        ChannelIDSeq get_all_channels();

        EventChannel get_event_channel ( in ChannelID id )
            raises (ChannelNotFound);

}; // EventChannelFactory

    }; // CosNotifyChannelAdmin
```

3.4.1 The ProxyConsumer Interface

The **ProxyConsumer** interface is intended to be an abstract interface that is inherited by the different varieties of proxy consumers that can be instantiated within a notification channel. It encapsulates the behaviors common to all Notification Service proxy consumers. In particular, the **ProxyConsumer** interface inherits the **QoSAdmin** interface defined within the **CosNotification** module, and the **FilterAdmin** interface defined within the **CosNotifyFilter** module. The former inheritance enables all proxy consumers to administer a list of associated QoS properties, while the latter inheritance enables all proxy consumers to administer a list of associated filter objects. Locally, the **ProxyConsumer** interface defines a readonly attribute that should be set upon creation of each proxy consumer instance to indicate

the specific type of proxy consumer the instance represents, and a readonly attribute which maintains a reference to the **SupplierAdmin** object that created it. In addition, the **ProxyConsumer** interface defines an operation that returns the list of event types a given proxy consumer instance is configured to forward, and an operation which can be queried to determine which message level QoS properties can be set on a per-event basis.

3.4.1.1 MyType

The **MyType** attribute is a readonly attribute that should be set upon creation of each proxy consumer instance to indicate the specific type of proxy consumer the instance represents. **[Reviewer please clarify the following:**Enumerations are possible to distinguish the type of proxy consumer among the following possibilities: **ProxyPushConsumer**, **ProxyPullConsumer**, **StructuredProxyPushConsumer**, **StructuredProxyPullConsumer**, **SequenceProxyPushConsumer**, **SequenceProxyPullConsumer,** **TypedProxyPushConsumer**, and **TypedProxyPullConsumer**.]

3.4.1.2 MyAdmin

The **MyAdmin** attribute is a readonly attribute that should be set upon creation of each proxy consumer instance to maintain the reference of the instance supporting the **SupplierAdmin** interface that created it.

3.4.1.3 obtain_subscription_types

The **obtain_subscription_types** operation returns a list of event type names. This returned list represents the names of event types which consumers connected to the channel are interested in receiving. Consumers express their interest in receiving particular types of events by configuring filters associated with the proxy suppliers to which they are connected to encapsulate constraints which express subscriptions to specific event instances. Such subscriptions could be based on the types and/or contents of events. The proxy suppliers extract the event type information from these subscriptions, and share it with the proxy consumer objects connected to the same channel. Supplier objects can thus obtain this information from the channel by invoking the **obtain_subscription_types** operation on the proxy consumer object to which they are connected. This information enables suppliers to suppress sending types of events to the channel in which no consumer is currently interested.

Note that suppliers can also receive updates to subscription information automatically by enabling the channel to invoke the **subscription_change** operation they support through inheritence of the **CosNotifyComm::NotifySubscribe** interface each time a new type of event is added or removed through modification of filters. The **obtain_subscription_types** operation accepts as input a flag that enables synchronization between the subscription information obtain through these automatic updates, and that obtained through invocation of **obtain_subscription_types**.

The table below summarizes the possible values and associated meanings this flag can take on.

Table 3-1 Possible values and associated meanings for “mode” argument

Value	Meaning
ALL_NOW_UPDATES_OFF	The invocation should return the current list of subscription types known by the target proxy consumer, but subsequent automatic sending of subscription update information should be disabled.
ALL_NOW_UPDATES_ON	The invocation should return the current list of subscription types known by the proxy consumer, and subsequent automatic sending of subscription update information should be enabled. Note these two actions should be atomic, guaranteeing that the supplier connected to the proxy consumer does not miss any subscription change updates that may be issued after the operation returns.
NONE_NOW_UPDATES_OFF	The invocation should not return any data, and should disable the subsequent automatic sending of subscription update information.
NONE_NOW_UPDATES_ON	The invocation should not return any data, but should enable the subsequent automatic sending of subscription update information.

3.4.1.4 *validate_event_qos*

The **validate_event_qos** operation accepts as input a sequence of QoS property name-value pairs which specify a set of QoS settings that a client is interested in setting on a per-event basis. Note that the QoS property settings contained in the optional header fields of a Structured Event may differ from those that are configured on a given proxy object. This operation is essentially a check to see if the target proxy object will honor the setting of a set of QoS properties on a per-event basis to values that may conflict with those set on the proxy itself. If any of the requested settings would not be honored by the target object on a per-event basis, the operation raises the UnsupportedQoS exception. This exception contains as data a sequence of data structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes which might be returned are described in Table 2-5 on page 2-46.

If all requested QoS property value settings could be satisfied by the target object, the operation returns successfully with an output parameter that contains a sequence of **PropertyRange** data structures. Each element in this sequence includes the name of a an additional QoS property whose setting is supported by the target object on a per-

event basis and which could have been included on the input list while still resulting in a successful return from the operation. Each element also includes the range of values that would have been acceptable for each such property.

3.4.2 The ProxySupplier Interface

The **ProxySupplier** interface is intended to be an abstract interface that is inherited by the different varieties of proxy suppliers that can be instantiated within a notification channel. It encapsulates the behaviors common to all Notification Service proxy suppliers. In particular, the **ProxySupplier** interface inherits the **QoSAdmin** interface defined within the **CosNotification** module, and the **FilterAdmin** interface defined within the **CosNotifyFilter** module. The former inheritance enables all proxy suppliers to administer a list of associated QoS properties, while the latter inheritance enables all proxy suppliers to administer a list of associated filter objects. Locally, the **ProxySupplier** interface defines a readonly attribute that should be set upon creation of each proxy supplier instance to indicate the specific type of proxy supplier the instance represents, and a readonly attribute which maintains a reference to the **ConsumerAdmin** object that created it. In addition, the **ProxySupplier** interface defines attributes that associate with each proxy supplier two mapping filter objects, one for priority and one for lifetime. As described in Section 2.3.1, "Mapping Filter Objects," on page 2-21, these mapping filter objects enable proxy suppliers to be configured to alter the way they treat events with respect to their priority and lifetime based on the type and contents of each individual event. Lastly, the **ProxySupplier** interface defines an operation that returns the list of event types that a given proxy supplier could potentially forward to its associated consumer, and an operation which can be queried to determine which message level QoS properties can be set on a per-event basis.

3.4.2.1 MyType

The **MyType** attribute is a readonly attribute that should be set upon creation of each proxy supplier instance to indicate the specific type of proxy supplier the instance represents. Enumerations are possible to distinguish the type of proxy supplier among the following possibilities: **ProxyPushSupplier**, **ProxyPullSupplier,** **StructuredProxyPushSupplier**, **StructuredProxyPullSupplier,** **SequenceProxyPushSupplier**, **SequenceProxyPullSupplier,** **TypedProxyPushSupplier**, and **TypedProxyPullSupplier**.

3.4.2.2 MyAdmin

The **MyAdmin** attribute is a readonly attribute that should be set upon creation of each proxy supplier instance to maintain the reference of the instance supporting the **ConsumerAdmin** interface that created it.

3.4.2.3 *priority_filter*

The **priority_filter** attribute contains a reference to an object supporting the **MappingFilter** interface defined in the **CosNotifyFilter** module. Such an object encapsulates a list of constraint-value pairs, where each constraint is a boolean expression based on the type and contents of an event, and the value is a possible priority setting for the event. Upon receipt of each event by a proxy supplier object whose **priority_filter** attribute contains a non-nil reference, the proxy supplier will invoke the appropriate variant of the **match** operation supported by the mapping filter object. The mapping filter object will proceed to apply its encapsulated constraints to the event, and return the one with the highest associated priority setting that evaluates to TRUE, or else its associated **default_value** if no constraints evaluate to TRUE. Upon return from the **match** operation, if the output parameter is TRUE, the proxy supplier treats the event with respect to its priority according to the return value, as opposed to a priority setting contained within the event. If the output parameter is FALSE, the proxy supplier will treat the event with respect to its priority according to the value set for the priority property in the event header if this property is present, otherwise it will use the output parameter returned from the **match** operation (i.e., the default value of the mapping filter object).

3.4.2.4 *lifetime_filter*

The **lifetime_filter** attribute contains a reference to an object supporting the **MappingFilter** interface defined in the **CosNotifyFilter** module. Such an object encapsulates a list of constraint-value pairs, where each constraint is a boolean expression based on the type and contents of an event, and the value is a possible lifetime setting for the event. Upon receipt of each event by a proxy supplier object whose **lifetime_filter** attribute contains a non-nil reference, the proxy supplier will invoke the appropriate variant of the *match* operation supported by the mapping filter object. The mapping filter object will proceed to apply its encapsulated constraints to the event, and return the one with the highest associated lifetime setting which evaluates to TRUE, or else its associated **default_value** if no constraints evaluate to TRUE. Upon return from the **match** operation, if the output parameter is TRUE, the proxy supplier treats the event with respect to its lifetime according to the return value, as opposed to a lifetime setting contained within the event. If the output parameter is FALSE, the proxy supplier will treat the event with respect to its lifetime according to the value set for the lifetime property in the event header if this property is present, otherwise it will use the output parameter returned from the **match** operation (i.e., the default value of the mapping filter object).

3.4.2.5 *obtain_offered_types*

The **obtain_offered_types** operation returns a list of event type names. Each element of the returned list names a type of event that the target proxy supplier object could potentially forward to its associated consumer. Note that through inheritance, all proxy consumer objects will support the **NotifyPublish** interface defined in the **CosNotifyComm** module. This interface supports the **offer_change** operation, which can be invoked by suppliers each time there is a change to the list of event types they plan to supply to their associated consumer. Thus, this mechanism relies on event

suppliers keeping the channel informed of the types of events they plan to supply by invoking the **offer_change** operation on their associated proxy consumer object. Internally to the channel, the proxy consumers will share the information about event types that will be supplied to the channel with the proxy supplier objects associated with the channel. This enables consumers to discover the types of events that could be supplied to them by the channel by invoking the **obtain_offered_types** operation on their associated proxy supplier.

Note that as mentioned above, consumers can also receive updates to offer information automatically by enabling the channel to invoke the **offer_change** operation they support through inheritence of the **CosNotifyComm::NotifyPublish** interface each time a supplier informs the channel of a change to the types of events they plan to supply. The **obtain_offered_types** operation accepts as input a flag that enables synchronization between the offer information obtained through these automatic updates, and that obtained through invocation of **obtain_offered_types**. The possible values and associated meanings this flag can take on are similar to those summarized in Table 3-1 on page 3-52, except that the information being shared is “offer” information instead of “subscription” information.

3.4.2.6 validate_event_qos

The **validate_event_qos** operation accepts as input a sequence of QoS property name-value pairs which specify a set of QoS settings that a client is interested in setting on a per-event basis. Note that the QoS property settings contained in the optional header fields of a Structured Event may differ from those that are configured on a given proxy object. This operation is essentially a check to see if the target proxy object will honor the setting of a set of QoS properties on a per-event basis to values that may conflict with those set on the proxy itself. If any of the requested settings would not be honored by the target object on a per-event basis, the operation raises the UnsupportedQoS exception. This exception contains as data a sequence of data structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes which might be returned are described in Table 2-5 on page 2-46.

If all requested QoS property value settings could be satisfied by the target object, the operation returns successfully with an output parameter that contains a sequence of **PropertyRange** data structures. Each element in this sequence includes the name of an additional QoS property whose setting is supported by the target object on a per-event basis and which could have been included on the input list while still resulting in a successful return from the operation. Each element also includes the range of values that would have been acceptable for each such property.

3.4.3 The ProxyPushConsumer Interface

The **ProxyPushConsumer** interface supports connections to the channel by suppliers who will push events to the channel as untyped Anys.

Through inheritance of the **ProxyConsumer** interface, the **ProxyPushConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **SupplierAdmin** object which created it. In addition, this inheritance implies that a **ProxyPushConsumer** instance supports an operation which will return the list of event types which consumers connected to the same channel are interested in receiving, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **ProxyPushConsumer** interface also inherits from the **PushConsumer** interface defined within the **CosNotifyComm** module. This interface supports the **push** operation, which the supplier connected to a **ProxyPushConsumer** instance will invoke to send an event to the channel in the form of an Any, and the operation required to disconnect the **ProxyPushConsumer** from its associated supplier. In addition, since the inherited **PushConsumer** interface inherits the **CosNotifyComm::NotifyPublish** interface, a supplier connected to an instance supporting the **ProxyPushConsumer** interface can inform it whenever the list of event types the supplier plans to supply changes.

Finally, the **ProxyPushConsumer** interface defines the operation which can be invoked by a push supplier to establish the connection over which the push supplier will send events to the channel. Note that this can be either a pure event service style, or a notification service style push supplier.

3.4.3.1 connect_any_push_supplier

The **connect_any_push_supplier** operation accepts as an input parameter the reference to an object supporting the **PushSupplier** interface defined within the **CosEventComm** module of the OMG Event Service. This reference is that of a supplier that plans to push events to the channel with which the target object is associated in the form of untyped Anys. This operation is thus invoked in order to establish a connection between a push-style supplier of events in the form of Anys, and the notification channel. Once established, the supplier can proceed to send events to the channel by invoking the **push** operation supported by the target **ProxyPushConsumer** instance. If the target object of this operation is already connected to a push supplier object, the AlreadyConnected exception will be raised.

Note that because the **PushSupplier** interface defined in the **CosNotifyComm** module inherits from the **PushSupplier** interface defined in the **CosEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style push supplier. The only difference between the two are that the latter also supports the **NotifySubscribe** interface, and thus can be the target of **subscription_change** invocations. The implementation of the **ProxyPushConsumer** interface should attempt to narrow the input parameter to **CosNotifyComm::PushSupplier** in order to determine which style of push supplier is connecting to it.

3.4.4 The StructuredProxyPushConsumer Interface

The **StructuredProxyPushConsumer** interface supports connections to the channel by suppliers who will push events to the channel as Structured Events. Through inheritance of the **ProxyConsumer** interface, the **StructuredProxyPushConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **SupplierAdmin** object which created it. In addition, this inheritance implies that a **StructuredProxyPushConsumer** instance supports an operation which will return the list of event types which consumers connected to the same channel are interested in receiving, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **StructuredProxyPushConsumer** interface also inherits from the **StructuredPushConsumer** interface defined in the **CosNotifyComm** module. This interface supports the operation that enables a supplier of Structured Events to push them to the **StructuredProxyPushConumer**, and also the operation that can be invoked to close down the connection from the supplier to the **StructuredProxyPushConsumer**. In addition, since the **StructuredPushConsumer** interface inherits from the **NotifyPublish** interface, a supplier can inform the **StructuredProxyPushConsumer** to which it is connected whenever the list of event types it plans to supply to the channel changes.

Lastly, the **StructuredProxyPushConsumer** interface defines a method that can be invoked by a push-style supplier of Structured Events in order to establish a connection between the supplier and a notification channel over which the supplier will proceed to send events.

3.4.4.1 connect_structured_push_supplier

The **connect_structured_push_supplier** operation accepts as an input parameter the reference to an object supporting the **StructuredPushSupplier** interface defined within the **CosNotifyComm** module. This reference is that of a supplier which plans to push events to the channel with which the target object is associated in the form of Structured Events. This operation is thus invoked in order to establish a connection between a push-style supplier of events in the form of Structured Events, and the notification channel. Once established, the supplier can proceed to send events to the channel by invoking the **push_structured_event** operation supported by the target **StructuredProxyPushConsumer** instance. If the target object of this operation is already connected to a push supplier object, the AlreadyConnected exception will be raised.

3.4.5 The SequenceProxyPushConsumer Interface

The **SequenceProxyPushConsumer** interface supports connections to the channel by suppliers who will push events to the channel as sequences of Structured Events. Through inheritance of the **ProxyConsumer** interface, the **SequenceProxyPushConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute

containing the reference of the **SupplierAdmin** object which created it. In addition, this inheritance implies that a **SequenceProxyPushConsumer** instance supports an operation which will return the list of event types which consumers connected to the same channel are interested in receiving, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **SequenceProxyPushConsumer** interface also inherits from the **SequencePushConsumer** interface defined in the **CosNotifyComm** module. This interface supports the operation which enables a supplier of sequences of Structured Events to push them to the **SequenceProxyPushConsumer**, and also the operation that can be invoked to close down the connection from the supplier to the **SequenceProxyPushConsumer**. In addition, since the **SequencePushConsumer** interface inherits from the **NotifyPublish** interface, a supplier can inform the **SequenceProxyPushConsumer** to which it is connected whenever the list of event types it plans to supply to the channel changes.

Lastly, the **SequenceProxyPushConsumer** interface defines a method that can be invoked by a push-style supplier of sequences of Structured Events in order to establish a connection between the supplier and a notification channel over which the supplier will proceed to send events.

3.4.5.1 connect_sequence_push_supplier

The **connect_sequence_push_supplier** operation accepts as an input parameter the reference to an object supporting the **SequencePushSupplier** interface defined within the **CosNotifyComm** module. This reference is that of a supplier, which plans to push events to the channel with which the target object is associated in the form of sequences of Structured Events. This operation is thus invoked in order to establish a connection between a push-style supplier of events in the form of sequences of Structured Events, and the notification channel. Once established, the supplier can proceed to send events to the channel by invoking the **push_structured_events** operation supported by the target **SequenceProxyPushConsumer** instance. If the target object of this operation is already connected to a push supplier object, the AlreadyConnected exception will be raised.

3.4.6 The ProxyPullSupplier Interface

The **ProxyPullSupplier** interface supports connections to the channel by consumers who will pull events from the channel as untyped Anys.

Through inheritance of the **ProxySupplier** interface, the **ProxyPullSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, mapping filters for event priority and lifetime, and a readonly attribute containing the reference of the **ConsumerAdmin** object that created it. In addition, this inheritance implies that a **ProxyPullSupplier** instance supports an operation which will return the list of event types which the proxy supplier will potentially by supplying, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **ProxyPullSupplier** interface also inherits from the **PullSupplier** interface defined within the **CosNotifyComm** module. This interface supports the **pull** and **try_pull** operations which the consumer connected to a **ProxyPullSupplier** instance will invoke to receive an event from the channel in the form of an Any, and the operation required to disconnect the **ProxyPullSupplier** from its associated consumer. In addition, since the inherited **PullSupplier** interface inherits the **CosNotifyComm::NotifySubscribe** interface, an instance supporting the **ProxyPullSupplier** interface can be informed whenever the list of event types that the consumer connected to it is interested in receiving changes.

Finally, the **ProxyPullSupplier** interface defines the operation which can be invoked by a pull consumer to establish the connection over which the pull consumer will receive events from the channel. Note that this can be either a pure event service style, or a notification service style pull consumer.

3.4.6.1 connect_any_pull_consumer

The **connect_any_pull_consumer** operation accepts as an input parameter the reference to an object supporting the **PullConsumer** interface defined within the **CosEventComm** module. This reference is that of a consumer that plans to pull events from the channel with which the target object is associated in the form of untyped Anys. This operation is thus invoked in order to establish a connection between a pull-style consumer of events in the form of Anys, and the notification channel. Once established, the consumer can proceed to receive events from the channel by invoking the **pull** or **try_pull** operations supported by the target **ProxyPullSupplier** instance. If the target object of this operation is already connected to a pull consumer object, the AlreadyConnected exception will be raised.

Note that because the **PullConsumer** interface defined in the **CosNotifyComm** module inherits from the **PullConsumer** interface defined in the **CosEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style pull consumer. The only difference between the two are that the latter also supports the **NotifyPublish** interface, and thus can be the target of **offer_change** invocations. The implementation of the **ProxyPullSupplier** interface should attempt to narrow the input parameter to **CosNotifyComm::PullConsumer** in order to determine which style of pull consumer is connecting to it.

3.4.7 The StructuredProxyPullSupplier Interface

The **StructuredProxyPullSupplier** interface supports connections to the channel by consumers who will pull events from the channel as Structured Events. Through inheritance of the **ProxySupplier** interface, the **StructuredProxyPullSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **ConsumerAdmin** object that created it. In addition, this inheritance implies that a **StructuredProxyPullSupplier** instance supports an operation that will return the list of event types, which the proxy supplier will potentially by supplying, and an operation which can return information about the instance’s ability to accept a per-event QoS request.

The **StructuredProxyPullSupplier** interface also inherits from the **StructuredPullSupplier** interface defined in the **CosNotifyComm** module. This interface supports the operations that enable a consumer of Structured Events to pull them from the **StructuredProxyPullSupplier**, and also the operation that can be invoked to close down the connection from the consumer to the **StructuredProxyPullSupplier**. In addition, since the **StructuredPullSupplier** interface inherits from the **NotifySubscribe** interface, a **StructuredProxyPullSupplier** can be notified whenever the list of event types, which its associated consumer is interested in receiving changes. This notification occurs via the callback mechanism described in Section 2.3, "Event Filtering with Filter Objects," on page 2-17.

Lastly, the **StructuredProxyPullSupplier** interface defines a method that can be invoked by a pull-style consumer of Structured Events in order to establish a connection between the consumer and a notification channel over which the consumer will proceed to receive events.

3.4.7.1 connect_structured_pull_consumer

The **connect_structured_pull_consumer** operation accepts as an input parameter the reference to an object supporting the **StructuredPullConsumer** interface defined within the **CosNotifyComm** module. This reference is that of a consumer that plans to pull events from the channel to which the target object is associated in the form of Structured Events. This operation is thus invoked in order to establish a connection between a pull-style consumer of events in the form of Structured Events, and the notification channel. Once established, the consumer can proceed to receive events from the channel by invoking the **pull_structured_event** or **try_pull_structured_event** operations supported by the target **StructuredProxyPullSupplier** instance. If the target object of this operation is already connected to a pull consumer object, the AlreadyConnected exception will be raised.

3.4.8 The SequenceProxyPullSupplier Interface

The **SequenceProxyPullSupplier** interface supports connections to the channel by consumers who will pull events from the channel as sequences of Structured Events. Through inheritance of the **ProxySupplier** interface, the **SequenceProxyPullSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **ConsumerAdmin** object that created it. In addition, this inheritance implies that a **SequenceProxyPullSupplier** instance supports an operation that will return the list of event types, which the proxy supplier will potentially be supplying, and an operation that can return information about the instance's ability to accept a per-event QoS request.

The **SequenceProxyPullSupplier** interface also inherits from the **SequencePullSupplier** interface defined in the **CosNotifyComm** module. This interface supports the operations that enable a consumer of sequences of Structured Events to pull them from the **SequenceProxyPullSupplier**, and the operation that

can be invoked to close down the connection from the consumer to the **SequenceProxyPullSupplier**. In addition, since the **SequencePullSupplier** interface inherits from the **NotifySubscribe** interface, a **SequenceProxyPullSupplier** can be notified whenever the list of event types, which its associated consumer is interested in receiving changes. This notification occurs via the callback mechanism described in Section 2.3, "Event Filtering with Filter Objects," on page 2-17.

Lastly, the **SequenceProxyPullSupplier** interface defines a method that can be invoked by a pull-style consumer of sequences of Structured Events in order to establish a connection between the consumer and a notification channel over which the consumer will proceed to receive events.

3.4.8.1 connect_sequence_pull_consumer

The **connect_sequence_pull_consumer** operation accepts as an input parameter the reference to an object supporting the **SequencePullConsumer** interface defined within the **CosNotifyComm** module. This reference is that of a consumer that plans to pull events from the channel to which the target object is associated in the form of sequences of Structured Events. This operation is thus invoked in order to establish a connection between a pull-style consumer of events in the form of sequences of Structured Events, and the notification channel. Once established, the consumer can proceed to receive events from the channel by invoking the **pull_structured_events** or **try_pull_structured_events** operations supported by the target **SequenceProxyPullSupplier** instance. If the target object of this operation is already connected to a pull consumer object, the AlreadyConnected exception will be raised.

3.4.9 The ProxyPullConsumer Interface

The **ProxyPullConsumer** interface supports connections to the channel by suppliers who will make events available for pulling to the channel as untyped Anys.

Through inheritance of the **ProxyConsumer** interface, the **ProxyPullConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **SupplierAdmin** object that created it. In addition, this inheritance implies that a **ProxyPullConsumer** instance supports an operation that will return the list of event types that consumers connected to the same channel are interested in receiving, and an operation that can return information about the instance's ability to accept a per-event QoS request.

The **ProxyPullConsumer** interface also inherits from the **PullConsumer** interface defined within the **CosEventComm** module of the OMG Event Service. This interface supports the operation required to disconnect the **ProxyPullConsumer** from its associated supplier. In addition, since the inherited **PullConsumer** interface inherits the **CosNotifyComm::NotifyPublish** interface, a supplier connected to an instance supporting the **ProxyPullConsumer** interface can inform it whenever the list of event types the supplier plans to supply changes.

Finally, the **ProxyPullConsumer** interface defines the operation which can be invoked by a pull supplier to establish the connection over which the pull supplier will send events to the channel. Note that this can be either a pure event service style, or a notification service style pull supplier. The **ProxyPullConsumer** interface also defines a pair of operations that can suspend and resume the connection between a **ProxyPullConsumer** instance and its associated **PullSupplier**. During the time such a connection is suspended, the **ProxyPullConsumer** will not attempt to pull events from its associated **PullSupplier**.

3.4.9.1 connect_any_pull_supplier

The **connect_any_pull_supplier** operation accepts as an input parameter the reference to an object supporting the **PullSupplier** interface defined within the **CosEventComm** module. This reference is that of a supplier which plans to make events available for pulling to the channel with which the target object is associated in the form of untyped Anys. This operation is thus invoked in order to establish a connection between a pull-style supplier of events in the form of Anys, and the notification channel. Once established, the channel can proceed to receive events from the supplier by invoking the **pull** or **try_pull** operations supported by the supplier (whether the channel will invoke **pull** or **try_pull**, and the frequency with which it will perform such invocations, is a detail which is specific to the implementation of the channel). If the target object of this operation is already connected to a pull supplier object, the AlreadyConnected exception will be raised. An implementation of the **ProxyPullConsumer** interface may impose additional requirements on the interface supported by a pull supplier (e.g., it may be designed to invoke some operation other than **pull** or **try_pull** in order to receive events). If the pull supplier being connected does not meet those requirements, this operation raises the TypeError exception.

Note that because the **PullSupplier** interface defined in the **CosNotifyComm** module inherits from the **PullSupplier** interface defined in the **CosEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style pull supplier. The only difference between the two is that the latter also supports the **NotifySubscribe** interface, and thus can be the target of **subscription_change** invocations. The implementation of the **ProxyPullConsumer** interface should attempt to narrow the input parameter to **CosNotifyComm::PullSupplier** in order to determine which style of pull supplier is connecting to it.

3.4.9.2 suspend_connection

The **suspend_connection** operation causes the target object supporting the **ProxyPullConsumer** interface to stop attempting to pull events (using **pull** or **try_pull**) from the **PullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **PullSupplier** has been connected to the target object when this operation is invoked,

the NotConnected exception is raised. Otherwise, the **ProxyPullConsumer** will not attempt to pull events from the **PullSupplier** connected to it until **resume_connection** is subsequently invoked.

3.4.9.3 resume_connection

The **resume_connection** operation causes the target object supporting the **ProxyPullConsumer** interface to resume attempting to pull events (using **pull** or **try_pull**) from the **PullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception is raised. If no **PullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **ProxyPullConsumer** will resume attempting to pull events from the **PullSupplier** connected to it.

3.4.10 The StructuredProxyPullConsumer Interface

The **StructuredProxyPullConsumer** interface supports connections to the channel by suppliers who will make events available for pulling to the channel as Structured Events. Through inheritance of the **ProxyConsumer** interface, the **StructuredProxyPullConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **SupplierAdmin** object that created it. In addition, this inheritance implies that a **StructuredProxyPullConsumer** instance supports an operation that will return the list of event types which consumers connected to the same channel are interested in receiving, and an operation that can return information about the instance's ability to accept a per-event QoS request.

The **StructuredProxyPullConsumer** interface also inherits from the **StructuredPullConsumer** interface defined in the **CosNotifyComm** module. This interface supports the operation that can be invoked to close down the connection from the supplier to the **StructuredProxyPullConsumer**. In addition, since the **StructuredPullConsumer** interface inherits from the **NotifyPublish** interface, a supplier can inform the **StructuredProxyPullConsumer** to which it is connected whenever the list of event types it plans to supply to the channel changes.

Lastly, the **StructuredProxyPullConsumer** interface defines a method that can be invoked by a pull-style supplier of Structured Events in order to establish a connection between the supplier and a notification channel over which the supplier will proceed to send events. The **StructuredProxyPullConsumer** interface also defines a pair of operations that can suspend and resume the connection between a **StructuredProxyPullConsumer** instance and its associated **StructuredPullSupplier**. During the time such a connection is suspended, the **StructuredProxyPullConsumer** will not attempt to pull events from its associated **StructuredPullSupplier**.

3.4.10.1 *connect_structured_pull_supplier*

The **connect_structured_pull_supplier** operation accepts as an input parameter the reference to an object supporting the **StructuredPullSupplier** interface defined within the **CosNotifyComm** module. This reference is that of a supplier which plans to make events available for pulling to the channel with which the target object is associated in the form of Structured Events. This operation is thus invoked in order to establish a connection between a pull-style supplier of events in the form of Structured Events, and the notification channel. Once established, the channel can proceed to receive events from the supplier by invoking the **pull_structured_event** or **try_pull_structured_event** operations supported by the supplier (whether the channel will invoke **pull_structured_event** or **try_pull_structured_event**, and the frequency with which it will perform such invocations, is a detail which is specific to the implementation of the channel). If the target object of this operation is already connected to a pull supplier object, the AlreadyConnected exception will be raised. An implementation of the **StructuredProxyPullConsumer** interface may impose additional requirements on the interface supported by a pull supplier (e.g., it may be designed to invoke some operation other than **pull_structured_event** or **try_pull_structured_event** in order to receive events). If the pull supplier being connected does not meet those requirements, this operation raises the TypeError exception.

3.4.10.2 *suspend_connection*

The **suspend_connection** operation causes the target object supporting the **StructuredProxyPullConsumer** interface to stop attempting to pull events (using **pull** or **try_pull**) from the **StructuredPullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **StructuredPullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **StructuredProxyPullConsumer** will not attempt to pull events from the **StructuredPullSupplier** connected to it until **resume_connection** is subsequently invoked.

3.4.10.3 *resume_connection*

The **resume_connection** operation causes the target object supporting the **StructuredProxyPullConsumer** interface to resume attempting to pull events (using **pull** or **try_pull**) from the **StructuredPullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception is raised. If no **StructuredPullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **StructuredProxyPullConsumer** will resume attempting to pull events from the **StrucuturedPullSupplier** connected to it.

3.4.11 The SequenceProxyPullConsumer Interface

The **SequenceProxyPullConsumer** interface supports connections to the channel by suppliers who will make events available for pulling to the channel as sequences of Structured Events. Through inheritance of the **ProxyConsumer** interface, the **SequenceProxyPullConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **SupplierAdmin** object which created it. In addition, this inheritance implies that a **SequenceProxyPullConsumer** instance supports an operation which will return the list of event types which consumers connected to the same channel are interested in receiving, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **SequenceProxyPullConsumer** interface also inherits from the **SequencePullConsumer** interface defined in the **CosNotifyComm** module. This interface supports the operation which can be invoked to close down the connection from the supplier to the **SequenceProxyPullConsumer**. In addition, since the **SequencePullConsumer** interface inherits from the **NotifyPublish** interface, a supplier can inform the **SequenceProxyPullConsumer** to which it is connected whenever the list of event types it plans to supply to the channel changes.

Lastly, the **SequenceProxyPullConsumer** interface defines a method that can be invoked by a pull-style supplier of sequences of Structured Events in order to establish a connection between the supplier and a notification channel over which the supplier will proceed to send events. The **SequenceProxyPullConsumer** interface also defines a pair of operations which can suspend and resume the connection between a **SequenceProxyPullConsumer** instance and its associated **SequencePullSupplier**. During the time such a connection is suspended, the **SequenceProxyPullConsumer** will not attempt to pull events from its associated **SequencePullSupplier**.

3.4.11.1 connect_sequence_pull_supplier

The **connect_sequence_pull_supplier** operation accepts as an input parameter the reference to an object supporting the **SequencePullSupplier** interface defined within the **CosNotifyComm** module. This reference is that of a supplier that plans to make events available for pulling to the channel with which the target object is associated in the form of sequences of Structured Events. This operation is thus invoked in order to establish a connection between a pull-style supplier of events in the form of sequences of Structured Events, and the notification channel. Once established, the channel can proceed to receive events from the supplier by invoking the **pull_structured_events** or **try_pull_structured_events** operations supported by the supplier (whether the channel will invoke **pull_structured_events** or **try_pull_structured_events**, and the frequency with which it will perform such invocations, is a detail that is specific to the implementation of the channel). If the target object of this operation is already connected to a pull supplier object, the AlreadyConnected exception will be raised. An implementation of the **SequenceProxyPullConsumer** interface may impose additional requirements on the interface supported by a pull supplier (e.g., it may be designed to invoke some operation other than **pull_structured_events** or

try_pull_structured_events in order to receive events). If the pull supplier being connected does not meet those requirements, this operation raises the TypeError exception.

3.4.11.2 suspend_connection

The **suspend_connection** operation causes the target object supporting the **SequenceProxyPullConsumer** interface to stop attempting to pull events (using **pull** or **try_pull**) from the **SequencePullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **SequencePullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **SequenceProxyPullConsumer** will not attempt to pull events from the **SequencePullSupplier** connected to it until **resume_connection** is subsequently invoked.

3.4.11.3 resume_connection

The **resume_connection** operation causes the target object supporting the **SequenceProxyPullConsumer** interface to resume attempting to pull events (using **pull** or **try_pull**) from the **SequencePullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception is raised. If no **SequencePullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **SequenceProxyPullConsumer** will resume attempting to pull events from the **SequencePullSupplier** connected to it.

3.4.12 The ProxyPushSupplier Interface

The **ProxyPushSupplier** interface supports connections to the channel by consumers who will receive events from the channel as untyped Anys.

Through inheritance of the **ProxySupplier** interface, the **ProxyPushSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, mapping filters for event priority and lifetime, and a readonly attribute containing the reference of the **ConsumerAdmin** object which created it. In addition, this inheritance implies that a **ProxyPushSupplier** instance supports an operation that will return the list of event types which the proxy supplier will potentially by supplying, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **ProxyPushSupplier** interface also inherits from the **PushSupplier** interface defined within the **CosNotifyComm** module. This interface supports the operation required to disconnect the **ProxyPushSupplier** from its associated consumer. In addition, since the inherited **PushSupplier** interface inherits the

CosNotifyComm::NotifySubscribe interface, an instance supporting the **ProxyPushSupplier** interface can be informed whenever the list of event types that the consumer connected to it is interested in receiving changes.

Lastly, the **ProxyPushSupplier** interface defines the operation which can be invoked by a push consumer to establish the connection over which the push consumer will receive events from the channel. Note that this can be either a pure event service style, or a notification service style push consumer.The **ProxyPushSupplier** interface also defines a pair of operations that can suspend and resume the connection between a **ProxyPushSupplier** instance and its associated **PushConsumer**. During the time such a connection is suspended, the **ProxyPushSupplier** will accumulate events destined for the consumer but not transmit them until the connection is resumed.

3.4.12.1 *connect_any_push_consumer*

The **connect_any_push_consumer** operation accepts as an input parameter the reference to an object supporting the **PushConsumer** interface defined within the **CosEventComm** module. This reference is that of a consumer that will receive events from the channel with which the target object is associated in the form of untyped Anys. This operation is thus invoked in order to establish a connection between a push-style consumer of events in the form of Anys, and the notification channel. Once established, the **ProxyPushSupplier** will proceed to send events destined for the consumer to it by invoking its **push** operation. If the target object of this operation is already connected to a push consumer object, the AlreadyConnected exception will be raised. An implementation of the **ProxyPushSupplier** interface may impose additional requirements on the interface supported by a push consumer (e.g., it may be designed to invoke some operation other than **push** in order to transmit events). If the push consumer being connected does not meet those requirements, this operation raises the TypeError exception.

Note that because the **PushConsumer** interface defined in the **CosNotifyComm** module inherits from the **PushConsumer** interface defined in the **CosEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style push consumer. The only difference between the two are that the latter also supports the **NotifyPublish** interface, and thus can be the target of **offer_change** invocations. The implementation of the **ProxyPushSupplier** interface should attempt to narrow the input parameter to **CosNotifyComm::PushConsumer** in order to determine which style of push consumer is connecting to it.

3.4.12.2 *suspend_connection*

The **suspend_connection** operation causes the target object supporting the **ProxyPushSupplier** interface to stop sending events to the **PushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **PushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **ProxyPushSupplier** will not forward events to

the **PushConsumer** connected to it until **resume_connection** is subsequently invoked. During this time, the **ProxyPushSupplier** will continue to queue events destined for the **PushConsumer**, although events that time out prior to resumption of the connection will be discarded. Upon resumption of the connection, all queued events will be forwarded to the **PushConsumer**.

3.4.12.3 resume_connection

The **resume_connection** operation causes the target object supporting the **ProxyPushSupplier** interface to resume sending events to the **PushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception is raised. If no **PushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **ProxyPushSupplier** will resume forwarding events to the **PushConsumer** connected to it, including those that have been queued during the time the connection was suspended, and have not yet timed out.

3.4.13 The StructuredProxyPushSupplier Interface

The **StructuredProxyPushSupplier** interface supports connections to the channel by consumers who will receive events from the channel as Structured Events. Through inheritance of the **ProxySupplier** interface, the **StructuredProxyPushSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **ConsumerAdmin** object which created it. In addition, this inheritance implies that a **StructuredProxyPushSupplier** instance supports an operation which will return the list of event types which the proxy supplier will potentially be supplying, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **StructuredProxyPushSupplier** interface also inherits from the **StructuredPushSupplier** interface defined in the **CosNotifyComm** module. This interface supports the operation that can be invoked to close down the connection from the consumer to the **StructuredProxyPushSupplier**. In addition, since the **StructuredPushSupplier** interface inherits from the **NotifySubscribe** interface, a **StructuredProxyPushSupplier** can be notified whenever the list of event types which its associated consumer is interested in receiving changes. This notification occurs via the callback mechanism described in Section 2.3, "Event Filtering with Filter Objects," on page 2-17.

Lastly, the **StructuredProxyPushSupplier** interface defines the operation that can be invoked by a push consumer to establish the connection over which the push consumer will receive events from the channel. The **StructuredProxyPushSupplier** interface also defines a pair of operations that can suspend and resume the connection between a **StructuredProxyPushSupplier** instance and its associated

StructuredPushConsumer. During the time such a connection is suspended, the **StructuredProxyPushSupplier** will accumulate events destined for the consumer but not transmit them until the connection is resumed.

3.4.13.1 connect_structured_push_consumer

The **connect_structured_push_consumer** operation accepts as an input parameter the reference to an object supporting the **StructuredPushConsumer** interface defined within the **CosNotifyComm** module. This reference is that of a consumer that will receive events from the channel with which the target object is associated in the form of Structured Events. This operation is thus invoked in order to establish a connection between a push-style consumer of events in the form of Structured Events, and the notification channel. Once established, the **StructuredProxyPushSupplier** will proceed to send events destined for the consumer to it by invoking its **push_structured_event** operation. If the target object of this operation is already connected to a push consumer object, the AlreadyConnected exception will be raised. An implementation of the **StructuredProxyPushSupplier** interface may impose additional requirements on the interface supported by a push consumer (e.g., it may be designed to invoke some operation other than **push_structured_event** in order to transmit events). If the push consumer being connected does not meet those requirements, this operation raises the TypeError exception.

3.4.13.2 suspend_connection

The **suspend_connection** operation causes the target object supporting the **StructuredProxyPushSupplier** interface to stop sending events to the **StructuredPushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **StructuredPushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **StructuredProxyPushSupplier** will not forward events to the **StructuredPushConsumer** connected to it until **resume_connection** is subsequently invoked. During this time, the **StructuredProxyPushSupplier** will continue to queue events destined for the **StructuredPushConsumer**, although events that time out prior to resumption of the connection will be discarded. Upon resumption of the connection, all queued events will be forwarded to the **StructuredPushConsumer**.

3.4.13.3 resume_connection

The **resume_connection** operation causes the target object supporting the **StructuredProxyPushSupplier** interface to resume sending events to the **StructuredPushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the

ConnectionAlreadyActive exception is raised. If no **StructuredPushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **StructuredProxyPushSupplier** will resume forwarding events to the **StructuredPushConsumer** connected to it, including those which have been queued during the time the connection was suspended, and have not yet timed out.

3.4.14 The SequenceProxyPushSupplier Interface

The **SequenceProxyPushSupplier** interface supports connections to the channel by consumers who will receive events from the channel as sequences of Structured Events. Through inheritance of the **ProxySupplier** interface, the **SequenceProxyPushSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the reference of the **ConsumerAdmin** object which created it. In addition, this inheritance implies that a **SequenceProxyPushSupplier** instance supports an operation that will return the list of event types which the proxy supplier will potentially be supplying, and an operation that can return information about the instance's ability to accept a per-event QoS request.

The **SequenceProxyPushSupplier** interface also inherits from the **SequencePushSupplier** interface defined in the **CosNotifyComm** module. This interface supports the operation that can be invoked to close down the connection from the consumer to the **SequenceProxyPushSupplier**. In addition, since the **SequencePushSupplier** interface inherits from the **NotifySubscribe** interface, a **SequenceProxyPushSupplier** can be notified whenever the list of event types which its associated consumer is interested in receiving changes. This notification occurs via the callback mechanism described in Section 2.3, "Event Filtering with Filter Objects," on page 2-17.

Lastly, the **SequenceProxyPushSupplier** interface defines the operation that can be invoked by a push consumer to establish the connection over which the push consumer will receive events from the channel. The **SequenceProxyPushSupplier** interface also defines a pair of operations that can suspend and resume the connection between a **SequenceProxyPushSupplier** instance and its associated **SequencePushConsumer**. During the time such a connection is suspended, the **SequenceProxyPushSupplier** will accumulate events destined for the consumer but not transmit them until the connection is resumed.

3.4.14.1 connect_sequence_push_consumer

The **connect_sequence_push_consumer** operation accepts as an input parameter the reference to an object supporting the **SequencePushConsumer** interface defined within the **CosNotifyComm** module. This reference is that of a consumer which will receive events from the channel with which the target object is associated in the form of sequences of Structured Events. This operation is thus invoked in order to establish a connection between a push-style consumer of events in the form of sequences of Structured Events, and the notification channel. Once established, the **SequenceProxyPushSupplier** will proceed to send events destined for the

consumer to it by invoking its **push_structured_events** operation. If the target object of this operation is already connected to a push consumer object, the AlreadyConnected exception will be raised. An implementation of the **SequenceProxyPushSupplier** interface may impose additional requirements on the interface supported by a push consumer (e.g., it may be designed to invoke some operation other than **push_structured_events** in order to transmit events). If the push consumer being connected does not meet those requirements, this operation raises the TypeError exception.

3.4.14.2 suspend_connection

The **suspend_connection** operation causes the target object supporting the **SequenceProxyPushSupplier** interface to stop sending events to the **SequencePushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **SequencePushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **SequenceProxyPushSupplier** will not forward events to the **SequencePushConsumer** connected to it until **resume_connection** is subsequently invoked. During this time, the **SequenceProxyPushSupplier** will continue to queue events destined for the **SequencePushConsumer**, although events that time out prior to resumption of the connection will be discarded. Upon resumption of the connection, all queued events will be forwarded to the **SequencePushConsumer**.

3.4.14.3 resume_connection

The **resume_connection** operation causes the target object supporting the **SequenceProxyPushSupplier** interface to resume sending events to the **SequencePushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception is raised. If no **SequencePushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **SequenceProxyPushSupplier** will resume forwarding events to the **SequencePushConsumer** connected to it, including those which have been queued during the time the connection was suspended, and have not yet timed out.

3.4.15 The ConsumerAdmin Interface

The **ConsumerAdmin** interface defines the behavior supported by objects which create and manage lists of proxy supplier objects within a Notification Service event channel. Recall that a Notification Service event channel can have any number of **ConsumerAdmin** instances associated with it. Each such instance is responsible for creating and managing a list of proxy supplier objects that share a common set of QoS

property settings, and a common set of filter objects. This feature enables clients to conveniently group proxy supplier objects within a channel into groupings that each support a set of consumers with a common set of QoS requirements and event subscriptions.

The **ConsumerAdmin** interface inherits the **QoSAdmin** interface defined within the **CosNotification** module, enabling each **ConsumerAdmin** instance to manage a set of QoS property settings. These QoS property settings are assigned as the default QoS property settings for any proxy supplier object created by a **ConsumerAdmin** instance. In addition, the **ConsumerAdmin** interface inherits from the **FilterAdmin** interface defined within the **CosNotifyFilter** module, enabling each **ConsumerAdmin** instance to maintain a list of filter objects. These filter objects encapsulate subscriptions that will apply to all proxy supplier objects that have been created by a given **ConsumerAdmin** instance. In order to enable optimizing the notification of a group of proxy supplier objects that have been created by the same **ConsumerAdmin** instance of changes to these shared filter objects, the **ConsumerAdmin** interface also inherits from the **NotifySubscribe** interface defined in the **CosNotifyComm** module. This inheritance enables a **ConsumerAdmin** instance to be registered as the callback object for notification of subscription changes made upon filter objects.

The **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module also inherits from the **ConsumerAdmin** interface defined in the **CosEventChannelAdmin** module. This inheritance enables clients to use the **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module to create pure OMG Event Service style proxy supplier objects. Proxy supplier objects created in this manner may not support configuration of QoS properties, and may not have associated filter objects. In addition, proxy supplier objects created through the inherited **ConsumerAdmin** interface will not have unique identifiers associated with them, whereas proxy supplier objects created by invoking the operations supported by the **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module will.

Locally, the **ConsumerAdmin** interface supports a readonly attribute which maintains a reference to the **EventChannel** instance that created a given **ConsumerAdmin** instance. The **ConsumerAdmin** interface also supports a readonly attribute which contains a numeric identifier which will be assigned to an instance supporting this interface by its associated Notification Service event channel upon creation of the **ConsumerAdmin** instance. This identifier will be unique among all **ConsumerAdmin** instances created by a given channel.

As described above, due to inheritance from the **FilterAdmin** interface, a **ConsumerAdmin** can maintain a list of filter objects that will be applied to all proxy suppliers it creates. Also recall that each proxy supplier may itself support a list of filter objects that apply only it. When combining multiple filter objects within each of these two lists of filter objects that may be associated with a given proxy supplier, OR semantics are applied. However when combining these two lists during the evaluation of a given event, either AND or OR semantics may be applied. The choice is

determined by an input flag upon creation of the **ConsumerAdmin**, and the operator that will be used for this purpose by a given **ConsumerAdmin** is maintained in a readonly attribute.

The **ConsumerAdmin** interface also supports attributes which maintain references to priority and lifetime mapping filter objects. These mapping filter objects will be applied to all proxy supplier objects created by a given **ConsumerAdmin** instance.

Each **ConsumerAdmin** instance assigns a unique numeric identifier to each proxy supplier object it maintains. The **ConsumerAdmin** interface supports attributes which maintain the list of these unique identifiers associated with the proxy pull and the proxy push suppliers created by a given **ConsumerAdmin** instance. The **ConsumerAdmin** interface also supports an operation which, given the unique identifier of a proxy supplier a given **ConsumerAdmin** instance has created as input, will return the object reference of that proxy supplier object. Additionally, the **ConsumerAdmin** interface supports operations that can create the various styles of proxy supplier objects supported by the Notification Service event channel. Finally, because clients of a given Notification Service event channel can create any number of **ConsumerAdmin** instances, a **destroy** operation is provided by this interface so that clients can clean up instances that are no longer needed.

3.4.15.1 MyID

The **MyID** attribute is a readonly attribute that maintains the unique identifier of the target **ConsumerAdmin** instance, which is assigned to it upon creation by the Notification Service event channel.

3.4.15.2 MyChannel

The **MyChannel** attribute is a readonly attribute that maintains the object reference of the Notification Service event channel, which created a given **ConsumerAdmin** instance.

3.4.15.3 MyOperator

The **MyOperator** attribute is a readonly attribute that maintains the information regarding whether AND or OR semantics will be used during the evaluation of a given event against a set of filter objects, when combining the filter objects associated with the target **ConsumerAdmin** and those defined locally on a given proxy supplier.

3.4.15.4 priority_filter

The **priority_filter** attribute maintains a reference to a mapping filter object that affects the way in which each proxy supplier object created by the target **ConsumerAdmin** instance treats each event it receives with respect to priority.

Note that each proxy supplier object also has an associated attribute that maintains a reference to a mapping filter object for the priority property. If this attribute is set to the reference of a valid mapping filter object, this mapping filter will override that set at the admin level..Otherwise, the mapping filter object referred to by the **priority_filter** attribute of the **ConsumerAdmin** is used.

3.4.15.5 *lifetime_filter*

The **lifetime_filter** attribute maintains a reference to a mapping filter object that affects the way in which each proxy supplier object created by the target **ConsumerAdmin** instance treats each event it receives with respect to lifetime.

Note that each proxy supplier object also has an associated attribute that maintains a reference to a mapping filter object for the lifetime property. If this attribute is set to the reference of a valid mapping filter object, this mapping filter will override that set at the admin level. Otherwise, the mapping filter object referred to by the **lifetime_filter** attribute of the **ConsumerAdmin** is used.

3.4.15.6 *pull_suppliers*

The **pull_suppliers** attribute is a readonly attribute that contains the list of unique identifiers which have been assigned by a **ConsumerAdmin** instance to each pull-style proxy supplier object it has created.

3.4.15.7 *push_suppliers*

The **push_suppliers** attribute is a readonly attribute that contains the list of unique identifiers which have been assigned by a **ConsumerAdmin** instance to each push-style proxy supplier object it has created.

3.4.15.8 *get_proxy_supplier*

The **get_proxy_supplier** operation accepts as an input parameter the numeric unique identifier associated with one of the proxy supplier objects that has been created by the target **ConsumerAdmin** instance. If the input parameter does correspond to the unique identifier of a proxy supplier object that has been created by the target **ConsumerAdmin** instance, that proxy supplier object's reference is returned as the result of the operation. Otherwise, the ProxyNotFound exception is raised.

3.4.15.9 *obtain_notification_pull_supplier*

The **obtain_notification_pull_supplier** operation can create instances of the various types of pull-style proxy supplier objects defined within the **CosNotifyChannelAdmin** module. Recall that three varieties of pull-style proxy supplier objects are defined within this module:

- instances of the **ProxyPullSupplier** interface support connections to pull consumers that receive events as Anys,

- instances of the **StructuredProxyPullSupplier** interface support connections to pull consumers that receive events as Structured Events, and
- instances of the **SequenceProxyPullSupplier** interface support connections to pull consumers that receive events as sequences of Structured Events.

The **obtain_notification_pull_supplier** operation thus accepts as an input parameter a flag that indicates which style of pull-style proxy supplier instance should be created. If the number of consumers currently connected to the channel with which the target **ConsumerAdmin** object is associated exceeds the value of the **MaxConsumers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **ConsumerAdmin** creates the new pull-style proxy supplier instance and assigns a numeric identifier to it that is unique among all proxy suppliers it has created. The unique identifier is returned as the output parameter of the operation, and the reference to the new proxy supplier instance is returned as the operation result.

3.4.15.10 obtain_notification_push_supplier

The **obtain_notification_push_supplier** operation can create instances of the various types of push-style proxy supplier objects defined within the **CosNotifyChannelAdmin** module. Recall that three varieties of push-style proxy supplier objects are defined within this module:

- instances of the **ProxyPushSupplier** interface support connections to push consumers which receive events as Anys,
- instances of the **StructuredProxyPushSupplier** interface support connections to push consumers which receive events as Structured Events, and
- instances of the **SequenceProxyPushSupplier** interface support connections to push consumers which receive events as sequences of Structured Events.

The **obtain_notification_push_supplier** operation thus accepts as an input parameter a flag that indicates which style of push-style proxy supplier instance should be created. If the number of consumers currently connected to the channel with which the target **ConsumerAdmin** object is associated exceeds the value of the **MaxConsumers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **ConsumerAdmin** creates the new push-style proxy supplier instance and assigns a numeric identifier to it that is unique among all proxy suppliers it has created. The unique identifier is returned as the output parameter of the operation, and the reference to the new proxy supplier instance is returned as the operation result.

3.4.15.11 destroy

The **destroy** operation can be invoked to destroy the target **ConsumerAdmin** instance, freeing all resources consumed by the instance. Note that **destroy** can be invoked on a **ConsumerAdmin** instance that is current managing proxy supplier objects that support open connections to consumers. In this case, the effect of invoking **destroy** on the **ConsumerAdmin** is that the operation will disconnect each of the

proxy supplier objects being managed by the target **ConsumerAdmin** from their consumers, and destroy each of these proxy suppliers. Ultimately, the **ConsumerAdmin** instance itself will be destroyed.

3.4.16 The SupplierAdmin Interface

The **SupplierAdmin** interface defines the behavior supported by objects which create and manage lists of proxy consumer objects within a Notification Service event channel. Recall that a Notification Service event channel can have any number of **SupplierAdmin** instances associated with it. Each such instance is responsible for creating and managing a list of proxy consumer objects that share a common set of QoS property settings, and a common set of filter objects. This feature enables clients to conveniently group proxy consumer objects within a channel into groupings that each support a set of suppliers with a common set of QoS requirements, and that make common event forwarding decisions driven by the association of a common set of filter objects.

The **SupplierAdmin** interface inherits the **QoSAdmin** interface defined within the **CosNotification** module, enabling each **SupplierAdmin** instance to manage a set of QoS property settings. These QoS property settings are assigned as the default QoS property settings for any proxy consumer object created by a **SupplierAdmin** instance. In addition, the **SupplierAdmin** interface inherits from the **FilterAdmin** interface defined within the **CosNotifyFilter** module, enabling each **SupplierAdmin** instance to maintain a list of filter objects. These filter objects encapsulate subscriptions that will apply to all proxy consumer objects that have been created by a given **SupplierAdmin** instance. In order to enable optimizing the notification of a group of proxy consumer objects that have been created by the same **SupplierAdmin** instance of changes to the types of events being offered by suppliers, the **SupplierAdmin** interface also inherits from the **NotifyPublish** interface defined in the **CosNotifyComm** module. This inheritance enables a **SupplierAdmin** instance to be the target of an **offer_change** request made by a supplier object, and for the change in event types being offered to be shared by all proxy consumer objects which were created by the target **SupplierAdmin**.

The **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module also inherits from the **SupplierAdmin** interface defined in the **CosEventChannelAdmin** module. This inheritance enables clients to use the **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module to create pure OMG Event Service style proxy consumer objects. Proxy consumer objects created in this manner may not support configuration of QoS properties, and may not have associated filter objects. In addition, proxy consumer objects created through the inherited **SupplierAdmin** interface will not have unique identifiers associated with them, whereas proxy consumer objects created by invoking the operations supported by the **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module will.

Locally, the **SupplierAdmin** interface supports a readonly attribute that maintains a reference to the **EventChannel** instance that created a given **SupplierAdmin** instance. The **SupplierAdmin** interface also supports a readonly attribute, which contains a numeric identifier which will be assigned to an instance supporting this

interface by its associated Notification Service event channel upon creation of the **SupplierAdmin** instance. This identifier will be unique among all **SupplierAdmin** instances created by a given channel.

As described above, due to inheritance from the **FilterAdmin** interface, a **SupplierAdmin** can maintain a list of filter objects that will be applied to all proxy consumers it creates. Also recall that each proxy consumer may itself support a list of filter objects that apply only to it. When combining multiple filter objects within each of these two lists of filter objects that may be associated with a given proxy consumer, OR semantics are applied. However when combining these two lists during the evaluation of a given event, either AND or OR semantics may be applied. The choice is determined by an input flag upon creation of the **SupplierAdmin**, and the operator that will be used for this purpose by a given **SupplierAdmin** is maintained in a readonly attribute.

Each **SupplierAdmin** instance assigns a unique numeric identifier to each proxy consumer object it maintains. The **SupplierAdmin** interface supports attributes that maintain the list of these unique identifiers associated with the proxy pull and the proxy push consumers created by a given **SupplierAdmin** instance. The **SupplierAdmin** interface also supports an operation which, given the unique identifier of a proxy consumer a given **SupplierAdmin** instance has created as input, will return the object reference of that proxy consumer object. Additionally, the **SupplierAdmin** interface supports operations that can create the various styles of proxy consumer objects supported by the Notification Service event channel. Finally, because clients of a given Notification Service event channel can create any number of **SupplierAdmin** instances, a **destroy** operation is provided by this interface so that clients can clean up instances that are no longer needed.

3.4.16.1 MyID

The **MyID** attribute is a readonly attribute that maintains the unique identifier of the target **SupplierAdmin** instance, which is assigned to it upon creation by the Notification Service event channel.

3.4.16.2 MyChannel

The **MyChannel** attribute is a readonly attribute that maintains the object reference of the Notification Service event channel, which created a given **SupplierAdmin** instance.

3.4.16.3 MyOperator

The **MyOperator** attribute is a readonly attribute that maintains the information regarding whether AND or OR semantics will be used during the evaluation of a given event against a set of filter objects, when combining the filter objects associated with the target **SupplierAdmin** and those defined locally on a given proxy consumer.

3.4.16.4 *pull_consumers*

The **pull_consumers** attribute is a readonly attribute that contains the list of unique identifiers, which have been assigned by a **SupplierAdmin** instance to each pull-style proxy consumer object it has created.

3.4.16.5 *push_consumers*

The **push_consumers** attribute is a readonly attribute that contains the list of unique identifiers, which have been assigned by a **SupplierAdmin** instance to each push-style proxy consumer object it has created.

3.4.16.6 *get_proxy_consumer*

The **get_proxy_consumer** operation accepts as an input parameter the numeric unique identifier associated with one of the proxy consumer objects that has been created by the target **SupplierAdmin** instance. If the input parameter does correspond to the unique identifier of a proxy consumer object that has been created by the target **SupplierAdmin** instance, that proxy consumer object's reference is returned as the result of the operation. Otherwise, the ProxyNotFound exception is raised.

3.4.16.7 *obtain_notification_pull_consumer*

The **obtain_notification_pull_consumer** operation can create instances of the various types of pull-style proxy consumer objects defined within the **CosNotifyChannelAdmin** module. Recall that three varieties of pull-style proxy consumer objects are defined within this module:

- instances of the **ProxyPullConsumer** interface support connections to pull suppliers which send events as Anys,
- instances of the **StructuredProxyPullConsumer** interface support connections to pull suppliers which send events as Structured Events, and
- instances of the **SequenceProxyPullConsumer** interface support connections to pull suppliers which send events as sequences of Structured Events.

The **obtain_notification_pull_consumer** operation thus accepts as an input parameter a flag that indicates which style of pull-style proxy consumer instance should be created. If the number of suppliers currently connected to the channel with which the target **SupplierAdmin** object is associated exceeds the value of the **MaxSuppliers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **SupplierAdmin** creates the new pull-style proxy consumer instance and assigns a numeric identifier to it that is unique among all proxy consumers it has created. The unique identifier is returned as the output parameter of the operation, and the reference to the new proxy consumer instance is returned as the operation result.

3.4.16.8 *obtain_notification_push_consumer*

The **obtain_notification_push_consumer** operation can create instances of the various types of push-style proxy consumer objects defined within the **CosNotifyChannelAdmin** module. Recall that three varieties of push-style proxy consumer objects are defined within this module:

- instances of the **ProxyPushConsumer** interface support connections to push suppliers which send events as Anys,
- instances of the **StructuredProxyPushConsumer** interface support connections to push suppliers which send events as Structured Events, and
- instances of the **SequenceProxyPushConsumer** interface support connections to push suppliers which send events as sequences of Structured Events.

The **obtain_notification_push_consumer** operation thus accepts as an input parameter a flag which indicates which style of push-style proxy consumer instance should be created. If the number of suppliers currently connected to the channel with which the target **SupplierAdmin** object is associated exceeds the value of the **MaxSuppliers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **SupplierAdmin** creates the new push-style proxy consumer instance and assigns a numeric identifier to it that is unique among all proxy consumers it has created. The unique identifier is returned as the output parameter of the operation, and the reference to the new proxy consumer instance is returned as the operation result.

3.4.16.9 *destroy*

The **destroy** operation can be invoked to destroy the target **SupplierAdmin** instance, freeing all resources consumed by the instance. Note that **destroy** can be invoked on a **SupplierAdmin** instance that is current managing proxy consumer objects that support open connections to suppliers. In this case, the effect of invoking **destroy** on the **SupplierAdmin** is that the operation will disconnect each of the proxy consumer objects being managed by the target **SupplierAdmin** from their suppliers, and destroy each of these proxy consumers. Ultimately, the **SupplierAdmin** instance itself will be destroyed.

3.4.17 *The EventChannel Interface*

The **EventChannel** interface encapsulates the behaviors supported by a Notification Service event channel. This interface inherits from the **EventChannel** interface defined within the **CosEventChannelAdmin** module of the OMG Event Service, making an instance of the Notification Service **EventChannel** interface fully backward compatible with an OMG Event Service style untyped event channel.

Inheritance of the **EventChannel** interface defined within the **CosEventChannelAdmin** module enables an instance of the **EventChannel** interface defined within the **CosNotifyChannelAdmin** module to create event service style **ConsumerAdmin** and **SupplierAdmin** instances. These instances can subsequently be used to create pure event service style proxy interfaces, which support

connections to pure event service style suppliers and consumers. Note that while Notification Service style proxies and admin objects have unique identifiers associated with them, enabling their references to be obtained by invoking operations on the Notification Service style admin and event channel interfaces, Event Service style proxies and admin objects do not have associated unique identifiers, and thus cannot be returned by invoking an operation on the Notification Service style admin or event channel interfaces.

The **EventChannel** interface defined within the **CosNotifyChannelAdmin** module also inherits from the **QoSAdmin** and the **AdminPropertiesAdmin** interfaces defined within the **CosNotification** module. Inheritance of these interfaces enables a Notification Service style event channel to manage lists of associated QoS and administrative properties, respectively.

Locally, the **EventChannel** interface supports a readonly attribute that maintains a reference to the **EventChannelFactory** instance that created it. In addition, each instance of the **EventChannel** interface has an associated default **ConsumerAdmin** and an associated default **SupplierAdmin** instance, both of which exist upon creation of the channel and which have the unique identifier of zero (note that admin object identifiers only need to be unique among a given type of admin, implying that the identifiers assigned to **ConsumerAdmin** objects can overlap those assigned to **SupplierAdmin** objects). The **EventChannel** interface supports readonly attributes which maintain references to these default admin objects.

The **EventChannel** interface supports operations that create new **ConsumerAdmin** and **SupplierAdmin** instances. In addition, the **EventChannel** interface supports operations which can return references to the **ConsumerAdmin** and **SupplierAdmin** instances associated with a given **EventChannel** instance, given the unique identifier of an admin object as input. Finally, the **EventChannel** interface supports operations, which return the sequence of unique identifiers of all **ConsumerAdmin** and **SupplierAdmin** instances associated with a given **EventChannel** instance.

3.4.17.1 MyFactory

The **MyFactory** attribute is a readonly attribute that maintains the object reference of the event channel factory, which created a given Notification Service **EventChannel** instance.

3.4.17.2 default_consumer_admin

The **default_consumer_admin** attribute is a readonly attribute that maintains a reference to the default **ConsumerAdmin** instance associated with the target **EventChannel** instance. Each **EventChannel** instance has an associated default **ConsumerAdmin** instance, which exists upon creation of the channel and is assigned the unique identifier of zero. Subsequently, clients can create additional Event Service style **ConsumerAdmin** instances by invoking the inherited **for_consumers** operation, and additional Notification Service style **ConsumerAdmin** instances by invoking the **new_for_consumers** operation defined by the **EventChannel** interface.

3.4.17.3 *default_supplier_admin*

The **default_supplier_admin** attribute is a readonly attribute that maintains a reference to the default **SupplierAdmin** instance associated with the target **EventChannel** instance. Each **EventChannel** instance has an associated default **SupplierAdmin** instance, which exists upon creation of the channel and is assigned the unique identifier of zero. Subsequently, clients can create additional Event Service style **SupplierAdmin** instances by invoking the inherited **for_suppliers** operation, and additional Notification Service style **SupplierAdmin** instances by invoking the **new_for_suppliers** operation defined by the **EventChannel** interface.

3.4.17.4 *default_filter_factory*

The **default_filter_factory** attribute is a readonly attribute that maintains an object reference to the default factory to be used by the **EventChannel** instance with which it's associated for creating filter objects. If the target channel does not support a default filter factory, the attribute will maintain the value of OBJECT_NIL.

3.4.17.5 *new_for_consumers*

The **new_for_consumers** operation is invoked to create a new Notification Service style **ConsumerAdmin** instance. The operation accepts as an input parameter a boolean flag, which indicates whether AND or OR semantics will be used when combining the filter objects associated with the newly created **ConsumerAdmin** instance with those associated with a supplier proxy, which was created by the **ConsumerAdmin** during the evaluation of each event against a set of filter objects. The new instance is assigned a unique identifier by the target **EventChannel** instance that is unique among all **ConsumerAdmin** instances currently associated with the channel. Upon completion, the operation returns the reference to the new **ConsumerAdmin** instance as the result of the operation, and the unique identifier assigned to the new **ConsumerAdmin** instance as the output parameter.

3.4.17.6 *new_for_suppliers*

The **new_for_suppliers** operation is invoked to create a new Notification Service style **SupplierAdmin** instance. The operation accepts as an input parameter a boolean flag, which indicates whether AND or OR semantics will be used when combining the filter objects associated with the newly created **SupplierAdmin** instance with those associated with a consumer proxy, which was created by the **SupplierAdmin** during the evaluation of each event against a set of filter objects. The new instance is assigned a unique identifier by the target **EventChannel** instance that is unique among all **SupplierAdmin** instances currently associated with the channel. Upon completion, the operation returns the reference to the new **SupplierAdmin** instance as the result of the operation, and the unique identifier assigned to the new **SupplierAdmin** instance as the output parameter.

3.4.17.7 *get_consumeradmin*

The **get_consumeradmin** operation returns a reference to one of the **ConsumerAdmin** instances associated with the target **EventChannel** instance. The operation accepts as an input parameter a numeric value which is intended to be the unique identifier of one of the **ConsumerAdmin** instances associated with the target **EventChannel** instance. If this turns out to be the case, the object reference of the associated **ConsumerAdmin** instance is returned as the operation result. Otherwise, the AdminNotFound exception is raised.

Note that while a Notification Service style event channel can support both Event Service and Notification Service style **ConsumerAdmin** instances, only Notification Service style **ConsumerAdmin** instances have associated unique identifiers.

3.4.17.8 *get_supplieradmin*

The **get_supplieradmin** operation returns a reference to one of the **SupplierAdmin** instances associated with the target **EventChannel** instance. The operation accepts as an input parameter a numeric value which is intended to be the unique identifier of one of the **SupplierAdmin** instances associated with the target **EventChannel** instance. If this turns out to be the case, the object reference of the associated **SupplierAdmin** instance is returned as the operation result. Otherwise, the AdminNotFound exception is raised.

Note that while a Notification Service style event channel can support both Event Service and Notification Service style **SupplierAdmin** instances, only Notification Service style **SupplierAdmin** instances have associated unique identifiers.

3.4.17.9 *get_all_consumeradmins*

The **get_all_consumeradmins** operation takes no input parameters and returns a sequence of the unique identifiers assigned to all Notification Service style **ConsumerAdmin** instances, which have been created by the target **EventChannel** instance.

3.4.17.10 *get_all_supplieradmins*

The **get_all_supplieradmins** operation takes no input parameters and returns a sequence of the unique identifiers assigned to all Notification Service style **SupplierAdmin** instances which have been created by the target **EventChannel** instance.

3.4.18 *The EventChannelFactory Interface*

The **EventChannelFactory** interface defines operations for creating and managing new Notification Service style event channels. It supports a routine that creates new instances of Notification Service event channels and assigns unique numeric identifiers to them. In addition, the **EventChannelFactory** interface supports a routine, which can return the unique identifiers assigned to all event channels created by a given

instance of **EventChannelFactory**, and another routine which, given the unique identifier of an event channel created by a target **EventChannelFactory** instance, returns the object reference of that event channel.

3.4.18.1 create_channel

The **create_channel** operation is invoked to create a new instance of the Notification Service style event channel. This operation accepts two input parameters. The first input parameter is a list of name-value pairs, which specify the initial QoS property settings for the new channel. The second input parameter is a list of name-value pairs, which specify the initial administrative property settings for the new channel.

If no implementation of the **EventChannel** interface exists that can support all of the requested QoS property settings, the UnsupportedQoS exception is raised. This exception contains as data a sequence of data structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property that could be satisfied. The meanings of the error codes that might be returned are described in Table 2-5 on page 2-46.

Likewise, if no implementation of the **EventChannel** interface exists that can support all of the requested administrative property settings, the UnsupportedAdmin exception is raised. This exception contains as data a sequence of data structures, each of which identifies the name of an administrative property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes that might be returned are described in Table 2-5 on page 2-46.

If neither of these exceptions is raised, the **create_channel** operation will return a reference to a new Notification Service style event channel. In addition, the operation assigns to this new event channel a numeric identifier, which is unique among all event channels created by the target object. This numeric identifier is returned as an output parameter.

3.4.18.2 get_all_channels

The **get_all_channels** operation returns a sequence of all of the unique numeric identifiers corresponding to Notification Service event channels, which have been created by the target object.

3.4.18.3 get_event_channel

The **get_event_channel** operation accepts as input a numeric value that is supposed to be the unique identifier of a Notification Service event channel, which has been created by the target object. If this input value does not correspond to such a unique identifier, the ChannelNotFound exception is raised. Otherwise, the operation returns the object reference of the Notification Service event channel corresponding to the input identifier.

3.5 The CosTypedNotifyComm Module

The **CosTypedNotifyComm** module defines the client interfaces required for doing typed event style communication. Note that typed client interfaces are only required for push-style consumers and pull-style suppliers, since these are the interfaces that need to support event type specific transmission operations. Since no new operations are required to perform typed event communication for pull-style consumers and push-style suppliers, the analogous client interfaces defined in **CosNotifyComm** can be reused for those types of typed clients.

Note that in addition to requiring special interfaces for typed style push consumers, and typed style pull suppliers, it is necessary that these clients support the **NotifyPublish** and **NotifySubscribe** interfaces, respectively, in order to support the offer and type information sharing mechanism provided by the Notification Service. This is the reason that a special module must be defined for these types of notification service clients, as opposed to reusing those defined in the **CosTypedEventComm** module of the OMG event service.

```
module CosTypedNotifyComm {

    interface TypedPushConsumer :
        CosTypedEventComm::TypedPushConsumer,
        CosNotifyComm::NotifyPublish {
    }; // TypedPushConsumer


    interface TypedPullSupplier :
        CosTypedEventComm::TypedPullSupplier,
        CosNotifyComm::NotifySubscribe {
    }; // TypedPullSupplier


}; // CosTypedNotifyComm
```

3.5.1 The TypedPushConsumer Interface

The **TypedPushConsumer** interface supports the behavior required by typed event style push consumers connected to a Notification Service typed channel. This interface inherits from the **TypedPushConsumer** interface defined by the **CosTypedEventComm** module of the OMG Event Service. This inherited interface supports the **get_typed_consumer** operation that enables an instance supporting the **TypedPushConsumer** interface to return a reference to a type-specific interface that supports type-specific event transmission.

In addition, the **TypedPushConsumer** interface inherits the **NotifyPublish** interface defined by the **CosNotifyComm** module. This inheritence enables an instance supporting the **TypedPushConsumer** interface to have its **offer_change** operation invoked, keeping it informed of the types of events being offered by suppliers connected to the same channel.

3.5.2 The TypedPullSupplier Interface

The **TypedPullSupplier** interface supports the behavior required by typed event style pull suppliers connected to a Notification Service typed channel. This interface inherits from the **TypedPullSupplier** interface defined by the **CosTypedEventComm** module of the OMG Event Service. This inherited interface supports the **get_typed_supplier** operation that enables an instance supporting the **TypedPullSupplier** interface to return a reference to a type-specific interface that supports type-specific event transmission.

In addition, the **TypedPullSupplier** interface inherits the **NotifySubscribe** interface defined by the **CosNotifyComm** module. This inheritence enables an instance supporting the **TypedPullSupplier** interface to have its **subscription_change** operation invoked, keeping it informed of the types of events being subscribed to by consumers connected to the same channel.

3.6 CosTypedNotifyChannelAdmin

The **CosTypedNotifyChannelAdmin** module defines the interfaces necessary to create, configure, and administer instances of a Notification Service typed event channel. The Notification Service typed event channel is essentially a hybrid of the typed event channel defined by the OMG Event Service, and the Notification Service event channel described in the previous section. The Notification Service typed event channel supports typed event service clients, exactly as defined in the OMG Event Service, but provides the advantages of QoS administration of the channel, admin, and proxy interfaces, and also enables filtering to be performed on typed events.

Through inheritance of analogous interfaces defined in the **CosTypedEventChannelAdmin** module of the OMG Event Service, a Notification Service typed event channel supports backward compatibility with an Event Service typed event channel. In addition, the **CosTypedNotifyChannelAdmin** module defines new versions of the **TypedEventChannel**, admin, and proxy interfaces that support connections from clients who will communicate via typed events, but also desire the ability to configure their connections to the channel to support various QoS properties, and the ability to define filters based on the type and contents of typed events.

The concepts involved with filter of typed events are described in Section 2.7, “Filtering Typed Events,” on page 2-52. The interfaces and modules that comprise the **CosTypedNotifyChannelAdmin** module are specified below.

```
module CosTypedNotifyChannelAdmin {

    // Forward declaration
    interface TypedEventChannelFactory;


    typedef string Key;


    interface TypedProxyPushConsumer :
```

```
        CosNotifyChannelAdmin::ProxyConsumer,
        CosTypedNotifyComm::TypedPushConsumer {

    void connect_typed_push_supplier (
            in CosEventComm::PushSupplier push_supplier )
        raises ( CosEventChannelAdmin::AlreadyConnected );

}; // TypedProxyPushConsumer

interface TypedProxyPullSupplier :
        CosNotifyChannelAdmin::ProxySupplier,
        CosTypedNotifyComm::TypedPullSupplier {

    void connect_typed_pull_consumer (
            in CosEventComm::PullConsumer pull_consumer )
        raises ( CosEventChannelAdmin::AlreadyConnected );

}; // TypedProxyPullSupplier

interface TypedProxyPullConsumer :
        CosNotifyChannelAdmin::ProxyConsumer,
        CosNotifyComm::PullConsumer {

    void connect_typed_pull_supplier (
            in CosTypedEventComm::TypedPullSupplier pull_supplier)
        raises ( CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises (CosNotifyChannelAdmin::ConnectionAlreadyInactive,
                CosNotifyChannelAdmin::NotConnected);

    void resume_connection()
        raises (CosNotifyChannelAdmin::ConnectionAlreadyActive,
                CosNotifyChannelAdmin::NotConnected);

}; // TypedProxyPullConsumer

interface TypedProxyPushSupplier :
        CosNotifyChannelAdmin::ProxySupplier,
        CosNotifyComm::PushSupplier {

    void connect_typed_push_consumer (
```

```
                in CosTypedEventComm::TypedPushConsumer push_consumer)
            raises ( CosEventChannelAdmin::AlreadyConnected,
                    CosEventChannelAdmin::TypeError );

        void suspend_connection()
            raises (CosNotifyChannelAdmin::ConnectionAlreadyInactive,
                    CosNotifyChannelAdmin::NotConnected);

        void resume_connection()
            raises (CosNotifyChannelAdmin::ConnectionAlreadyActive,
                    CosNotifyChannelAdmin::NotConnected);

    }; // TypedProxyPushSupplier

    interface TypedConsumerAdmin :
            CosNotifyChannelAdmin::ConsumerAdmin,
            CosTypedEventChannelAdmin::TypedConsumerAdmin {

     TypedProxyPullSupplier obtain_typed_notification_pull_supplier(
                in Key supported_interface,
                out CosNotifyChannelAdmin::ProxyID proxy_id )
            raises( CosTypedEventChannelAdmin::InterfaceNotSupported,
                    CosNotifyChannelAdmin::AdminLimitExceeded );

     TypedProxyPushSupplier obtain_typed_notification_push_supplier(
                in Key uses_interface,
                out CosNotifyChannelAdmin::ProxyID proxy_id )
            raises( CosTypedEventChannelAdmin::NoSuchImplementation,
                CosNotifyChannelAdmin::AdminLimitExceeded );

    }; // TypedConsumerAdmin

    interface TypedSupplierAdmin :
        CosNotifyChannelAdmin::SupplierAdmin,
        CosTypedEventChannelAdmin::TypedSupplierAdmin {

     TypedProxyPushConsumer obtain_typed_notification_push_consumer(
                in Key supported_interface,
                out CosNotifyChannelAdmin::ProxyID proxy_id )
            raises( CosTypedEventChannelAdmin::InterfaceNotSupported,
                    CosNotifyChannelAdmin::AdminLimitExceeded );

     TypedProxyPullConsumer obtain_typed_notification_pull_consumer(
                in Key uses_interface,
                out CosNotifyChannelAdmin::ProxyID proxy_id )
```

```
                raises( CosTypedEventChannelAdmin::NoSuchImplementation,
                        CosNotifyChannelAdmin::AdminLimitExceeded );

        }; // TypedSupplierAdmin


        interface TypedEventChannel :
                CosNotification::QoSAdmin,
                CosNotification::AdminPropertiesAdmin,
                CosTypedEventChannelAdmin::TypedEventChannel {

            readonly attribute TypedEventChannelFactory MyFactory;

            readonly attribute TypedConsumerAdmin default_consumer_admin;
            readonly attribute TypedSupplierAdmin default_supplier_admin;

            readonly attribute CosNotifyFilter::FilterFactory

                                        default_filter_factory;

            TypedConsumerAdmin new_for_typed_notification_consumers(
                    in CosNotifyChannelAdmin::InterFilterGroupOperator op,
                    out CosNotifyChannelAdmin::AdminID id );

            TypedSupplierAdmin new_for_typed_notification_suppliers(
                    in CosNotifyChannelAdmin::InterFilterGroupOperator op,
                    out CosNotifyChannelAdmin::AdminID id );

            TypedConsumerAdmin get_consumeradmin (
                    in CosNotifyChannelAdmin::AdminID id )
                raises ( CosNotifyChannelAdmin::AdminNotFound );

            TypedSupplierAdmin get_supplieradmin (
                    in CosNotifyChannelAdmin::AdminID id )
                raises ( CosNotifyChannelAdmin::AdminNotFound );

            CosNotifyChannelAdmin::AdminIDSeq get_all_consumeradmins();
            CosNotifyChannelAdmin::AdminIDSeq get_all_supplieradmins();

        }; // TypedEventChannel

        interface TypedEventChannelFactory {
```

```
TypedEventChannel create_typed_channel (
        in CosNotification::QoSProperties initial_QoS,
        in CosNotification::AdminProperties initial_admin,
        out CosNotifyChannelAdmin::ChannelID id)
    raises( CosNotification::UnsupportedQoS,
            CosNotification::UnsupportedAdmin );

CosNotifyChannelAdmin::ChannelIDSeq get_all_typed_channels();

TypedEventChannel get_typed_event_channel (
        in CosNotifyChannelAdmin::ChannelID id )
    raises ( CosNotifyChannelAdmin::ChannelNotFound );

}; // TypedEventChannelFactory

}; // CosTypedNotifyChannelAdmin
```

3.6.1 The TypedProxyPushConsumer Interface

The **TypedProxyPushConsumer** interface supports connections to the channel by suppliers who will push OMG Event Service style typed events to the channel.

Through inheritance of the **ProxyConsumer** interface defined in the **CosNotifyChannelAdmin** module, the **TypedProxyPushConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the object reference of the **SupplierAdmin**[1] instance, which created a given **TypedProxyPushConsumer** instance. In addition, this inheritance implies that a **TypedProxyPushConsumer** instance supports an operation which will return the list of event types which consumers connected to the same channel are interested in receiving, and an operation which can return information about the instance's ability to accept a per-event QoS request.

The **TypedProxyPushConsumer** interface also inherits from the **TypedPushConsumer** interface defined within the **CosTypedNotifyComm** module. This interface supports the event type specific operation(s), which the supplier connected to a **TypedProxyPushConsumer** instance will invoke to send events to the channel in the form of typed events. And, since the **TypedPushConsumer** interface inherits from the **PushConsumer** interface defined in the

1. In this case, the reference is really to a TypedSupplierAdmin instance, which is valid since TypedSupplierAdmin inherits from CosNotifyChannelAdmin::SupplierAdmin.

CosEventComm module, an instance supporting the **TypedProxyPushConsumer** interface supports the standard **push** operation with which it can be supplied untyped events, and the operation required to disconnect the **TypedProxyPushConsumer** from its associated supplier. In addition, since the inherited **TypedPushConsumer** interface inherits the **CosNotifyComm::NotifyPublish** interface, a supplier connected to an instance supporting the **TypedProxyPushConsumer** interface can inform it whenever the list of event types the supplier plans to supply changes.

Finally, the **TypedProxyPushConsumer** interface defines the operation, which can be invoked by a push supplier to establish the connection over which the push supplier will send events to the channel. Note that this can be either a pure event service style, or a notification service style push supplier.

3.6.1.1 connect_typed_push_supplier

The **connect_typed_push_supplier** operation accepts as an input parameter the reference to an object supporting the **PushSupplier** interface defined within the **CosEventComm** module. This reference is that of a supplier that plans to push typed events to the channel with which the target object is associated. This operation is thus invoked in order to establish a connection between a push-style supplier of typed events, and the notification channel. Once established, the supplier can proceed to send events to the channel by invoking the event type specific operation(s) supported by the target **TypedProxyPushConsumer** instance. If the target object of this operation is already connected to a push supplier object, the AlreadyConnected exception will be raised.

Note that since there is no difference between the interfaces of suppliers of untyped and typed events, it would have sufficed to have the **TypedProxyPushConsumer** interface to inherit from the **ProxyPushConsumer** interface defined in the **CosNotifyChannelAdmin** module, and to not define a separate “connect” method for push-style suppliers of typed events. It was felt, however, that explicitly defining this operation makes the usage model of the **TypedProxyPushConsumer** interface more intuitive.

Note also that because the **PushSupplier** interface defined in the **CosNotifyComm** module inherits from the **PushSupplier** interface defined in the **CosEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style push supplier. The only difference between the two are that the latter also supports the **NotifySubscribe** interface, and thus can be the target of **subscription_change** invocations. The implementation of the **TypedProxyPushConsumer** interface should attempt to narrow the input parameter to **CosNotifyComm::PushSupplier** in order to determine which style of push supplier is connecting to it.

3.6.2 The TypedProxyPullSupplier Interface

The **TypedProxyPullSupplier** interface supports connections to the channel by consumers who will pull OMG Event Service style typed events from the channel.

Through inheritance of the **ProxySupplier** interface, the **TypedProxyPullSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, mapping filters for event priority and lifetime, and a readonly attribute containing the object reference of the **ConsumerAdmin**[2] instance, which created a given **TypedProxyPullSupplier** instance. In addition, this inheritance implies that a **TypedProxyPullSupplier** instance supports an operation that will return the list of event types, which the proxy supplier will potentially be supplying, and an operation that can return information about the instance's ability to accept a per-event QoS request.

The **TypedProxyPullSupplier** interface also inherits from the **TypedPullSupplier** interface defined within the **CosTypedNotifyComm** module. This interface supports the event type specific operation(s), which the consumer connected to a **TypedProxyPullSupplier** instance will invoke to receive events from the channel in the form of typed events. And, since the **TypedPullSupplier** interface inherits from the **PullSupplier** interface defined in the **CosEventComm** module, an instance supporting the **TypedProxyPullSupplier** interface supports the standard **pull** and **try_pull** operations with which it can supply untyped events, and the operation required to disconnect the **TypedProxyPullSupplier** from its associated consumer. In addition, since the inherited **TypedPullSupplier** interface inherits the **CosNotifyComm::NotifySubscribe** interface, an instance supporting the **TypedProxyPullSupplier** interface can be informed whenever the list of event types that the consumer connected to it is interested in receiving changes.

Finally, the **TypedProxyPullSupplier** interface defines the operation which can be invoked by a pull consumer to establish the connection over which the pull consumer will receive events from the channel. Note that this can be either a pure event service style, or a notification service style pull consumer.

3.6.2.1 *connect_typed_pull_consumer*

The **connect_typed_pull_consumer** operation accepts as an input parameter the reference to an object supporting the **PullConsumer** interface defined within the **CosEventComm** module. This reference is that of a consumer, which plans to pull typed events from the channel with which the target object is associated. This operation is thus invoked in order to establish a connection between a pull-style consumer of typed events, and the notification channel. Once established, the consumer can proceed to receive events from the channel by invoking the event type specific operation(s) supported by the target **TypedProxyPullSupplier** instance. If the target object of this operation is already connected to a pull consumer object, the AlreadyConnected exception will be raised.

2. In this case, the reference is really to a TypedConsumerAdmin instance, which is valid since TypedConsumerAdmin inherits from CosNotifyChannelAdmin::ConsumerAdmin.

Note that since there is no difference between the interfaces of consumers of untyped and typed events, it would have sufficed to have the **TypedProxyPullSupplier** interface to inherit from the **ProxyPullSupplier** interface defined in the **CosNotifyChannelAdmin** module, and to not define a separate “connect” method for pull-style consumers of typed events. It was felt, however, that explicitly defining this operation makes the usage model of the **TypedProxyPullSupplier** interface more intuitive.

Note also that because the **PullConsumer** interface defined in the **CosNotifyComm** module inherits from the **PullConsumer** interface defined in the **CosEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style pull consumer. The only difference between the two are that the latter also supports the **NotifyPublish** interface, and thus can be the target of **offer_change** invocations. The implementation of the **TypedProxyPullSupplier** interface should attempt to narrow the input parameter to **CosNotifyComm::PullConsumer** in order to determine which style of pull consumer is connecting to it.

3.6.3 The TypedProxyPullConsumer Interface

The **TypedProxyPullConsumer** interface supports connections to the channel by suppliers who will make OMG Event Service style typed events available for pulling to the channel.

Through inheritance of the **ProxyConsumer** interface, the **ProxyPullConsumer** interface supports administration of various QoS properties, administration of a list of associated filter objects, and a readonly attribute containing the object reference of the **SupplierAdmin**[3] instance, which created a given **TypedProxyPullConsumer** instance. In addition, this inheritance implies that a **TypedProxyPullConsumer** instance supports an operation that will return the list of event types, which consumers connected to the same channel are interested in receiving, and an operation that can return information about the instance’s ability to accept a per-event QoS request.

The **TypedProxyPullConsumer** interface also inherits from the **PullConsumer** interface defined within the **CosNotifyComm** module. This interface supports the operation required to disconnect the **TypedProxyPullConsumer** from its associated supplier. In addition, since the inherited **PullConsumer** interface inherits the **CosNotifyComm::NotifyPublish** interface, a supplier connected to an instance supporting the **TypedProxyPullConsumer** interface can inform it whenever the list of event types the supplier plans to supply changes.

3. In this case, the reference is really to a TypedSupplierAdmin instance, which is valid since TypedSupplierAdmin inherits from CosNotifyChannelAdmin::SupplierAdmin.

Finally, the **TypedProxyPullConsumer** interface defines the operation that can be invoked by a typed pull supplier to establish the connection over which the typed pull supplier will send events to the channel. Note that this can be either a pure event service style, or a notification service style typed pull supplier. The **TypedProxyPullConsumer** interface also defines a pair of operations, which can suspend and resume the connection between a **TypedProxyPullConsumer** instance and its associated **TypedPullSupplier**. During the time such a connection is suspended, the **TypedProxyPullConsumer** will not attempt to pull events from its associated **TypedPullSupplier**.

3.6.3.1 *connect_typed_pull_supplier*

The **connect_typed_pull_supplier** operation accepts as an input parameter the reference to an object supporting the **TypedPullSupplier** interface defined within the **CosTypedEventComm** module. This reference is that of a supplier that plans to make OMG Event Service style typed events available for pulling to the channel with which the target object is associated. This operation is thus invoked in order to establish a connection between a pull-style supplier of typed events, and the notification channel. Once established, the channel can proceed to receive events from the supplier by invoking the event type specific operation(s) supported by the supplier. If the target object of this operation is already connected to a typed pull supplier object, the AlreadyConnected exception will be raised. An implementation of the **TypedProxyPullConsumer** interface may impose additional requirements on the interface supported by a typed pull supplier (e.g., it may be designed to invoke some specific pull-style operation to receive events). If the typed pull supplier being connected does not meet those requirements, this operation raises the TypeError exception.

Note that because the **TypedPullSupplier** interface defined in the **CosTypedNotifyComm** module inherits from the **TypedPullSupplier** interface defined in the **CosTypedEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style typed pull supplier. The only difference between the two are that the latter also supports the **NotifySubscribe** interface, and thus can be the target of **subscription_change** invocations. The implementation of the **TypedProxyPullConsumer** interface should attempt to narrow the input parameter to **CosTypedNotifyComm::TypedPullSupplier** in order to determine which style of typed pull supplier is connecting to it.

3.6.3.2 *suspend_connection*

The **suspend_connection** operation causes the target object supporting the **TypedProxyPullConsumer** interface to stop attempting to pull events (using **pull** or **try_pull**) from the **TypedPullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception is raised. If no **TypedPullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the

TypedProxyPullConsumer will not attempt to pull events from the **TypedPullSupplier** connected to it until **resume_connection** is subsequently invoked.

3.6.3.3 *resume_connection*

The **resume_connection** operation causes the target object supporting the **TypedProxyPullConsumer** interface to resume attempting to pull events (using **pull** or **try_pull**) from the **TypedPullSupplier** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception is raised. If no **TypedPullSupplier** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **TypedProxyPullConsumer** will resume attempting to pull events from the **TypePullSupplier** connected to it.

3.6.4 *The TypedProxyPushSupplier Interface*

The **TypedProxyPushSupplier** interface supports connections to the channel by consumers who will receive OMG Event Service style events from the channel.

Through inheritance of the **ProxySupplier** interface, the **TypedProxyPushSupplier** interface supports administration of various QoS properties, administration of a list of associated filter objects, mapping filters for event priority and lifetime, and a readonly attribute containing the object reference of the **ConsumerAdmin**[4] instance, which created a given **TypedProxyPushSupplier** instance. In addition, this inheritance implies that a **TypedProxyPushSupplier** instance supports an operation, which will return the list of event types that the proxy supplier will potentially by supplying, and an operation that can return information about the instance's ability to accept a per-event QoS request.

The **TypedProxyPushSupplier** interface also inherits from the **PushSupplier** interface defined within the **CosNotifyComm** module. This interface supports the operation required to disconnect the **TypedProxyPushSupplier** from its associated consumer. In addition, since the inherited **PushSupplier** interface inherits the **CosNotifyComm::NotifySubscribe** interface, an instance supporting the **TypedProxyPushSupplier** interface can be informed whenever the list of event types that the consumer connected to it is interested in receiving changes.

Lastly, the **TypedProxyPushSupplier** interface defines the operation, which can be invoked by a typed push consumer to establish the connection over which the typed push consumer will receive events from the channel. Note that this can be either a pure

4. In this case, the reference is really to a TypedConsumerAdmin instance, which is valid since TypedConsumerAdmin inherits from CosNotifyChannelAdmin::ConsumerAdmin.

event service style, or a notification service style typed push consumer. The **TypedProxyPushSupplier** interface also defines a pair of operations that can suspend and resume the connection between a **TypedProxyPushSupplier** instance and its associated **TypedPushConsumer**. During the time such a connection is suspended, the **TypedProxyPushSupplier** will accumulate events destined for the consumer but not transmit them until the connection is resumed.

3.6.4.1 *connect_typed_push_consumer*

The **connect_typed_push_consumer** operation accepts as an input parameter the reference to an object supporting the **TypedPushConsumer** interface defined within the **CosTypedEventComm** module. This reference is that of a consumer, which will receive OMG Event Service style typed events from the channel with which the target object is associated. This operation is thus invoked in order to establish a connection between a push-style consumer of typed events, and the notification channel. Once established, the **TypedProxyPushSupplier** will proceed to send events destined for the consumer to it by invoking its event type specific push-style operation(s). If the target object of this operation is already connected to a typed push consumer object, the AlreadyConnected exception will be raised. An implementation of the **TypedProxyPushSupplier** interface may impose additional requirements on the interface supported by a typed push consumer (e.g., it may be designed to invoke some specific operation in order to transmit events). If the typed push consumer being connected does not meet those requirements, this operation raises the TypeError exception.

Note that because the **TypedPushConsumer** interface defined in the **CosTypedNotifyComm** module inherits from the **TypedPushConsumer** interface defined in the **CosTypedEventComm** module, the input parameter to this operation could be either a pure event service style, or a notification service style typed push consumer. The only difference between the two are that the latter also supports the **NotifyPublish** interface, and thus can be the target of **offer_change** invocations. The implementation of the **TypedProxyPushSupplier** interface should attempt to narrow the input parameter to **CosTypedNotifyComm::TypedPushConsumer** in order to determine which style of typed push consumer is connecting to it.

3.6.4.2 *suspend_connection*

The **suspend_connection** operation causes the target object supporting the **TypedProxyPushSupplier** interface to stop sending events to the **TypedPushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has been previously suspended using this operation and not resumed by invoking **resume_connection** (described below), the ConnectionAlreadyInactive exception defined in the **CosNotifyChannelAdmin** module is raised. If no **TypedPushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **TypedProxyPushSupplier** will not forward events to the **TypedPushConsumer** connected to it until **resume_connection** is subsequently invoked. During this time, the **TypedProxyPushSupplier** will continue

to queue events destined for the **TypedPushConsumer**, although events that time out prior to resumption of the connection will be discarded. Upon resumption of the connection, all queued events will be forwarded to the **TypedPushConsumer**.

3.6.4.3 resume_connection

The **resume_connection** operation causes the target object supporting the **TypedProxyPushSupplier** interface to resume sending events to the **TypedPushConsumer** instance connected to it. This operation takes no input parameters and returns no values. If the connection has not been previously suspended using this operation by invoking **suspend_connection** (described above), the ConnectionAlreadyActive exception defined in the **CosNotifyChannelAdmin** module is raised. If no **TypedPushConsumer** has been connected to the target object when this operation is invoked, the NotConnected exception is raised. Otherwise, the **TypedProxyPushSupplier** will resume forwarding events to the **TypedPushConsumer** connected to it, including those which have been queued during the time the connection was suspended, and have not yet timed out.

3.6.5 The TypedConsumerAdmin Interface

The **TypedConsumerAdmin** interface defines the behavior supported by objects which create and manage lists of proxy supplier objects within a Notification Service typed event channel. Similar to its untyped counterpart, a Notification Service typed event channel can have any number of **TypedConsumerAdmin** instances associated with it. Each such instance is responsible for creating and managing a list of proxy supplier objects that share a common set of QoS property settings, and a common set of filter objects. This feature enables clients to conveniently group proxy supplier objects within a channel into groupings that each support a set of consumers with a common set of QoS requirements and event subscriptions.

Note that the **TypedConsumerAdmin** interface inherits from **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module, and the **TypedConsumerAdmin** interface defined in the **CosTypedEventChannelAdmin** module. These inheritance relationships have several implications for a Notification Service style **TypedConsumerAdmin** instance.

First, inheritance of the **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module implies that in addition to being capable of creating and managing Notification Service style typed proxy supplier objects, a **TypedConsumerAdmin** instance can also create and manage instances supporting any of the proxy supplier interfaces defined in the **CosNotifyChannelAdmin** module. In addition, since the **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module inherits from the **ConsumerAdmin** interface defined in the **CosEventChannelAdmin** module, a **TypedConsumerAdmin** can also create and manage OMG Event service style untyped proxy supplier objects.

Likewise, inheritance of the **TypedConsumerAdmin** interface defined in the **CosTypedEventChannelAdmin** module implies that an instance supporting the **CosTypedNotifyChannelAdmin**'s version of **TypedConsumerAdmin** can create and manage OMG Event Service style typed proxy supplier objects as well.

Thus, instances supporting the **TypedConsumerAdmin** interface defined in the **CosTypedNotifyChannelAdmin** module can potentially create and manage instances supporting any of the proxy supplier interfaces defined in the **CosEventChannelAdmin**, **CosNotifyChannelAdmin**, **CosTypedEventChannelAdmin**, and the **CosTypedNotifyChannelAdmin** (due to locally defined factory operations) modules. The implication of this is that a Notification Service style typed event channel can support OMG Event Service style untyped and typed consumers, along with all variations of consumers defined in the Notification Service as clients.

Note also that the inherited **CosNotifyChannelAdmin::ConsumerAdmin** interface provides an instance supporting the **CosTypedNotifyChannelAdmin::TypedConsumerAdmin** interface with the behaviors necessary to associate unique identifiers with the proxy supplier objects it creates. While the **TypedConsumerAdmin** interface defined here is capable of creating OMG Event Service style untyped and typed proxy supplier objects, only instances of the proxy supplier interfaces defined in the Notification Service can have associated unique identifiers.

Similarly, the inheritance of the **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module provides an instance supporting the **TypedConsumerAdmin** interface defined in the **CosTypedNotifyChannelAdmin** module with the behaviors necessary to maintain associated QoS property settings and filter objects. The relationships between the QoS property settings and filter objects to the proxy supplier objects created by a **TypedConsumerAdmin** instance are identical to those described in Section 3.4.15, "The ConsumerAdmin Interface," on page 3-71. Note again that QoS property settings and filter objects can only be associated with Notification Service style proxy suppliers, both typed and untyped.

Inheritance of the **ConsumerAdmin** interface defined in **CosNotifyChannelAdmin** also implies that **TypedConsumerAdmin** also inherits from the **NotifySubscribe** interface defined in **CosNotifyComm**. This inheritance enables optimizing the notification of a group of proxy supplier objects that have been created by the same **TypedConsumerAdmin** instance of changes to shared filter objects, since this inheritance enables a **TypedConsumerAdmin** instance to be registered as the callback object for notification of subscription changes made upon filter objects. Lastly, inheritance of the **ConsumerAdmin** interface defined in **CosNotifyChannelAdmin** implies that an instance of the **TypedConsumerAdmin** interface supports readonly attributes that maintain the unique identifier of the instance supplied to it by its creating channel, the object reference of the creating channel, and the flag which indicates whether AND or OR semantics will be used when combining the filter objects associated with a **TypedConsumerAdmin** with those defined on specific proxy suppliers created by the **TypedConsumerAdmin**.

Locally, the **TypedConsumerAdmin** interface supports the operations that create new Notification Service style typed proxy supplier instances. Note lastly that due to inheritance of the **ConsumerAdmin** interface defined in the **CosNotifyChannelAdmin** module, an instance supporting the **TypedConsumerAdmin** interface supports a readonly attribute which maintains a unique identifier assigned to the instance by the channel which created it.

3.6.5.1 *obtain_typed_notification_pull_supplier*

The **obtain_typed_notification_pull_supplier** operation is used to create a new Notification Service style proxy supplier instance, which will support a connection to the channel with which the target **TypedConsumerAdmin** instance is associated by a pull-style consumer of typed events. The operation accepts as input a string, which identifies the name of the strongly typed interface that the newly created **TypedProxyPullSupplier** instance should support. The consumer that connects to the new **TypedProxyPullSupplier** would use this strongly typed interface to invoke operations to receive typed events.

If the target **TypedConsumerAdmin** instance cannot locate an occurrence of the **TypedProxyPullSupplier** interface, which also supports the requested strongly typed interface, the InterfaceNotSupported exception defined the **CosTypedEventChannelAdmin** module is raised. If the number of consumers currently connected to the channel with which the target **TypedConsumerAdmin** object is associated exceeds the value of the **MaxConsumers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **TypedConsumerAdmin** instance creates a new instance supporting the **TypedProxyPullSupplier** interface defined in the **CosTypedNotifyChannelAdmin** module. This interface can subsequently be used for a pull-style consumer of typed events to establish a connection to the Notification Service typed event channel. Upon creating the new **TypedProxyPullSupplier** instance, the target **TypedConsumerAdmin** instance associates with it a unique identifier which it returns as an output parameter. This unique identifier could subsequently be used as the input parameter to the **get_proxy_supplier** operation inherited by the target **TypedConsumerAdmin** instance in order to obtain the reference to the newly created **TypedProxyPullSupplier** instance. This reference is returned as the result of the **obtain_typed_notification_pull_supplier** operation.

3.6.5.2 *obtain_typed_notification_push_supplier*

The **obtain_typed_notification_push_supplier** operation is used to create a new Notification Service style proxy supplier instance, which will support a connection to the channel with which the target **TypedConsumerAdmin** instance is associated by a push-style consumer of typed events. The operation accepts as input a string, which identifies the name of a strongly typed interface that the newly created **TypedProxyPushSupplier** instance should use when invoking operations upon its associated **TypedPushConsumer** instance to send it events.

If the target **TypedConsumerAdmin** instance cannot locate an implementation of the **TypedProxyPullSupplier** interface, which will use the requested strongly typed interface to send events to a **TypedPushConsumer**, the NoSuchImplementation exception defined in the **CosTypedEventChannelAdmin** module is raised. If the number of consumers currently connected to the channel with which the target **TypedConsumerAdmin** object is associated exceeds the value of the **MaxConsumers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **TypedConsumerAdmin** instance creates a new instance supporting the **TypedProxyPushSupplier** interface defined in the **CosTypedNotifyChannelAdmin** module. This interface can subsequently be used for a push-style consumer of typed events to establish a connection to the Notification Service typed event channel. Upon creating the new **TypedProxyPushSupplier** instance, the target **TypedConsumerAdmin** instance associates with it a unique identifier, which it returns as an output parameter. This unique identifier could subsequently be used as the input parameter to the **get_proxy_supplier** operation inherited by the target **TypedConsumerAdmin** instance in order to obtain the reference to the newly created **TypedProxyPushSupplier** instance. This reference is returned as the result of the **obtain_typed_notification_push_supplier** operation.

3.6.6 The TypedSupplierAdmin Interface

The **TypedSupplierAdmin** interface defines the behavior supported by objects that create and manage lists of proxy consumer objects within a Notification Service typed event channel. Similar to its untyped counterpart, a Notification Service typed event channel can have any number of **TypedSupplierAdmin** instances associated with it. Each such instance is responsible for creating and managing a list of proxy consumer objects that share a common set of QoS property settings, and a common set of filter objects. This feature enables clients to conveniently group proxy supplier objects within a channel into groupings that each support a set of consumers with a common set of QoS requirements and event subscriptions.

Note that the **TypedSupplierAdmin** interface inherits from **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module, and the **TypedSupplierAdmin** interface defined in the **CosTypedEventChannelAdmin** module. These inheritance relationships have several implications for a Notification Service style **TypedSupplierAdmin** instance.

First, inheritance of the **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module implies that in addition to being capable of creating and managing Notification Service style typed proxy consumer objects, a **TypedSupplierAdmin** instance can also create and manage instances supporting any of the proxy consumer interfaces defined in the **CosNotifyChannelAdmin** module. In addition, since the **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module inherits from the **SupplierAdmin** interface defined in the **CosEventChannelAdmin** module, a **TypedSupplierAdmin** can also create and manage OMG Event service style untyped proxy consumer objects.

Likewise, inheritance of the **TypedSupplierAdmin** interface defined in the **CosTypedEventChannelAdmin** module implies that an instance supporting the **CosTypedNotifyChannelAdmin**'s version of **TypedSupplierAdmin** can create and manage OMG Event Service style typed proxy consumer objects as well.

Thus, instances supporting the **TypedSupplierAdmin** interface defined in the **CosTypedNotifyChannelAdmin** module can potentially create and manage instances supporting any of the proxy consumer interfaces defined in the **CosEventChannelAdmin**, **CosNotifyChannelAdmin**, **CosTypedEventChannelAdmin**, and the **CosTypedNotifyChannelAdmin** (due to locally defined factory operations) modules. The implication of this is that a Notification Service style typed event channel can support OMG Event Service style untyped and typed suppliers, along with all variations of suppliers defined in the Notification Service as clients.

Note also that the inherited **CosNotifyChannelAdmin::SupplierAdmin** interface provides an instance supporting the **CosTypedNotifyChannelAdmin::TypedSupplierAdmin** interface with the behaviors necessary to associate unique identifiers with the proxy consumer objects it creates. While the **TypedSupplierAdmin** interface defined here is capable of creating OMG Event Service style untyped and typed proxy supplier objects, only instances of the proxy supplier interfaces defined in the Notification Service can have associated unique identifiers.

Similarly, the inheritance of the **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module provides an instance supporting the **TypedSupplierAdmin** interface defined in the **CosTypedNotifyChannelAdmin** module with the behaviors necessary to maintain associated QoS property settings and filter objects. The relationships between the QoS property settings and filter objects to the proxy consumer objects created by a **TypedSupplierAdmin** instance are identical to those described in Section 3.4.16, "The SupplierAdmin Interface," on page 3-76. Note again that QoS property settings and filter objects can only be associated with Notification Service style proxy consumers, both typed and untyped.

Inheritance of the **SupplierAdmin** interface defined in **CosNotifyChannelAdmin** also implies that **TypedSupplierAdmin** also inherits from the **NotifyPublish** interface defined in **CosNotifyComm**. This inheritance enables optimizing the notification of a group of proxy consumer objects that have been created by the same **TypedSupplierAdmin** instance of changes to the types of events being offered to them by suppliers, since this inheritance enables a **TypedSupplierAdmin** instance to be the target of an **offer_change** operation. Lastly, inheritance of the **SupplierAdmin** interface defined in **CosNotifyChannelAdmin** implies that an instance of the **TypedSupplierAdmin** interface supports readonly attributes that maintain the unique identifier of the instance supplied to it by its creating channel, the object reference of the creating channel, and the flag which indicates whether AND or OR semantics will be used when combining the filter objects associated with a **TypedSupplierAdmin** with those defined on specific proxy consumers created by the **TypedSupplierAdmin**.

Locally, the **TypedSupplierAdmin** interface supports the operations that create new Notification Service style typed proxy consumer instances. Note lastly that due to inheritance of the **SupplierAdmin** interface defined in the **CosNotifyChannelAdmin** module, an instance supporting the **TypedSupplierAdmin** interface supports a readonly attribute which maintains a unique identifier assigned to the instance by the channel which created it.

3.6.6.1 *obtain_typed_notification_push_consumer*

The **obtain_typed_notification_push_consumer** operation is used to create a new Notification Service style proxy consumer instance, which will support a connection to the channel with which the target **TypedSupplierAdmin** instance is associated by a push-style supplier of typed events. The operation accepts as input a string, which identifies the name of the strongly typed interface that the newly created **TypedProxyPushConsumer** instance should support. The supplier, which connects to the new **TypedProxyPushConsumer** would use this strongly typed interface to invoke operations to send typed events to the channel.

If the target **TypedSupplierAdmin** instance cannot locate an occurrence of the **TypedProxyPushConsumer** interface, which also supports the requested strongly typed interface, the InterfaceNotSupported exception defined the **CosTypedEventChannelAdmin** module is raised. If the number of suppliers currently connected to the channel with which the target **TypedSupplierAdmin** object is associated exceeds the value of the **MaxSuppliers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **TypedSupplierAdmin** instance creates a new instance supporting the **TypedProxyPushConsumer** interface defined in the **CosTypedNotifyChannelAdmin** module. This interface can subsequently be used for a push-style supplier of typed events to establish a connection to the Notification Service typed event channel. Upon creating the new **TypedProxyPushConsumer** instance, the target **TypedSupplierAdmin** instance associates with it a unique identifier, which it returns as an output parameter. This unique identifier could subsequently be used as the input parameter to the **get_proxy_consumer** operation inherited by the target **TypedSupplierAdmin** instance in order to obtain the reference to the newly created **TypedProxyPushConsumer** instance. This reference is returned as the result of the **obtain_typed_notification_push_consumer** operation.

3.6.6.2 *obtain_typed_notification_pull_consumer*

The **obtain_typed_notification_pull_consumer** operation is used to create a new Notification Service style proxy consumer instance, which will support a connection to the channel with which the target **TypedSupplierAdmin** instance is associated by a pull-style supplier of typed events. The operation accepts as input a string, which identifies the name of a strongly typed interface that the newly created **TypedProxyPullConsumer** instance should use when invoking operations upon its associated **TypedPullSupplier** instance to receive events.

If the target **TypedSupplierAdmin** instance cannot locate an implementation of the **TypedProxyPullConsumer** interface, which will use the requested strongly typed interface to receive events from a **TypedPullSupplier**, the NoSuchImplementation exception defined the **CosTypedEventChannelAdmin** module is raised. If the number of suppliers currently connected to the channel with which the target **TypedSupplierAdmin** object is associated exceeds the value of the **MaxSuppliers** administrative property, the AdminLimitExceeded exception is raised. Otherwise, the target **TypedSupplierAdmin** instance creates a new instance supporting the **TypedProxyPullConsumer** interface defined in the **CosTypedNotifyChannelAdmin** module. This interface can subsequently be used for a pull-style supplier of typed events to establish a connection to the Notification Service typed event channel. Upon creating the new **TypedProxyPullConsumer** instance, the target **TypedSupplierAdmin** instance associates with it a unique identifier, which it returns as an output parameter. This unique identifier could subsequently be used as the input parameter to the **get_proxy_consumer** operation inherited by the target **TypedSupplierAdmin** instance in order to obtain the reference to the newly created **TypedProxyPullConsumer** instance. This reference is returned as the result of the **obtain_typed_notification_pull_consumer** operation.

3.6.7 The TypedEventChannel Interface

The **TypedEventChannel** interface defines the behaviors supported by the Notification Service version of a typed event channel. As previously stated, the Notification Service version of a typed event channel is really a hybrid between the Notification Service event channel defined in the **CosNotifyChannelAdmin** module, and the typed event channel defined in the OMG Event Service. This is evidenced by the fact that the **TypedEventChannel** interface defined here supports similar inheritence and defines similar attributes and operations of the former, and directly inherits from the latter[5].

Inheritance of the **TypedEventChannel** of the **CosTypedEventChannel** module essentially implies backward compatibility between the Notification Service style typed event channel, and the OMG Event Service style typed event channel. It means that pure OMG Event Service style typed admin instances, which can create pure OMG Event Service style typed proxy instances, which can support pure OMG Event Service style typed event clients, can be associated with the Notification Service style typed event channel. As usual, none of the pure OMG style objects will support QoS property configuration, associated filter objects, or administration by unique identifiers.

5. In fact the **TypedEventChannel** interface defined here would multiply inherit from the **EventChannel** interface defined in **CosNotifyChannelAdmin**, and the **TypedEventChannel** interface defined in **CosTypedNotifyChannelAdmin**, except that this multiple inheritence would result in the new interface inheriting two different versions of the **for_consumers** and **for_suppliers** operations, which is not allowed in IDL.

Inheritance of the **QoSAdmin** and **AdminPropertiesAdmin** interfaces defined in the **CosNotification** module implies that instances of the **TypedEventChannel** interface can have associated QoS property and administrative property settings. Locally, the **TypedEventChannel** interface defined here supports a readonly attribute which maintains the reference to the factory that created it, and a pair of readonly attributes that maintain references to the default **TypedConsumerAdmin** and **TypedSupplierAdmin** instances that exist upon creation of an instance of the **TypedEventChannel** interface. The **TypedEventChannel** interface also defines a readonly attribute that maintains the reference of the default filter factory used by the channel to create new filter objects.

Additionally, the **TypedEventChannel** interface defines operations to create new instances of the **TypedConsumerAdmin** and the **TypedSupplierAdmin** interfaces defined in the **CosTypedNotifyChannelAdmin** module. These instances also have associated unique identifiers. Similarly to the **EventChannel** interface defined in the **CosNotifyChannelAdmin** module, the **TypedEventChannel** interface defines operations that can return the lists of identifiers associated with the **TypedConsumerAdmin** and **TypedSupplierAdmin** instances it has created, and operations that given the unique identifier to one of its Admin instance, the object reference associated with it.

3.6.7.1 MyFactory

The **MyFactory** attribute is a readonly attribute that maintains the object reference of the event channel factory, which created a given Notification Service **TypedEventChannel** instance.

3.6.7.2 default_consumer_admin

The **default_consumer_admin** attribute is a readonly attribute which maintains a reference to the default **TypedConsumerAdmin** instance associated with the target **TypedEventChannel** instance. Each **TypedEventChannel** instance has an associated default **TypedConsumerAdmin** instance, which exists upon creation of the channel and is assigned the unique identifier of zero. Subsequently, clients can create additional Event Service style **TypedConsumerAdmin** instances by invoking the inherited **for_consumers** operation, and additional Notification Service style **TypedConsumerAdmin** instances by invoking the **new_for_typed_consumers** operation defined by the **TypedEventChannel** interface.

3.6.7.3 default_supplier_admin

The **default_supplier_admin** attribute is a readonly attribute, which maintains a reference to the default **TypedSupplierAdmin** instance associated with the target **TypedEventChannel** instance. Each **TypedEventChannel** instance has an associated default **TypedSupplierAdmin** instance, which exists upon creation of the channel and is assigned the unique identifier of zero. Subsequently, clients can create additional Event Service style **TypedSupplierAdmin** instances by invoking the

inherited **for_suppliers** operation, and additional Notification Service style **TypedSupplierAdmin** instances by invoking the **new_for_typed_suppliers** operation defined by the **TypedEventChannel** interface.

3.6.7.4 *default_filter_factory*

The **default_filter_factory** attribute is a readonly attribute, which maintains an object reference to the default factory to be used by the **TypedEventChannel** instance with which it's associated for creating filter objects. If the target channel does not support a default filter factory, the attribute will maintain the value of OBJECT_NIL.

3.6.7.5 *new_for_typed_notification_consumers*

The **new_for_typed_consumers** operation is invoked to create a new Notification Service style **TypedConsumerAdmin** instance. The operation accepts as an input parameter a boolean flag that indicates whether AND or OR semantics will be used when combining the filter objects associated with the newly created **TypedConsumerAdmin** instance with those associated with a supplier proxy, which was created by the **TypedConsumerAdmin** during the evaluation of each event against a set of filter objects. The new instance is assigned a unique identifier by the target **TypedEventChannel** instance that is unique among all **ConsumerAdmin** and **TypedConsumerAdmin** instances currently associated with the channel. Upon completion, the operation returns the reference to the new **TypedConsumerAdmin** instance as the result of the operation, and the unique identifier assigned to the new **TypedConsumerAdmin** instance as the output parameter.

3.6.7.6 *new_for_typed_notification_suppliers*

The **new_for_typed_suppliers** operation is invoked to create a new Notification Service style **TypedSupplierAdmin** instance. The operation accepts as an input parameter a boolean flag, which indicates whether AND or OR semantics will be used when combining the filter objects associated with the newly created **TypedSupplierAdmin** instance with those associated with a consumer proxy, which was created by the **TypedSupplierAdmin** during the evaluation of each event against a set of filter objects. The new instance is assigned a unique identifier by the target **TypedEventChannel** instance that is unique among all **SupplierAdmin** and **TypedSupplierAdmin** instances currently associated with the channel. Upon completion, the operation returns the reference to the new **TypedSupplierAdmin** instance as the result of the operation, and the unique identifier assigned to the new **TypedSupplierAdmin** instance as the output parameter.

3.6.7.7 *get_consumeradmin*

The **get_consumeradmin** operation returns a reference to one of the **TypedConsumerAdmin** instances associated with the target **TypedEventChannel** instance. The operation accepts as an input parameter a numeric value, which is intended to be the unique identifier of one of the **TypedConsumerAdmin** instances

associated with the target **TypedEventChannel** instance. If this turns out to be the case, the object reference of the associated **TypedConsumerAdmin** instance is returned as the operation result. Otherwise, the AdminNotFound exception is raised.

Note that while a Notification Service style event channel can support both Event Service and Notification Service style **TypedConsumerAdmin** instances, only Notification Service style **TypedConsumerAdmin** instances have associated unique identifiers.

3.6.7.8 get_supplieradmin

The **get_supplieradmin** operation returns a reference to one of the **TypedSupplierAdmin** instances associated with the target **TypedEventChannel** instance. The operation accepts as an input parameter a numeric value, which is intended to be the unique identifier of one of the **TypedSupplierAdmin** instances associated with the target **TypedEventChannel** instance. If this turns out to be the case, the object reference of the associated **TypedSupplierAdmin** instance is returned as the operation result. Otherwise, the AdminNotFound exception is raised.

Note that while a Notification Service style event channel can support both Event Service and Notification Service style **TypedSupplierAdmin** instances, only Notification Service style **TypedSupplierAdmin** instances have associated unique identifiers.

3.6.7.9 get_all_consumeradmins

The **get_all_consumeradmins** operation takes no input parameters and returns a sequence of the unique identifiers assigned to all Notification Service style **TypedConsumerAdmin** instances, which have been created by the target **TypedEventChannel** instance.

3.6.7.10 get_all_supplieradmins

The **get_all_supplieradmins** operation takes no input parameters and returns a sequence of the unique identifiers assigned to all Notification Service style **TypedSupplierAdmin** instances, which have been created by the target **TypedEventChannel** instance.

3.6.8 The TypedEventChannelFactory Interface

The **TypedEventChannelFactory** interface defines an operation for creating new Notification Service style typed event channels. This interface also supports an operation which can return the list of unique numeric identifiers assigned to all channels that have been created by such an instance, and another which, given the unique identifier of a channel that has been created by the target instance, can return the object reference associated with that channel.

3.6.8.1 *create_typed_channel*

The **create_typed_channel** operation is invoked to create a new instance of the Notification Service style typed event channel. This operation accepts two input parameters. The first input parameter is a list of name-value pairs which specify the initial QoS property settings for the new channel. The second input parameter is a list of name-value pairs which specify the initial administrative property settings for the new channel.

If no implementation of the **TypedEventChannel** interface exists that can support all of the requested QoS property settings, the UnsupportedQoS exception defined in the **CosNotifyChannelAdmin** module is raised. This exception contains as data a sequence of data structures, each of which identifies the name of a QoS property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes which might be returned are described in Table 2-5 on page 2-46.

Likewise, if no implementation of the **TypedEventChannel** interface exists that can support all of the requested administrative property settings, the UnsupportedAdmin exception defined in the **CosNotifyChannelAdmin** module is raised. This exception contains as data a sequence of data structures, each of which identifies the name of an administrative property in the input list whose requested setting could not be satisfied, along with an error code and a range of settings for the property which could be satisfied. The meanings of the error codes which might be returned are described in Table 2-5 on page 2-46.

If neither of these exceptions is raised, the **create_typed_channel** operation will return a reference to a new Notification Service style typed event channel. In addition, the operation assigns to this new typed event channel a numeric identifier, which is unique among all event channels created by the target object. This numeric identifier is returned as an output parameter.

3.6.8.2 *get_all_typed_channels*

The **get_all_typed_channels** operation returns a sequence of all of the unique numeric identifiers corresponding to Notification Service typed event channels which have been created by the target object.

3.6.8.3 *get_typed_event_channel*

The **get_typed_event_channel** operation accepts as input a numeric value, which is supposed to be the unique identifier of a Notification Service typed event channel that has been created by the target object. If this input value does not correspond to such a unique identifier, the ChannelNotFound exception is raised. Otherwise, the operation returns the object reference of the Notification Service typed event channel corresponding to the input identifier.

Event Type Repository — A

The Event Type Repository is a specification of a set of interfaces that are used to store type information about events. The interfaces are generated using the mapping from a meta-model of event types to IDL representing this type information, as specified in the Meta-Object Facility (ad/97-08-14 and ad/97-08-015). This appendix begins by providing an explanation of the meta-model of event types and their names. It then presents the model in terms of equivalent UML (Unified Modeling Language) and MODL (Meta Object Definition Language). Finally, the IDL that is generated from these high-level notations is given in full.

A.1 Event Type Meta-Model

The Event Type Repository is designed to store information about the kinds of filterable data that events with certain names will provide to consumers. It is a queriable store that can be used by event suppliers to determine the names and types of the properties that an event of a certain type must contain to be conformant to that type. It can also be used by consumers to discover the properties that they can expect to be contained in events of a certain type, so that they can write well-formed subscription expressions to match events in which they are interested.

The Repository acts as a common reference so that the events transmitted by suppliers can contain the right properties to match against subscriptions which contain expressions over those property names. It is complementary to the use of the interfaces NotifyPublish and NotifySubscribe, which convey the names of event types that are offered or required between consumers and suppliers of events. A supplier which indicates that it will supply events of a certain type, by passing the name of that type to the *offer_change* operation, can register that type in the Repository so that it can be looked up by clients receiving the offer.

Properties consist simply of a string name and a TypeCode to indicate the type of the value associated with that name. An Event Type can be composed of zero or more properties. These correspond to the properties in the *filterable_data* component of a Structured Event. They can also be interpreted as the named members of a struct,

which includes the string members *domain* and *event_type*. The variable name notation of the standard constraint grammar will allow either of these kinds of events to be matched,

Event Types also have names. In the simplest case they will have a string name that is an attribute. This name will be unique within the naming domain, which is indicated by the “domain” attribute of an Event Type. They also have a *full name* which is returned from an operation. In the simplest case this will be the same as its name attribute. In the default domain, represented by the empty string, this will always be the case. In other domains, however, the name of an Event Type can be derived from the Event Types which are inherited.

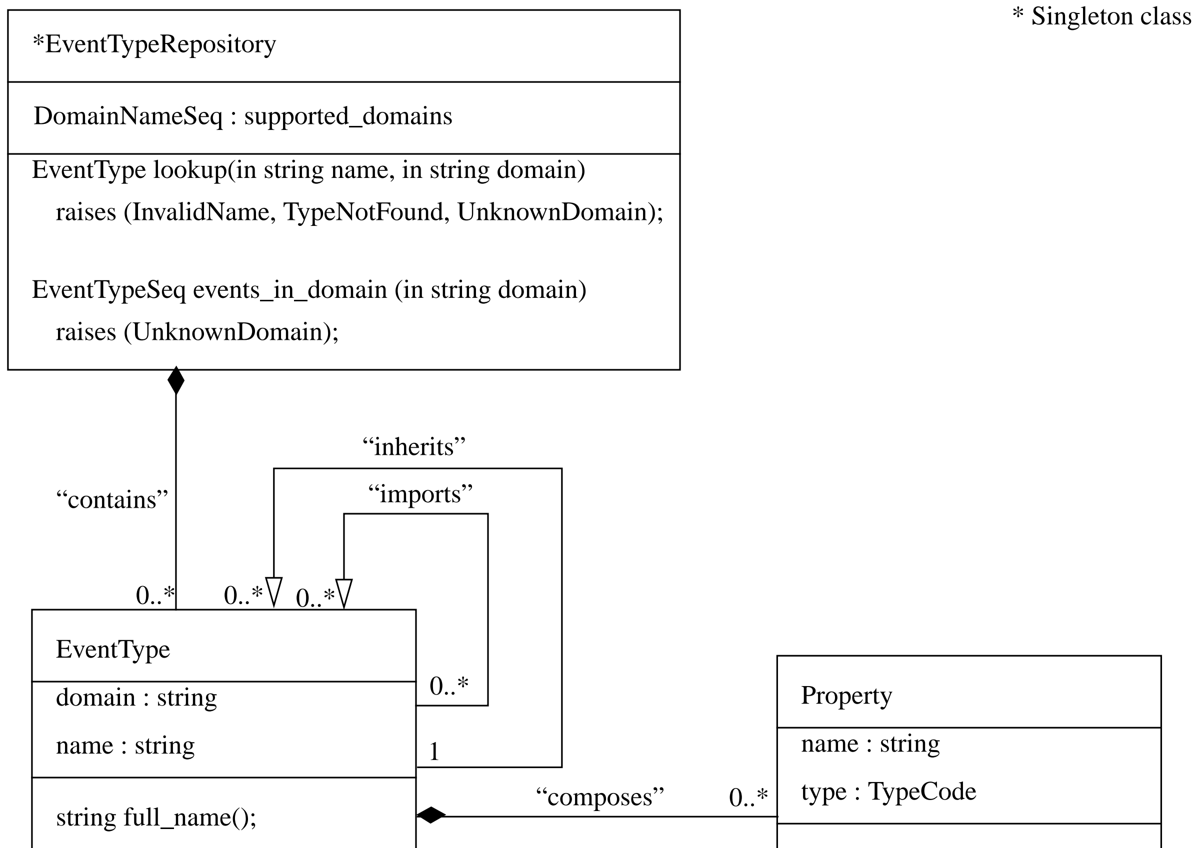

Figure A-1 UML meta-model of the Event Type Repository

New event types may be based upon the definition of existing event types in two ways:

- new types may *inherit* an existing type, implying that the new type “is an” instance of the existing type. In this case the name of the derived type may be generated from its base types’ names by some domain naming scheme.

- new types may *import* the definition of existing types to obtain a number of property names and types, but there is no asserted semantic relationship between them.

The naming scheme adopted by a domain is implementation dependent. An example would be the use of all the names of the derived types separated by dots. As only single inheritance is permitted, the composition of such names is very straight-forward.

Property names defined in an inherited or imported type cannot be re-defined or overridden in a new type. When importing properties from previously declared Event Types, the same property name may not be declared in more than one imported type unless the property type is also the same. For example an event type X:A containing a property K of type string may not be inherited or imported into the same derived type as X:B which contains a property K or type short.

A.2 Other functionality

The EventTypeRepository class has only one instance in any given implementation of a Repository. It provides a lookup interface to users of the repository so that they can find a event type by name, within its domain naming scheme. The default domain, nominated by the empty string, is a flat name space, and each of its event types must have a unique name. Implementations may implement other domains with the same naming scheme, or with a more complex naming scheme that reflects an inheritance hierarchy.

The EventTypeRepository also provides a mechanism to look up all the types in a particular domain, as well as an attribute indicating which domains it supports. All repositories will support the default domain, and must not return the empty string as one of the results of the supported_domains() attribute.

A.3 MODL Model

The Meta Object Definition Language (MODL) is a language for the textual representation of meta-models. It is explained in an appendix to the Meta Object Facility specification. The following MODL represents the same meta-model of event types as shown in UML in Figure A-1.

```
//
// Notification Service - Event Type Meta-Model
//
package Notification_Types {

  type Property {
    attribute string name;
    attribute TypeCode type_code;
  };

  type EventType {
    attribute string domain;
    attribute string name;
```

```
  string get_full_name();
};

exception InvalidName { string name; };
exception UnknownDomain { string domain; };
exception TypeNotFound { string name; };


singleton type EventTypeRepository {
  readonly attribute set [1..*] of string supported_domains;

  EventType lookup (in string name, in string domain)
    raises (InvalidName, TypeNotFound, UnknownDomain);

  ordered set [0..*] of EventType events_in_domain (in string domain)
    raises (UnknownDomain);
};


association Contains {
  role single EventTypeRepository container;
  composite role set [0..*] of EventType contained;
};

association Inherits {
  role set [0..*] of EventType sub_type;
  role single EventType super_type;
};

association Imports {
  role set [0..*] of EventType importer;
  role set [0..*] of EventType imported;
};

association Composes {
  role single EventType composition;
  composite role ordered set [0..*] of Property component;
};

};
```

A.4 Generated IDL

The following IDL is generated using the mapping for meta-models as specified in the MOF. It provides operations corresponding to the attributes and associations shown in the meta-model, and also inherits from the standard MOF interfaces defined in the module **Reflective**. This means that implementations of the repository can be interrogated using the operations specific to Event Types, and via the reflective interfaces that generic browsing tools use.

```
#ifndef Notification_Types_IDL
#define Notification_Types_IDL

#include <Reflective.idl>
```

```
//
//  Notification Service - Event Type Meta-Model
//

module NotificationTypes {

    typedef sequence < string > StringSet;
    interface NotificationTypesPackage;

    interface PropertyClass;
    interface Property;

    typedef sequence < Property > PropertyUList;
    interface EventTypeClass;
    interface EventType;

    typedef sequence < EventType > EventTypeSet;
    typedef sequence < EventType > EventTypeUList;
    interface EventTypeRepositoryClass;
    interface EventTypeRepository;

    typedef sequence < EventTypeRepository > EventTypeRepositoryUList;

    // typedef string TypeCode;

    interface PropertyClass
        : Reflective::RefObject
    {
        // get all property including subtypes of property
        readonly attribute PropertyUList all_of_kind_property;

        // get all property excluding subtypes of property
        readonly attribute PropertyUList all_of_type_property;

        // Factory operation for Property objects
        Property create_property (
            /* from Property */ in string name,
            /* from Property */ in TypeCode type_code)
            raises (Reflective::SemanticError);


    }; // end of interface PropertyClass

    interface Property :
        PropertyClass

    {

        string name ()
            raises (Reflective::StructuralError,
                Reflective::SemanticError);
        void set_name (in string new_value)
            raises (Reflective::SemanticError);
```

```
    TypeCode type_code ()
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void set_type_code (in TypeCode new_value)
        raises (Reflective::SemanticError);

}; // end of interface Property

interface EventTypeClass
    : Reflective::RefObject
{
    // get all event_type including subtypes of event_type
    readonly attribute EventTypeUList all_of_kind_event_type;

    // get all event_type excluding subtypes of event_type
    readonly attribute EventTypeUList all_of_type_event_type;

    // Factory operation for EventType objects
    EventType create_event_type (
        /* from EventType */ in string domain,
        /* from EventType */ in string name)
        raises (Reflective::SemanticError);

}; // end of interface EventTypeClass

interface EventType :
    EventTypeClass

{

    string domain ()
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void set_domain (in string new_value)
        raises (Reflective::SemanticError);

    string name ()
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void set_name (in string new_value)
        raises (Reflective::SemanticError);

    string get_full_name ()
        raises (
            Reflective::SemanticError);

}; // end of interface EventType

exception InvalidName {
    string name;
};

exception UnknownDomain {
    string domain;
};
```

```
exception TypeNotFound {
    string name;
};

interface EventTypeRepositoryClass
        : Reflective::RefObject
{
    // get all event_type_repository including subtypes of event_type_repository
        readonly attributeEventTypeRepositoryUListall_of_kind_event_type_repository;

    // get all event_type_repository excluding subtypes of event_type_repository
    readonly attribute EventTypeRepositoryUList all_of_type_event_type_repository;

    // Factory operation for EventTypeRepository objects
    EventTypeRepository create_event_type_repository (
        /* from EventTypeRepository */ in StringSet supported_domains)
        raises (Reflective::AlreadyCreated,
            Reflective::SemanticError);

}; // end of interface EventTypeRepositoryClass

interface EventTypeRepository :
    EventTypeRepositoryClass

{

    StringSet supported_domains ()
        raises (Reflective::SemanticError);

    EventType lookup (
            in string name,
            in string domain)
        raises (
            InvalidName,
            TypeNotFound,
            UnknownDomain,
            Reflective::SemanticError);

    EventTypeUList events_in_domain (
            in string domain)
        raises (
            UnknownDomain,
            Reflective::SemanticError);

}; // end of interface EventTypeRepository

// data types for Association Contains

struct ContainsLink {
    EventTypeRepository container;
    EventType contained;
};

typedef sequence <ContainsLink> ContainsLinkSet;
```

```
interface Contains : Reflective::RefAssociation {
    ContainsLinkSet all_Contains_links ();
    boolean exists (in EventTypeRepository container, in EventType contained);
    EventTypeSet with_container (in EventTypeRepository container);
    EventTypeRepository with_contained (in EventType contained);
    void add (in EventTypeRepository container, in EventType contained)
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void modify_container (in EventTypeRepository container, in EventType contained, in EventTypeRepository
new_container)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void modify_contained (in EventTypeRepository container, in EventType contained, in EventType
new_contained)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void remove (in EventTypeRepository container, in EventType contained)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
};

// data types for Association Inherits

struct InheritsLink {
    EventType sub_type;
    EventType super_type;
};

typedef sequence <InheritsLink> InheritsLinkSet;

interface Inherits : Reflective::RefAssociation {
    InheritsLinkSet all_Inherits_links ();
    boolean exists (in EventType sub_type, in EventType super_type);
    EventType with_sub_type (in EventType sub_type);
    EventTypeSet with_super_type (in EventType super_type);
    void add (in EventType sub_type, in EventType super_type)
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void modify_sub_type (in EventType sub_type, in EventType super_type, in EventType new_sub_type)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void modify_super_type (in EventType sub_type, in EventType super_type, in EventType new_super_type)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void remove (in EventType sub_type, in EventType super_type)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
};
```

```
// data types for Association Imports

struct ImportsLink {
    EventType importer;
    EventType imported;
};

typedef sequence <ImportsLink> ImportsLinkSet;

interface Imports : Reflective::RefAssociation {
    ImportsLinkSet all_Imports_links ();
    boolean exists (in EventType importer, in EventType imported);
    EventTypeSet with_importer (in EventType importer);
    EventTypeSet with_imported (in EventType imported);
    void add (in EventType importer, in EventType imported)
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void modify_importer (in EventType importer, in EventType imported, in EventType new_importer)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void modify_imported (in EventType importer, in EventType imported, in EventType new_imported)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void remove (in EventType importer, in EventType imported)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
};

// data types for Association Composes

struct ComposesLink {
    EventType composition;
    Property component;
};

typedef sequence <ComposesLink> ComposesLinkSet;

interface Composes : Reflective::RefAssociation {
    ComposesLinkSet all_Composes_links ();
    boolean exists (in EventType composition, in Property component);
    PropertyUList with_composition (in EventType composition);
    EventType with_component (in Property component);
    void add (in EventType composition, in Property component)
        raises (Reflective::StructuralError,
            Reflective::SemanticError);
    void add_before_component (in EventType composition, in Property component, in Property before)
        raises (Reflective::StructuralError,
            Reflective::NotFound,
            Reflective::SemanticError);
    void modify_composition (in EventType composition, in Property component, in EventType
new_composition)
```

```
            raises (Reflective::StructuralError,
                Reflective::NotFound,
                Reflective::SemanticError);
        void modify_component (in EventType composition, in Property component, in Property new_component)
            raises (Reflective::StructuralError,
            Reflective::NotFound,
                Reflective::SemanticError);
        void remove (in EventType composition, in Property component)
            raises (Reflective::StructuralError,
                Reflective::NotFound,
                Reflective::SemanticError);
    };

    interface NotificationTypesPackageFactory
    {
    NotificationTypesPackage create_notification_types_package ()
            raises (
                Reflective::SemanticError);

    };

    interface NotificationTypesPackage
                    : Reflective::RefPackage
    {
        readonly attribute PropertyClass property_class_ref;
        readonly attribute EventTypeClass event_type_class_ref;
        readonly attribute EventTypeRepositoryClass event_type_repository_class_ref;
        readonly attribute Contains contains_ref;
        readonly attribute Inherits inherits_ref;
        readonly attribute Imports imports_ref;
        readonly attribute Composes composes_ref;
        };

    }; // end of module NotificationTypes

    #endif
    // end of IDL generation
```

Complete IDL — B

The following lists the full IDL of the Notification Service:

```
module CosNotification {

    typedef string Istring;
    typedef Istring PropertyName;
    typedef any PropertyValue;


    struct Property {
    PropertyName name;
    PropertyValue value;
    };
    typedef sequence<Property> PropertySeq;


    // The following are the same, but serve different purposes.
    typedef PropertySeq OptionalHeaderFields;
    typedef PropertySeq FilterableEventBody;
    typedef PropertySeq QoSProperties;
    typedef PropertySeq AdminProperties;


    struct EventType {
    string domain_name;
    string type_name;
    };
    typedef sequence<EventType> EventTypeSeq;


    struct PropertyRange {
    PropertyValue low_val;
    PropertyValue high_val;
    };
```

```
struct NamedPropertyRange {
PropertyName name;
PropertyRange range;
};
typedef sequence<NamedPropertyRange> NamedPropertyRangeSeq;

enum QoSError_code {
UNSUPPORTED_PROPERTY,
UNAVAILABLE_PROPERTY,
UNSUPPORTED_VALUE,
UNAVAILABLE_VALUE,
BAD_PROPERTY,
BAD_TYPE,
BAD_VALUE
};

struct PropertyError {
QoSError_code code;
PropertyName name;
PropertyRange available_range;
};
typedef sequence<PropertyError> PropertyErrorSeq;

exception UnsupportedQoS { PropertyErrorSeq qos_err; };
exception UnsupportedAdmin { PropertyErrorSeq admin_err; };

// Define the Structured Event structure
struct FixedEventHeader {
EventType event_type;
string event_name;
};

struct EventHeader {
FixedEventHeader fixed_header;
OptionalHeaderFields variable_header;
};

struct StructuredEvent {
EventHeader header;
FilterableEventBody filterable_data;
any remainder_of_body;
}; // StructuredEvent
typedef sequence<StructuredEvent> EventBatch;

// The following constant declarations define the standard
// QoS property names and the associated values each property
// can take on. The name/value pairs for each standard property
```

```
// are grouped, beginning with a string constant defined for the
// property name, followed by the values the property can take on.

const string EventReliability = “EventReliability”;
const short BestEffort = 0;
const short Persistent = 1;

const string ConnectionReliability = “ConnectionReliability”;
// Can take on the same values as EventReliability

const string Priority = “Priority”;
const short LowestPriority = -32767;
const short HighestPriority = 32767;
const short DefaultPriority = 0;

const string StartTime = “StartTime”;
// StartTime takes a value of type TimeBase::UtcT.

const string StopTime = “StopTime”;
// StopTime takes a value of type TimeBase::UtcT.

const string Timeout = “Timeout”;
// Timeout takes on a value of type TimeBase::TimeT

const string OrderPolicy = “OrderPolicy”;
const short AnyOrder = 0;
const short FifoOrder = 1;
const short PriorityOrder = 2;
const short DeadlineOrder = 3;

const string DiscardPolicy = “DiscardPolicy”;
// DiscardPolicy takes on the same values as OrderPolicy, plus
const short LifoOrder = 4;

const string MaximumBatchSize = “MaximumBatchSize”;
// MaximumBatchSize takes on a value of type long

const string PacingInterval = “PacingInterval”;
// PacingInterval takes on a value of type TimeBase::TimeT

const string StartTimeSupported = “StartTimeSupported”;
// StartTimeSupported takes on a boolean value
```

```
const string StopTimeSupported = “StopTimeSupported”;
// StopTimeSupported takes on a boolean value

const string MaxEventsPerConsumer = “MaxEventsPerConsumer”;
// MaxEventsPerConsumer takes on a value of type long

interface QoSAdmin {

QoSProperties get_qos();

void set_qos ( in QoSProperties qos)
    raises ( UnsupportedQoS );

void validate_qos (
    in QoSProperties required_qos,
    out NamedPropertyRangeSeq available_qos )
    raises ( UnsupportedQoS );

}; // QosAdmin

// Admin properties are defined in similar manner as QoS
// properties. The only difference is that these properties
// are related to channel administration policies, as opposed
// message quality of service

const string MaxQueueLength = “MaxQueueLength”;
// MaxQueueLength takes on a value of type long

const string MaxConsumers = “MaxConsumers”;
// MaxConsumers takes on a value of type long

const string MaxSuppliers = “MaxSuppliers”;
// MaxSuppliers takes on a value of type long

const string RejectNewEvents = “RejectNewEvents”;
// RejectNewEvents takes on a value of type Boolean

interface AdminPropertiesAdmin {

AdminProperties get_admin();
```

```
void set_admin (in AdminProperties admin)
    raises ( UnsupportedAdmin);

}; // AdminPropertiesAdmin

}; // CosNotification

module CosNotifyFilter {

    typedef long ConstraintID;

    struct ConstraintExp {
    CosNotification::EventTypeSeq event_types;
    string constraint_expr;
    };

    typedef sequence<ConstraintID> ConstraintIDSeq;
    typedef sequence<ConstraintExp> ConstraintExpSeq;

    struct ConstraintInfo {
    ConstraintExp constraint_expression;
    ConstraintID constraint_id;
    };

    typedef sequence<ConstraintInfo> ConstraintInfoSeq;

    struct MappingConstraintPair {
    ConstraintExp constraint_expression;
    any result_to_set;
    };

    typedef sequence<MappingConstraintPair> MappingConstraintPairSeq;

    struct MappingConstraintInfo {
    ConstraintExp constraint_expression;
    ConstraintID constraint_id;
    any value;
    };

    typedef sequence<MappingConstraintInfo> MappingConstraintInfoSeq;

    typedef long CallbackID;
```

```
typedef sequence<CallbackID> CallbackIDSeq;

exception UnsupportedFilterableData {};
exception InvalidGrammar {};
exception InvalidConstraint {ConstraintExp constr;};
exception DuplicateConstraintID {ConstraintID id;};

exception ConstraintNotFound {ConstraintID id;};
exception CallbackNotFound {};

exception InvalidValue {ConstraintExp constr; any value;};

interface Filter {

readonly attribute string constraint_grammar;

ConstraintInfoSeq add_constraints (
    in ConstraintExpSeq constraint_list)
    raises (InvalidConstraint);

void modify_constraints (
    in ConstraintIDSeq del_list,
    in ConstraintInfoSeq modify_list)
    raises (InvalidConstraint, ConstraintNotFound);

ConstraintInfoSeq get_constraints(
    in ConstraintIDSeq id_list)
    raises (ConstraintNotFound);

ConstraintInfoSeq get_all_constraints();

void remove_all_constraints();

void destroy();

boolean match ( in any filterable_data )
    raises (UnsupportedFilterableData);

boolean match_structured (
    in CosNotification::StructuredEvent filterable_data )
    raises (UnsupportedFilterableData);
```

```
boolean match_typed (
      in CosNotification::PropertySeq filterable_data )
      raises (UnsupportedFilterableData);

CallbackID attach_callback (
      in CosNotifyComm::NotifySubscribe callback);

void detach_callback ( in CallbackID callback)
      raises ( CallbackNotFound );

CallbackIDSeq get_callbacks();

}; // Filter

interface MappingFilter {

readonly attribute string constraint_grammar;

readonly attribute CORBA::TypeCode value_type;

readonly attribute any default_value;

MappingConstraintInfoSeq add_mapping_constraints (
      in MappingConstraintPairSeq pair_list)
      raises (InvalidConstraint, InvalidValue);

void modify_mapping_constraints (
      in ConstraintIDSeq del_list,
      in MappingConstraintInfoSeq modify_list)
      raises (InvalidConstraint, InvalidValue,
            ConstraintNotFound);

MappingConstraintInfoSeq get_mapping_constraints (
      in ConstraintIDSeq id_list)
      raises (ConstraintNotFound);

MappingConstraintInfoSeq get_all_mapping_constraints();

void remove_all_mapping_constraints();
```

```
void destroy();

boolean match ( in any filterable_data,
      out any result_to_set )
      raises (UnsupportedFilterableData);

boolean match_structured (
      in CosNotification::StructuredEvent filterable_data,
      out any result_to_set)
      raises (UnsupportedFilterableData);

boolean match_typed (
      in CosNotification::PropertySeq filterable_data,
      out any result_to_set)
      raises (UnsupportedFilterableData);

}; // MappingFilter

interface FilterFactory {

Filter create_filter (
      in string constraint_grammar)
      raises (InvalidGrammar);

MappingFilter create_mapping_filter (
      in string constraint_grammar,
      in any default_value)
      raises(InvalidGrammar);

}; // FilterFactory

typedef long FilterID;
typedef sequence<FilterID> FilterIDSeq;

exception FilterNotFound {};

interface FilterAdmin {

FilterID add_filter ( in Filter new_filter );

void remove_filter ( in FilterID filter )
      raises ( FilterNotFound );
```

```
        Filter get_filter ( in FilterID filter )
            raises ( FilterNotFound );

        FilterIDSeq get_all_filters();

        void remove_all_filters();

        }; // FilterAdmin

}; // CosNotifyFilter


module CosNotifyComm {

        exception InvalidEventType { CosNotification::EventType type; };

        interface NotifyPublish {

        void offer_change (
            in CosNotification::EventTypeSeq added,
            in CosNotification::EventTypeSeq removed )
            raises ( InvalidEventType );

        }; // NotifyPublish

        interface NotifySubscribe {

        void subscription_change(
            in CosNotification::EventTypeSeq added,
            in CosNotification::EventTypeSeq removed )
            raises ( InvalidEventType );

        }; // NotifySubscribe

        interface PushConsumer :
            NotifyPublish,
            CosEventComm::PushConsumer {
            }; // PushConsumer
```

```
interface PullConsumer :
NotifyPublish,
CosEventComm::PullConsumer {
}; // PullConsumer

interface PullSupplier :
NotifySubscribe,
CosEventComm::PullSupplier {
}; // PullSupplier

interface PushSupplier :
NotifySubscribe,
CosEventComm::PushSupplier {
};

interface StructuredPushConsumer : NotifyPublish {

void push_structured_event(
    in CosNotification::StructuredEvent notification)
    raises(CosEventComm::Disconnected);

void disconnect_structured_push_consumer();

}; // StructuredPushConsumer

interface StructuredPullConsumer : NotifyPublish {
void disconnect_structured_pull_consumer();
}; // StructuredPullConsumer

interface StructuredPullSupplier : NotifySubscribe {

CosNotification::StructuredEvent pull_structured_event()
    raises(CosEventComm::Disconnected);
CosNotification::StructuredEvent try_pull_structured_event(
    out boolean has_event)
    raises(CosEventComm::Disconnected);

void disconnect_structured_pull_supplier();

}; // StructuredPullSupplier

interface StructuredPushSupplier : NotifySubscribe {
void disconnect_structured_push_supplier();
```

```
}; // StructuredPushSupplier

interface SequencePushConsumer : NotifyPublish {

void push_structured_events(
    in CosNotification::EventBatch notifications)
    raises(CosEventComm::Disconnected);

void disconnect_sequence_push_consumer();

}; // SequencePushConsumer

interface SequencePullConsumer : NotifyPublish {
void disconnect_sequence_pull_consumer();
}; // SequencePullConsumer

interface SequencePullSupplier : NotifySubscribe {

CosNotification::EventBatch pull_structured_events(
    in long max_number )
    raises(CosEventComm::Disconnected);

CosNotification::EventBatch try_pull_structured_events(
    in long max_number,
    out boolean has_event)
    raises(CosEventComm::Disconnected);

void disconnect_sequence_pull_supplier();

}; // SequencePullSupplier

interface SequencePushSupplier : NotifySubscribe {
void disconnect_sequence_push_supplier();
}; // SequencePushSupplier

}; // CosNotifyComm


module CosNotifyChannelAdmin {

    exception ConnectionAlreadyActive {};
```

```
exception ConnectionAlreadyInactive {};
exception NotConnected {};

// Forward declarations
interface ConsumerAdmin;
interface SupplierAdmin;
interface EventChannel;
interface EventChannelFactory;

enum ProxyType {
PUSH_ANY,
PULL_ANY,
PUSH_STRUCTURED,
PULL_STRUCTURED,
PUSH_SEQUENCE,
PULL_SEQUENCE,
PUSH_TYPED,
PULL_TYPED
};

enum ObtainInfoMode {
ALL_NOW_UPDATES_OFF,
ALL_NOW_UPDATES_ON,
NONE_NOW_UPDATES_OFF,
NONE_NOW_UPDATES_ON
};

interface ProxyConsumer :
    CosNotification::QoSAdmin,
    CosNotifyFilter::FilterAdmin {

readonly attribute ProxyType MyType;
readonly attribute SupplierAdmin MyAdmin;

CosNotification::EventTypeSeq obtain_subscription_types(
    in ObtainInfoMode mode );

void validate_event_qos (
        in CosNotification::QoSProperties required_qos,
        out CosNotification::NamedPropertyRangeSeq available_qos)
    raises (CosNotification::UnsupportedQoS);

}; // ProxyConsumer

interface ProxySupplier :
    CosNotification::QoSAdmin,
```

```
      CosNotifyFilter::FilterAdmin {

readonly attribute ProxyType MyType;
readonly attribute ConsumerAdmin MyAdmin;

attribute CosNotifyFilter::MappingFilter priority_filter;
attribute CosNotifyFilter::MappingFilter lifetime_filter;

CosNotification::EventTypeSeq obtain_offered_types(
      in ObtainInfoMode mode );

void validate_event_qos (
            in CosNotification::QoSProperties required_qos,
            out CosNotification::NamedPropertyRangeSeq available_qos)
      raises (CosNotification::UnsupportedQoS);

}; // ProxySupplier

interface ProxyPushConsumer :
      ProxyConsumer,
      CosNotifyComm::PushConsumer {

void connect_any_push_supplier (
      in CosEventComm::PushSupplier push_supplier)
      raises(CosEventChannelAdmin::AlreadyConnected);

}; // ProxyPushConsumer

interface StructuredProxyPushConsumer :
      ProxyConsumer,
      CosNotifyComm::StructuredPushConsumer {

void connect_structured_push_supplier (
      in CosNotifyComm::StructuredPushSupplier push_supplier)
      raises(CosEventChannelAdmin::AlreadyConnected);

}; // StructuredProxyPushConsumer

interface SequenceProxyPushConsumer :
      ProxyConsumer,
      CosNotifyComm::SequencePushConsumer {
```

```
void connect_sequence_push_supplier (
    in CosNotifyComm::SequencePushSupplier push_supplier)
    raises(CosEventChannelAdmin::AlreadyConnected);

}; // SequenceProxyPushConsumer

interface ProxyPullSupplier :
    ProxySupplier,
    CosNotifyComm::PullSupplier {

void connect_any_pull_consumer (
    in CosEventComm::PullConsumer pull_consumer)
    raises(CosEventChannelAdmin::AlreadyConnected);

}; // ProxyPullSupplier

interface StructuredProxyPullSupplier :
    ProxySupplier,
    CosNotifyComm::StructuredPullSupplier {

void connect_structured_pull_consumer (
    in CosNotifyComm::StructuredPullConsumer pull_consumer)
    raises(CosEventChannelAdmin::AlreadyConnected);

}; // StructuredProxyPullSupplier

interface SequenceProxyPullSupplier :
    ProxySupplier,
    CosNotifyComm::SequencePullSupplier {

void connect_sequence_pull_consumer (
    in CosNotifyComm::SequencePullConsumer pull_consumer)
    raises(CosEventChannelAdmin::AlreadyConnected);

}; // SequenceProxyPullSupplier

interface ProxyPullConsumer :
    ProxyConsumer,
    CosNotifyComm::PullConsumer {

void connect_any_pull_supplier (
    in CosEventComm::PullSupplier pull_supplier)
    raises(CosEventChannelAdmin::AlreadyConnected,
```

```
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises(ConnectionAlreadyInactive, NotConnected);

void resume_connection()
    raises(ConnectionAlreadyActive, NotConnected);

}; // ProxyPullConsumer

interface StructuredProxyPullConsumer :
    ProxyConsumer,
    CosNotifyComm::StructuredPullConsumer {

void connect_structured_pull_supplier (
    in CosNotifyComm::StructuredPullSupplier pull_supplier)
    raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises(ConnectionAlreadyInactive, NotConnected);

void resume_connection()
    raises(ConnectionAlreadyActive, NotConnected);

}; // StructuredProxyPullConsumer

interface SequenceProxyPullConsumer :
    ProxyConsumer,
    CosNotifyComm::SequencePullConsumer {

void connect_sequence_pull_supplier (
    in CosNotifyComm::SequencePullSupplier pull_supplier)
    raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises(ConnectionAlreadyInactive, NotConnected);

void resume_connection()
    raises(ConnectionAlreadyActive, NotConnected);
```

```
}; // SequenceProxyPullConsumer

interface ProxyPushSupplier :
    ProxySupplier,
    CosNotifyComm::PushSupplier {

void connect_any_push_consumer (
    in CosEventComm::PushConsumer push_consumer)
    raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises(ConnectionAlreadyInactive, NotConnected);

void resume_connection()
    raises(ConnectionAlreadyActive, NotConnected);

}; // ProxyPushSupplier

interface StructuredProxyPushSupplier :
    ProxySupplier,
    CosNotifyComm::StructuredPushSupplier {

void connect_structured_push_consumer (
    in CosNotifyComm::StructuredPushConsumer push_consumer)
    raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises(ConnectionAlreadyInactive, NotConnected);

void resume_connection()
    raises(ConnectionAlreadyActive, NotConnected);

}; // StructuredProxyPushSupplier

interface SequenceProxyPushSupplier :
    ProxySupplier,
    CosNotifyComm::SequencePushSupplier {

void connect_sequence_push_consumer (
    in CosNotifyComm::SequencePushConsumer push_consumer)
    raises(CosEventChannelAdmin::AlreadyConnected,
```

```
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises(ConnectionAlreadyInactive, NotConnected);

void resume_connection()
    raises(ConnectionAlreadyActive, NotConnected);

}; // SequenceProxyPushSupplier

typedef long ProxyID;
typedef sequence <ProxyID> ProxyIDSeq;

enum ClientType {
ANY_EVENT,
STRUCTURED_EVENT,
SEQUENCE_EVENT
};

enum InterFilterGroupOperator { AND_OP, OR_OP };

typedef long AdminID;
typedef sequence<AdminID> AdminIDSeq;

exception AdminNotFound {};
exception ProxyNotFound {};

struct AdminLimit {
CosNotification::PropertyName name;
CosNotification::PropertyValue value;
};

exception AdminLimitExceeded { AdminLimit admin_property_err; };

interface ConsumerAdmin :
    CosNotification::QoSAdmin,
    CosNotifyComm::NotifySubscribe,
    CosNotifyFilter::FilterAdmin,
    CosEventChannelAdmin::ConsumerAdmin {

readonly attribute AdminID MyID;
readonly attribute EventChannel MyChannel;
```

```
readonly attribute InterFilterGroupOperator MyOperator;

attribute CosNotifyFilter::MappingFilter priority_filter;
attribute CosNotifyFilter::MappingFilter lifetime_filter;

readonly attribute ProxyIDSeq pull_suppliers;
readonly attribute ProxyIDSeq push_suppliers;

ProxySupplier get_proxy_supplier (
      in ProxyID proxy_id )
      raises ( ProxyNotFound );

ProxySupplier obtain_notification_pull_supplier (
      in ClientType ctype,
      out ProxyID proxy_id)
      raises ( AdminLimitExceeded );

ProxySupplier obtain_notification_push_supplier (
      in ClientType ctype,
      out ProxyID proxy_id)
      raises ( AdminLimitExceeded );

void destroy();

}; // ConsumerAdmin

interface SupplierAdmin :
      CosNotification::QoSAdmin,
      CosNotifyComm::NotifyPublish,
      CosNotifyFilter::FilterAdmin,
      CosEventChannelAdmin::SupplierAdmin {

readonly attribute AdminID MyID;
readonly attribute EventChannel MyChannel;

readonly attribute InterFilterGroupOperator MyOperator;

readonly attribute ProxyIDSeq pull_consumers;
readonly attribute ProxyIDSeq push_consumers;

ProxyConsumer get_proxy_consumer (
      in ProxyID proxy_id )
```

```
raises ( ProxyNotFound );

ProxyConsumer obtain_notification_pull_consumer (
    in ClientType ctype,
    out ProxyID proxy_id)
    raises ( AdminLimitExceeded );

ProxyConsumer obtain_notification_push_consumer (
    in ClientType ctype,
    out ProxyID proxy_id)
    raises ( AdminLimitExceeded );

void destroy();

}; // SupplierAdmin

interface EventChannel :
CosNotification::QoSAdmin,
CosNotification::AdminPropertiesAdmin,
CosEventChannelAdmin::EventChannel {

readonly attribute EventChannelFactory MyFactory;

readonly attribute ConsumerAdmin default_consumer_admin;
readonly attribute SupplierAdmin default_supplier_admin;

readonly attribute CosNotifyFilter::FilterFactory
                    default_filter_factory;

ConsumerAdmin new_for_consumers(
    in InterFilterGroupOperator op,
    out AdminID id );

SupplierAdmin new_for_suppliers(
    in InterFilterGroupOperator op,
    out AdminID id );

ConsumerAdmin get_consumeradmin ( in AdminID id )
    raises (AdminNotFound);

SupplierAdmin get_supplieradmin ( in AdminID id )
    raises (AdminNotFound);
```

```
        AdminIDSeq get_all_consumeradmins();
        AdminIDSeq get_all_supplieradmins();

    }; // EventChannel

    typedef long ChannelID;
    typedef sequence<ChannelID> ChannelIDSeq;

    exception ChannelNotFound {};

    interface EventChannelFactory {

        EventChannel create_channel (
            in CosNotification::QoSProperties initial_qos,
            in CosNotification::AdminProperties initial_admin,
            out ChannelID id)
            raises(CosNotification::UnsupportedQoS,
                CosNotification::UnsupportedAdmin );

        ChannelIDSeq get_all_channels();

        EventChannel get_event_channel ( in ChannelID id )
            raises (ChannelNotFound);

    }; // EventChannelFactory

}; // CosNotifyChannelAdmin


module CosTypedNotifyComm {

    interface TypedPushConsumer :
    CosTypedEventComm::TypedPushConsumer,
    CosNotifyComm::NotifyPublish {
    }; // TypedPushConsumer

    interface TypedPullSupplier :
    CosTypedEventComm::TypedPullSupplier,
    CosNotifyComm::NotifySubscribe {
    }; // TypedPullSupplier
```

```
}; // CosTypedNotifyComm


module CosTypedNotifyChannelAdmin {

    // Forward declaration
    interface TypedEventChannelFactory;


    typedef string Key;


    interface TypedProxyPushConsumer :
        CosNotifyChannelAdmin::ProxyConsumer,
        CosTypedNotifyComm::TypedPushConsumer {


    void connect_typed_push_supplier (
        in CosEventComm::PushSupplier push_supplier )
        raises ( CosEventChannelAdmin::AlreadyConnected );


    }; // TypedProxyPushConsumer


    interface TypedProxyPullSupplier :
        CosNotifyChannelAdmin::ProxySupplier,
        CosTypedNotifyComm::TypedPullSupplier {


    void connect_typed_pull_consumer (
        in CosEventComm::PullConsumer pull_consumer )
        raises ( CosEventChannelAdmin::AlreadyConnected );


    }; // TypedProxyPullSupplier


    interface TypedProxyPullConsumer :
        CosNotifyChannelAdmin::ProxyConsumer,
        CosNotifyComm::PullConsumer {


    void connect_typed_pull_supplier (
        in CosTypedEventComm::TypedPullSupplier pull_supplier)
        raises ( CosEventChannelAdmin::AlreadyConnected,
            CosEventChannelAdmin::TypeError );


    void suspend_connection()
        raises (CosNotifyChannelAdmin::ConnectionAlreadyInactive,
            CosNotifyChannelAdmin::NotConnected);
```

```
void resume_connection()
    raises (CosNotifyChannelAdmin::ConnectionAlreadyActive,
        CosNotifyChannelAdmin::NotConnected);

}; // TypedProxyPullConsumer

interface TypedProxyPushSupplier :
    CosNotifyChannelAdmin::ProxySupplier,
    CosNotifyComm::PushSupplier {

void connect_typed_push_consumer (
    in CosTypedEventComm::TypedPushConsumer push_consumer)
    raises ( CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

void suspend_connection()
    raises (CosNotifyChannelAdmin::ConnectionAlreadyInactive,
        CosNotifyChannelAdmin::NotConnected);

void resume_connection()
    raises (CosNotifyChannelAdmin::ConnectionAlreadyActive,
        CosNotifyChannelAdmin::NotConnected);

}; // TypedProxyPushSupplier

interface TypedConsumerAdmin :
    CosNotifyChannelAdmin::ConsumerAdmin,
    CosTypedEventChannelAdmin::TypedConsumerAdmin {

 TypedProxyPullSupplier obtain_typed_notification_pull_supplier(
    in Key supported_interface,
    out CosNotifyChannelAdmin::ProxyID proxy_id )
    raises( CosTypedEventChannelAdmin::InterfaceNotSupported,
        CosNotifyChannelAdmin::AdminLimitExceeded );

 TypedProxyPushSupplier obtain_typed_notification_push_supplier(
    in Key uses_interface,
    out CosNotifyChannelAdmin::ProxyID proxy_id )
    raises( CosTypedEventChannelAdmin::NoSuchImplementation,
        CosNotifyChannelAdmin::AdminLimitExceeded );

}; // TypedConsumerAdmin

interface TypedSupplierAdmin :
```

```
        CosNotifyChannelAdmin::SupplierAdmin,
        CosTypedEventChannelAdmin::TypedSupplierAdmin {

  TypedProxyPushConsumer obtain_typed_notification_push_consumer(
        in Key supported_interface,
        out CosNotifyChannelAdmin::ProxyID proxy_id )
        raises( CosTypedEventChannelAdmin::InterfaceNotSupported,
              CosNotifyChannelAdmin::AdminLimitExceeded );

  TypedProxyPullConsumer obtain_typed_notification_pull_consumer(
        in Key uses_interface,
        out CosNotifyChannelAdmin::ProxyID proxy_id )
        raises( CosTypedEventChannelAdmin::NoSuchImplementation,
              CosNotifyChannelAdmin::AdminLimitExceeded );

}; // TypedSupplierAdmin


interface TypedEventChannel :
        CosNotification::QoSAdmin,
        CosNotification::AdminPropertiesAdmin,
        CosTypedEventChannelAdmin::TypedEventChannel {

readonly attribute TypedEventChannelFactory MyFactory;

readonly attribute TypedConsumerAdmin default_consumer_admin;
readonly attribute TypedSupplierAdmin default_supplier_admin;

readonly attribute CosNotifyFilter::FilterFactory
                              default_filter_factory;

TypedConsumerAdmin new_for_typed_notification_consumers(
        in CosNotifyChannelAdmin::InterFilterGroupOperator op,
        out CosNotifyChannelAdmin::AdminID id );

TypedSupplierAdmin new_for_typed_notification_suppliers(
        in CosNotifyChannelAdmin::InterFilterGroupOperator op,
        out CosNotifyChannelAdmin::AdminID id );

TypedConsumerAdmin get_consumeradmin (
        in CosNotifyChannelAdmin::AdminID id )
        raises ( CosNotifyChannelAdmin::AdminNotFound );

TypedSupplierAdmin get_supplieradmin (
```

```
        in CosNotifyChannelAdmin::AdminID id )
        raises ( CosNotifyChannelAdmin::AdminNotFound );

    CosNotifyChannelAdmin::AdminIDSeq get_all_consumeradmins();
    CosNotifyChannelAdmin::AdminIDSeq get_all_supplieradmins();

    }; // TypedEventChannel

    interface TypedEventChannelFactory {

    TypedEventChannel create_typed_channel (
        in CosNotification::QoSProperties initial_QoS,
        in CosNotification::AdminProperties initial_admin,
        out CosNotifyChannelAdmin::ChannelID id)
        raises( CosNotification::UnsupportedQoS,
            CosNotification::UnsupportedAdmin );

    CosNotifyChannelAdmin::ChannelIDSeq get_all_typed_channels();

    TypedEventChannel get_typed_event_channel (
        in CosNotifyChannelAdmin::ChannelID id )
        raises ( CosNotifyChannelAdmin::ChannelNotFound );

    }; // TypedEventChannelFactory

}; // CosTypedNotifyChannelAdmin
```

Changes to CORBA — C

C.1 Changes to the CORBA Standard

C.1.1 A New Standard Exception

This specification defines a user exception called UnsupportedQoS that is raised within many invocations to indicate that a component of the notification channel, or the channel itself, is unable to satisfy a client's QoS request. In addition, however, there may be cases in which the client requests QoS at the message level when sending an event to the channel (e.g., by specifying QoS properties within the header of a Structured Event). Note, though, that it would only make sense to raise an exception in the case when the client is delivering the event to the channel, but not when using the same send operation (e.g., **push_structured_event**) for the channel to deliver the event to the consumer. For this reason, we feel it is appropriate to introduce a new standard system exception in CORBA, called BAD_QOS, which can be raised whenever a supplier sends an event to a channel over a connection that cannot support the request message-level QoS.

Section 3.15.1 of the CORBA specification will have the following IDL text appended:

```
exception BAD_QOS ex_body; // bad quality of service
```

A new section 3.15.4 will be added:

3.15.4 Bad Quality of Service

> The BAD_QOS exception is raised whenever an object cannot support the quality of service required by an invocation parameter that has a quality of service semantics.

Note that this same exception will be meaningful with regard to the Messaging Service specification under roughly the same development schedule as this specification. We highly recommend this exception be used for the purpose described above within implementations of both the Notification and Messaging Services.

C.1.2 Resolving Initial References

This service may be required, in some installations, to be accessible via the ORB's **resolve_initial_references** operation. In order to facilitate this the following changes to CORBA v2.1 are required:

Section 5.6 will be changed so that the following text:

> In addition to defining the id, the type of object being returned must be defined (i.e., "InterfaceRepository" returns a object of type Repository, and "NameService" returns a CosNamingContext object).

is replaced by:

In addition to defining the id, the type of object being returned must be defined. The following table represents the ids and types of current object services:

Service Id	**Object Type**
InterfaceRepository	CORBA::Repository
NameService	CosNaming::Context
TradingService	CosTrading::Lookup
NotificationService	CosNotifyChannelAdmin:: EventChannelFactory
TypedNotificationService	CosTypedNotifyChannelAdm in:: TypedEventChannelFactory

Also, the following text:

> In the future, specifications for Object Services (in *CORBAservices: Common Object Services Specification*) will state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not (i.e., whether the service is necessary or desirable for bootstrap purposes).

is replaced by:

In the future, specifications for Object Services (in *CORBAservices: Common Object Services Specification*) will state whether it is desirable that a service's initial reference be made available via the **resolve_initial_references** operation or not (i.e., whether the service is necessary or desirable for bootstrap purposes). If so, then the service must specify a change to the table above to add the necessary id and type information.

C.2 RFP Requirement Not Addressed

Note that a conscious decision was made to not address the Notification Service RFP requirement related to Federated Channels. Essentially, satisfaction of that requirement encompasses the specification of interfaces that support creation and management of networks of connected notification channels. During the authors' discussion of that topic, we determined that it raises many complex issues related to transactions, security, and unique message identification.

Thus, due to the fact that considerable effort was spent to address all other requirements from the Notification Service RFP within the specification of the notification channel itself, and that networks of notification channels raise several additional issues of considerable complexity, we decided to defer addressing issues related to federations of notification channels altogether. *The authors of this Notification Service specification feel this is a very important topic, however, and we highly recommend to the OMG Telecommunications Domain Task Force that a new RFP be drafted to specifically address this issue.*

Index

A
AdminPropertiesAdmin Interface
  get_admin 3-9
  set_admin 3-9

B
BAD_QOS system exception 2-43

C
Changes to the CORBA Standard C-1
CORBA
  contributors iv
  documentation set iv
CosNotification Module 3-2
CosNotifyChannelAdmin Module 3-41
CosNotifyComm Module 3-28
CosNotifyFilter Module 3-9
CosTypedNotifyChannelAdmin 3-85
CosTypedNotifyComm Module 3-84

D
Default Filter Constraint Language 2-23

E
End-to-End QoS 2-36
Event Filtering 2-17
Event Type Meta-Model A-1
Event Type Repository 2-55
EventBatch Data Type 3-7
EventChannel Interface
  default_consumer_admin 3-80
  default_filter_factory 3-81
  default_supplier_admin 3-80
  get_all_consumeradmins 3-82
  get_all_supplieradmins 3-82
  get_consumeradmin 3-81
  get_supplieradmin 3-82
  MyFactory 3-80
  new_for_consumers 3-81
  new_for_suppliers 3-81
EventChannelFactory
  create_channel 3-83
EventChannelFactory Interface
  get_all_channels 3-83
  get_event_channel 3-83

F
Filter Interface
  add_constraints 3-16
  attach_callback 3-19
  constraint_grammar 3-15
  destroy 3-18
  detach_callback 3-19
  get_all_constraints 3-17
  get_callbacks 3-19
  get_constraints 3-17
  match 3-18
  match_structured 3-18
  match_typed 3-18
  modify_constraints 3-16
  remove_all_constraints 3-17
FilterAdmin Interface
  add_filter 3-27
  get_all_filters 3-27
  get_filter 3-27
  remove_all_filters 3-28
  remove_filter 3-27
FilterFactory Interface
  create_filter 3-26
  create_mapping_filter 3-26
Filtering Typed Events 2-52

G
Generated IDL A-4
get_qos 3-8
get_qos operation 2-43

I
Intended Applications 2-30
Interoperability
  Issues 2-56

M
Mapping Filter Interface
  add_mapping_constraints 3-22
  constraint_grammar 3-21
  default_value 3-22
  destroy 3-24
  get_all_mapping_constraints 3-24
  get_mapping_constraints 3-24
  match 3-24
  match_structured 3-25
  match_typed 3-25
  modify_mapping_constraints 3-23
  remove_all_mapping_constraints 3-24
  value_type 3-21
Mapping Filter Objects 2-21
MODL Model A-3

N
Name-Value Pairs 2-28
New Standard Exception C-1
Notification Service Constraints 2-31
Notification Service Event Channel 2-5
Notification Service Event Channel Factory 2-4
Notification Service Style Admin Objects 2-6
Notification Service Style Proxy Interfaces 2-7
NotifyPublish Interface
  offer_change 3-31
NotifySubscribe Interface
  subscription_change 3-31

O
Object Management Group iii
  address of iv

P
Positional Notation 2-30
ProxyConsumer Interface
  MyAdmin 3-51
  MyType 3-51
  obtain_subscription_types 3-51
  validate_event_qos 3-52
ProxyPullConsumer Interface

Index

connect_any_pull_supplier 3-62
resume_connection 3-63
suspend_connection 3-62
ProxyPullSupplier Interface
connect_any_pull_consumer 3-59
ProxyPushConsumer Interface
connect_any_push_supplier 3-56
ProxyPushSupplier Interface
connect_any_push_consumer 3-67
connect_structured_push_consumer 3-69
resume_connection 3-68, 3-69
suspend_connection 3-67, 3-69
ProxySupplier Interface
lifetime_filter 3-54
MyAdmin 3-53
MyType 3-53
obtain_offered_types 3-54
priority_filter 3-54
validate_event_qos 3-55
PullConsumer Interface 3-32
PullSupplier Interface 3-32
PushConsumer Interface 3-31
PushSupplier Interface 3-32

Q
Qos
Property Representation 2-35
Setting 2-35
QoS and Administrative Constant Declarations 3-8
Qos Model
Components 2-35
Qos Properties
Earliest Delivery Time 2-39
Expiry times 2-39
Maximum Events Per Consumer 2-40
Priority 2-39
Reliability 2-37
QoSAdmin Interface 3-8

R
Resolving Initial References C-2
RFP Requirement Not Addressed C-3

S
SequenceProxyPullConsumer Inteface
resume_connection 3-66
SequenceProxyPullConsumer Interface
connect_sequence_pull_supplier 3-65
suspend_connection 3-66
SequenceProxyPullSupplier Interface
connect_sequence_pull_consumer 3-61
SequenceProxyPushConsumer Interface
connect_sequence_push_supplier 3-58
SequenceProxySupplier Interface
connect_sequence_push_consumer 3-70
destroy 3-75
get_proxy_supplier 3-74
lifetime_filter 3-74
MyChannel 3-73
MyID 3-73
MyOperator 3-73
obtain_notification_pull_supplier 3-74
obtain_notification_push_supplier 3-75
priority_filter 3-73
pull_suppliers 3-74
push_suppliers 3-74
resume_connection 3-71
suspend_connection 3-71
SequencePullConsumer Interface
disconnect_sequence_pull_consumer 3-38
SequencePullSupplier Interface
disconnect_sequence_pull_supplier 3-41
pull_structured_events 3-39
try_pull_structured_events 3-40
SequencePushConsumer Interface
disconnect_sequence_push_consumer 3-38
push_structured_events 3-37
SequencePushSupplier Interface
disconnect_sequence_push_supplier 3-41
set_qos 3-8
set_qos operation 2-42
Sharing Subscriptions
Offer 2-49
Sharing Subscriptions Between Channels and Clients 2-49
Subscription Change 2-50
Special Event Types 2-51
Structured Events 2-12
StructuredEvent Data Structure
Body of a Structured Event 3-7
Fixed Header 3-6
Variable Header 3-6
StructuredProxyPullConsumer Interface
connect_structured_pull_supplier 3-64
resume_connection 3-64
suspend_connection 3-64
StructuredProxyPullSupplier Interface
connect_structured_pull_consumer 3-60
StructuredProxyPushConsumer Interface
connect_structured_push_supplier 3-57
StructuredPullConsumer Interface
disconnect_structured_pull_consumer 3-34
StructuredPullSupplier Interface
disconnect_structured_pull_supplier 3-36
pull_structured_event 3-34
try_pull_structured_event 3-35
StructuredPushConsumer Interface
disconnect_structured_push_consumer 3-33
push_structured_event 3-33
StructuredPushSupplier Interface
disconnect_structured_push_supplier 3-36
SupplierAdmin Interface
destroy 3-79
get_proxy_consumer 3-78
MyChannel 3-77
MyID 3-77
MyOperator 3-77
obtain_notification_pull_consumer 3-78
obtain_notification_push_consumer 3-78
pull_consumers 3-77
push_consumers 3-78

T
Trader Constraint Language 2-23
Trader Constraint Language BNF 2-32
TransactionalObject interface 2-12
TypedConsumerAdmin Interface
obtain_typed_notification_pull_supplier 3-98
obtain_typed_notification_push_supplier 3-98
TypedEventChannel Interface
default_consumer_admin 3-103
default_filter_factory 3-104
default_supplier_admin 3-103
get_all_consumeradmins 3-105
get_all_supplieradmins 3-105
get_consumeradmin 3-104
get_supplieradmin 3-105
MyFactory 3-103
new_for_notification_consumers 3-104
new_for_typed_notification_suppliers 3-104
TypedEventChannelFactory Interface
create_typed_channel 3-106
get_all_typed_channels 3-106
get_typed_event_channel 3-106
TypedProxyPullConsumer Interface
connect_typed_pull_supplier 3-93
resume_connection 3-94
suspend_connection 3-93
TypedProxyPullSupplier Interface
connect_typed_pull_consumer 3-91
TypedProxyPushConsumer Interface
conect_typed_push_supplier 3-90
TypedProxyPushSupplier Interface
connect_typed_push_consumer 3-95
resume_connection 3-96
suspend_connection 3-95
TypedPullSupplier Interface 3-85
TypedPushConsumer Interface 3-84
TypedSupplierAdmin Interface
obtain_typed_notification_pull_consumer 3-101
obtain_typed_notification_push_consumer 3-101

U
UnsupportedQoS user exception 2-43

V
validate_event_qos operation 2-43
validate_qos 3-8
validate_qos operation 2-43