Management of Event Domains

Revised Submission =

Submitted By

Alcatel

FUJITSU LIMITED

International Business Machines Corporation

NEC Corporation

Nippon Telegraph and Telephone (NTT) Corporation

Supported By

IONA Technologies, Plc.

January 6, 2000

OMG TC Document telecom/2000-01-01

Submission Contact Points

Michel Ruffin
Alcatel Corporate Research Center[1]
Route de Nozay
91461 Marcoussis cedex, France
telephone: +33-(0)1-69-63-13-57
facsimile: +33-(0)1-69-63-17-89
email: michel.ruffin@alcatel.fr

Masayoshi Shimamura
FUJITSU LIMITED
Nikko Fudousan Building, 15-16, Shinyokohama 2-chome
Kohoku-ku, Yokohama 222, Japan
telephone: +81-45-476-4591
facsimile: +81-45-476-4726
email: shima@rp.open.cs.fujitsu.co.jp

David Chang
International Business Machines
11400 Burnet Road
Austin, TX 78758
telephone: +1-512-838-0559
facsimile: +1-512-838-1032
email: dchang@austin.ibm.com

Michael J. Greenberg
NEC Corporation
305 Foster Street
Littleton, MA 01460-2004
telephone: +1-978-742-8127
facsimile: +1-978-742-8557
e-mail: mjg@nectech.com

Hiroshi Shibata
NTT Corporation
Network Innovation Laboratories
3-9-11 Midori-chou Musashino-shi
Tokyo 180-8585, Japan
telephone: +81-422-59-3954
facsimile: +81-422-59-4810
email: shiba@ma.onlab.ntt.co.jp

Martin Chapman
IONA Technologies
Shelbourne Road
Dublin 4, Ireland
telephone: +353-1-662-5255
facsimile: +353-1-662-5244
email: mchapman@iona.com

[1] Alcatel's contribution to this submission has been partly funded by the European ACTS project RETINA (AC0048).

Table of Contents

1. Overview

This document specifies an architecture and interfaces for managing *event domains*. As used throughout this document, an event domain is a set of one or more *event channels* that are grouped together for the purposes of management, and/or for providing enhanced capabilities to the clients of those channels such as improved scalability. The event channels that are managed by an event domain support one of the channel interfaces defined in the OMG Notification Service specification. Following the structure of the OMG Notification Service, a generic domain interface is defined for managing generic, untyped channels. Another domain interface is defined that can manage both untyped and typed channels. Additionally, a specialized domain is defined that can manage both channels and logs as defined by the OMG Telecom Log Service specification. Note that event domains may contain one or more disjoint topologies of interconnected channels, and possibly one or more channels that are not connected to any other channels.

The main features of the event domains defined in this specification are as follows:

- They extend all of the features of the OMG Notification Service to the *domain* level, which allows managing a group of channels as a related entity. Client applications developed to the Notification Service interfaces can use the channels within a domain unchanged.
- They provide an interface for readily setting up and managing connections between event channels.
- Even when connections between event channels are changed during runtime, it is still possible to acquire the EventType information offered by all upstream channels, and the EventType information subscribed to by all downstream channels.
- They can detect when a new connection between channels leads to the creation of cycle or diamond configurations.
- They provide clients with interfaces to connect themselves to the event domain.

Section 2 of this document provides a detailed description of the architecture underlying this specification. The IDLs defined by this specification are listed and explained in sections 3 through 5.

2. Architectural Features

The architectural features supported by this specification are described in this chapter. After looking at the overall structure, the following items are explained.

- Connections between event channels,
- Event forwarding within a channel topology,
- Sharing of event offer and subscription information in the event domain,
- Topology management,
- Connection management of clients to the domain.

2.1. Event Domain Architecture Overview

The service architecture is outlined in this section. An event domain defined in this specification extends the features supported by event channels as defined by the Notification Service to groups of inter-related channels. Interfaces are defined that enable the creation and management of these groups of inter-related channels, and for forming and managing connections between channels within a domain.

The event domains defined by this specification are designed to manage a group of event channels that support the interfaces defined by the OMG Notification Service specification. A clear goal of this specification is to define the capability to manage an inter-related group of channels that can be created via an implementation of the OMG Notification Service. This will enable implementations of this specification that can manage channels that are created using existing implementations of the OMG Notification Service. While this point may seem insignificant, it is subtly important due to the reality that implementations of the OMG Notification Service are already commercially available, and end-users require tools that enable the management of groups of inter-related channels created using their existing products. Another way to state this goal is that this specification is defined to ensure that new Notification Service implementations are not required in order to support the channels necessary for management by event domains.

The figure below shows the relationships of the interfaces defined in this specification, and those of the OMG Notification Service. In this figure, the IDL module names are abbreviated as follows:

- NC: CosNotifyComm
- NCA: CosNotifyChannelAdmin
- EDA: CosEventDomainAdmin

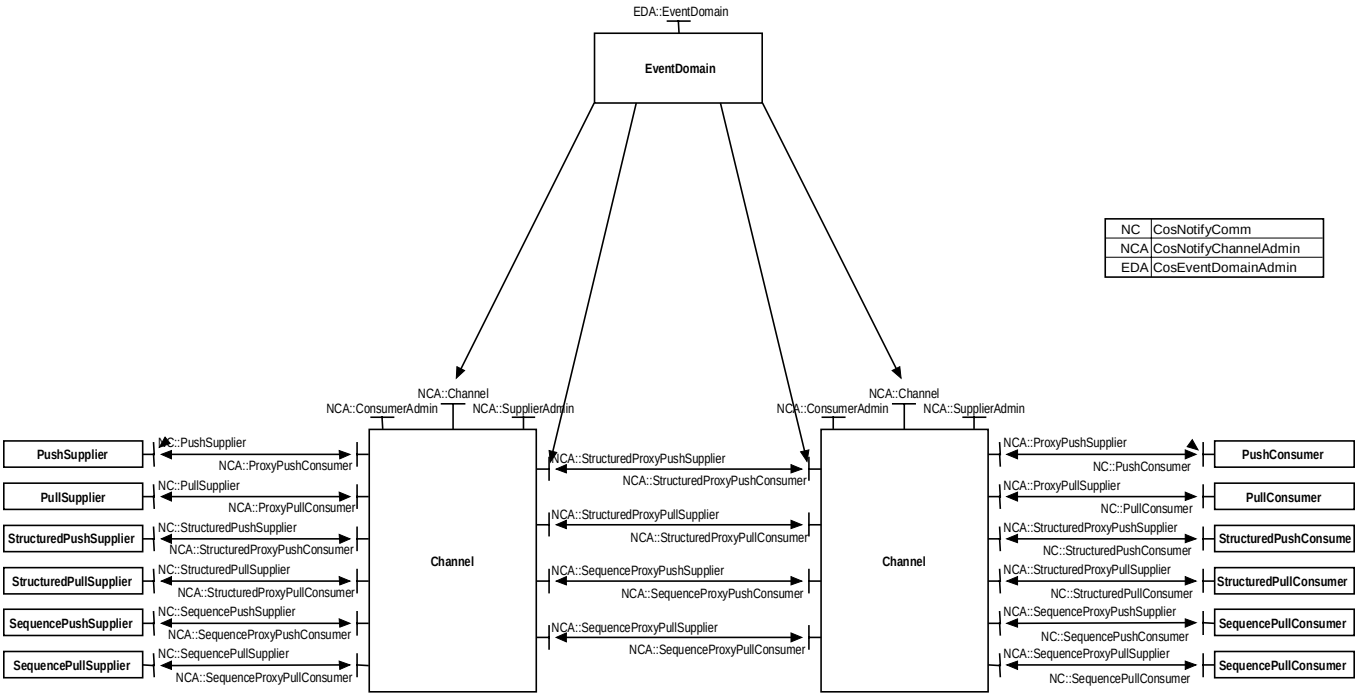

Figure 2-1: General Architecture of the Event Domain.

Figure 2-1 represents the general architecture of the Event Domain. It shows the different relationships it has with the elements it manages and there are between those elements. In this figure, the Event Domain manages classic Notification Service channels, which are connected to be classic Notification Service clients (any, structured, sequence push/pull suppliers and consumers). Note that this figure shows only the version of an event domain that supports untyped Notification Service channels. A subtype of this domain is also defined that supports both untyped and typed Notification Service channels, and another that supports logs as defined by the Telecom Log Service as well.

Note that a given channel within an event domain may be connected to any combination of other channels in the same domain, endpoint suppliers, and endpoint consumers. Channels are interconnected using Notification Service style proxy interfaces, while clients connect to channels using the interfaces and mechanisms defined by the Notification Service. Alternatively, clients can use the operations supported by the EventDomain to request connection to the default channel within the domain, or a specific channel identified by the client.

Event domains have a notion of membership. Before a domain can be used to form a connection between two channels, each channel must be added as a member of the domain. Channels may be members of multiple domains. Thus, event domains assign identifiers to each channel that is added as a member, which uniquely identifies that channel within the given domain.

2.2. Connection Between Event Channels

In the OMG Notification Service, endpoint suppliers connect to proxy consumers provided on event channels, and endpoint consumers connect to proxy suppliers. Due to the facts that proxy consumers inherit the endpoint consumer interface, and proxy suppliers inherit the endpoint supplier interface, even with the OMG Notification Service by itself it is possible to connect two channels as suppliers and consumers of each other's events. This is achieved by creating a proxy supplier object within one channel (e.g., Channel A), and a proxy consumer object within another channel (e.g., Channel B), connecting the proxy supplier of Channel A as the supplier to Channel B's proxy consumer, and the proxy consumer of Channel B as the consumer of Channel A's proxy supplier.

While it is possible to achieve such connections using the interfaces defined in the OMG Notification Service without any extensions, doing so is extremely cumbersome. Essential, the programmatic steps required involve:

- Getting a reference to the **SupplierAdmin** instance from one **EventChannel**.
- Creating a **ProxyConsumer** instance using this **SupplierAdmin**.
- Getting a reference to the **ConsumerAdmin** instance from another **EventChannel**.
- Creating a **ProxySupplier** instance using this **ConsumerAdmin**.
- Connect to the **ProxyConsumer** using the obtained **ProxySupplier** instance reference as input parameter.
- Similarly, connect to the **ProxySupplier** using the **ProxyConsumer** instance reference as input parameter.

An **EventDomain** interface has been defined for handling the above steps in one operation. In addition, the **EventDomain** interface supports operations for managing the proxy instances and connections created as a result of these steps, and for replying to queries about their status.

Instances supporting the **EventDomain** interface manage the event channels they logically contain, and any connections between those channels. The interface supports operations for creating channels and registering them with the domain. Upon registering a channel within its domain, the event domain assigns a **MemberID** to the channel that is unique among all channels within the same domain. Thereafter, this ID is used when establishing, checking or removing connections between channels within the domain. Connections are defined using a **Connection** data structure, which includes the fields displayed in the figure below. Details of the procedures involved are given in section 3.

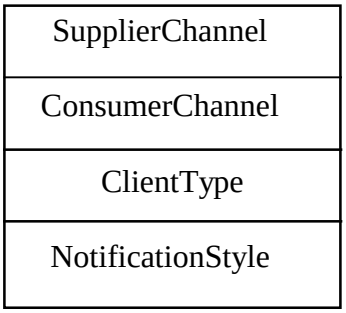

Figure 2-1: The structure of a Connection.

In this figure, the *SupplierChannel* and *ConsumerChannel* fields are the MemberIDs of the two channels involved in the connection. Note that an event domain can only be used to connect two channels that have both been added as members of the target domain. The *ClientType* field indicates the form of events the two channels will communicate with over the connection (i.e. Any, Structured, Sequence, or Typed), and the *NotificationStyle* field indicates whether the two channels will communicate using push or pull style.

2.3.Event Forwarding

Using the interfaces supported by the **EventDomain** interface, it is easy to create topologies of interconnected channels. The topologies can be of arbitrary complexity, including topologies that contain *cycles* in the directed graph of inter-connected channels, or *diamond* shapes in the graph of interconnected channels meaning the same event may reach a point in the graph by more than one route.

This specification defines mechanisms that enable detection of cycles or diamonds in the graph of interconnected channels as connections are established between channels using operations supported by the **EventDomain** interface. Quality of service properties can be set upon an event domain that control whether or not cycles and/or diamond shaped topologies are allowed within the domain. For instance, the property *CycleDetection* can be set to either *AuthorizeCycles* or *ForbidCycles*, indicating whether or not the operations that establish connections between channels within the domain should raise an exception if establishing a particular connection will cause the introduction of a cycle into the topology of channels to

which the connection is being added.

Note that by allowing clients to turn on or off cycle and diamond detection in this fashion, whether or not cyclical and/or diamond topologies are allowed within an event domain is controllable by end-users. This allows for the possibility that there may in fact be scenarios in which end-users really do want to create such topologies, and also allows for the possibility that the administrator of an event domain may want to prevent end-users from creating such topologies. End-users who do choose to create topologies that contain cycles should be aware of the fact that unless they set timeout on events, events that are not filtered will loop endlessly through the topology. Likewise, end-users who choose to create topologies that contain diamonds should be aware of the fact that consumers may receive the same event multiple times (the number of times that is equal to the number of paths by which the event may arrive at the consumer).

2.4.Sharing Event Type Offer and Subscription Information in the Event Domain

This section describes how event type offer and subscription information is shared, managed and referenced across event channels in an event domain.

The *offer_change* mechanism defined by the Notification Service is such that when an end-point supplier invokes *offer_change* on its proxy consumer to inform the channel to which it is connected of a change in the set of event types it will potentially be supplying, the channel is responsible for sharing this information with all of its consumers. This is done by the channel invoking *offer_change* on all consumers to which it was connected, assuming the supplier's *offer_change* resulted in a change to the union of all event types that the channel can receive (which is not necessarily the case). Note that one or more of these "consumers" upon which a channel is invoking *offer_change* could actually be the proxy consumer(s) of another channel. Thus, in a topology of interconnected channels, these *offer_change* invocations can potentially be propagated throughout the topology.

A similar scenario exists in the case of *subscription_change*. A channel is responsible for invoking the *subscription_change* operation on all of its suppliers whenever the change to a client's subscriptions (due to filters being added or removed, or filter constraints being added, removed, or modified) results in a change to the set of event types being subscribed to by consumers of the channel's events. As was the case for *offer_change* invocations, in a topology of interconnected channels these *subscription_change* invocations can potentially be propagated throughout the topology.

Note that this propagation of offer and subscription information is subject to the same potential problems that arise when events are propagated throughout a topology of interconnected channels. If the topology involved contains cycles or diamonds, offer and subscription information can potentially be propagated endlessly, or at least the same information may be forwarded to the same points in the topology multiple times. End-users who choose to setup channel topologies that contain cycles or diamonds should be aware of this situation, and should likely consider turning off offer and subscription information propagation (this can be done using mechanisms supported by the OMG Notification Service).

End-point suppliers of events to a channel that is part of a channel topology managed by an event domain can obtain information about all types of events being subscribed to by channels anywhere downstream in the channel topology by invoking *obtain_subscription_types* on the proxy consumers they to which they are connected. Whether a given channel relies on its internal database of subscription types obtained from previous invocations of *obtain_subscription_types* that it has propagated to all channels to which it is connected as a supplier and information passed to it from its consumer channels from previous propagations of *subscription_change*, or if it propagates every invocation of *obtain_subscription_types* to its consumer channels in order to obtain updated subscription information before replying to the end-point supplier's request is an implementation decision. Figure 2-3 below depicts one possible implementation of event subscription information propagation within a channel topology.

Similarly, end-point consumers of events supplied by a channel that is part of a channel topology managed by an event domain can obtain information about all types of events being offered to it by channels anywhere downstream in the channel topology by invoking *obtain_offered_types* on the proxy suppliers they to which they are connected. Once again, whether a given channel relies on its internal database of offer types obtained from previous invocations of *obtain_offered_types* that it has propagated to all channels to which it is connected as a consumer and information passed to it from its consumer channels from previous

propagations of *offer_change*, or if it propagates every invocation of *obtain_offered_types* to its supplier channels in order to obtain updated offer information before replying to the end-point supplier's request is an implementation decision.

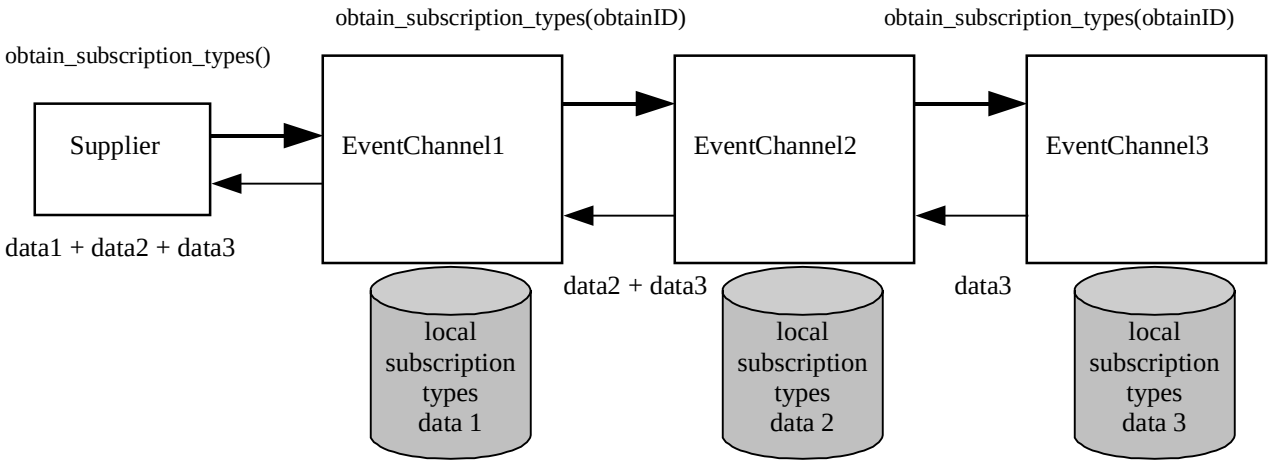

Figure 2-1: Sharing Subscription Information in the Event Domain.

2.5. Topology management of an Event Domain

The topology management functionality of an event domain provides domain clients with cycle and diamond configuration detection while processing connection of channels requests. These functionnalities are enabled setting the **CycleDetection** and **DiamondDetection** QoS at the Event Domain level.

2.6. Connection of Clients to the Event Domain

The connection of clients to the event domain functionality allows clients to either:

- Connect to a particular channel, given its unique channel identifier in the scope of the event domain.
- Connect to the event domain itself. Such a connection is redirected toward a default channel selected at the event domain level.

2.7. Quality of Service properties

The standard Notification Service QoS administrative interface is inherited by the interfaces of this specification that need to support QoS settings, which provides users with a simple and already known way of setting QoS.

The table below lists levels at which additional QoS properties defined for the Management of Event Domains are supported.

Property	Per-Message	Per-Proxy	Per-Admin	Per-Channel	Per-Domain
CycleDetection					X

DiamondDetection	X

Table 2-1: Additional QoS properties supported levels.

The following section defines and describes the interfaces supported by the generic event domain. Section 4 provides definition and description of the interfaces supported by the typed event domain, while section 5 provides similar coverage of the interfaces supported by event domains that also support logs as members.

3. *Event Domain Interfaces*

This section describes the semantic behavior of the interfaces that comprise the Event Domain. The Event Domain IDL is defined within the **CosEventDomainAdmin** 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. A more detailed explanation of the behavior of each functional aspect of the Event Domain is given in section 2 of this document.

```
#ifndef _COS_EVENT_DOMAIN_ADMIN_IDL_
#define _COS_EVENT_DOMAIN_ADMIN_IDL_

// Event Domain Interface
#include "CosNotification.idl"
#include "CosEventComm.idl"
#include "CosNotifyComm.idl"
#include "CosNotifyChannelAdmin.idl"


module CosEventDomainAdmin {

  // The following constant declarations define the Event Domain
  // QoS property names and the associated values each property can
  // take on. The name/value pairs for each Event Domain property
  // are grouped, beginning with a string constant defined for the
  // property name, followed by the values the property can take on.

  const string CycleDetection = "CycleDetection";
  const short AuthorizeCycles = 0; // Default value
  const short ForbidCycles = 1;

  const string DiamondDetection = "DiamondDetection";
  const short AuthorizeDiamonds = 0; // Default value
  const short ForbidDiamonds = 1;


  // The following enum declaration defines the types that a channel
  // can be of. It is used to specify channel types while externalizing
  // and instantiating topologies.
  enum ChannelType
  {
    CHANNEL,
    TYPED_CHANNEL,
    LOG_CHANNEL,
    TYPED_LOG_CHANNEL
  };

  enum NotificationStyle {
    Push,
    Pull
  };

  typedef long MemberID;
  typedef sequence <MemberID> MemberIDSeq;
  typedef long ConnectionID;
  typedef sequence <ConnectionID> ConnectionIDSeq;

  struct Connection {
    MemberID supplier_id;
    MemberID consumer_id;
    CosNotifyChannelAdmin::ClientType ctype;
    NotificationStyle notification_style;
  };

  typedef MemberIDSeq Route;
  typedef sequence<Route> RouteSeq;

  typedef Route Cycle;
  typedef sequence<Cycle> CycleSeq;

  typedef RouteSeq Diamond;
  typedef sequence<Diamond> DiamondSeq;

  exception CycleCreationForbidden
  {
    Cycle cyc;
  };

  exception DiamondCreationForbidden
  {
    Diamond diam;
```

```
};

// Forward declarations
interface ConsumerAdmin;
interface SupplierAdmin;


typedef long DomainID;
typedef sequence <DomainID> DomainIDSeq;
typedef long ItemID;


// EventDomain administrates EventChannels that reside in the same administrative domain
exception ConnectionNotFound {};
exception AlreadyExists {};


interface EventDomain :
  CosNotification::QoSAdmin ,
                  CosNotification::AdminPropertiesAdmin {

  MemberID add_channel (
                               in CosNotifyChannelAdmin::EventChannel channel );

  MemberIDSeq get_all_channels ();

  CosNotifyChannelAdmin::EventChannel get_channel (
                                    in MemberID channel )
    raises ( CosNotifyChannelAdmin::ChannelNotFound );

  void  remove_channel (
                               in MemberID channel)
    raises (CosNotifyChannelAdmin::ChannelNotFound);

  ConnectionID add_connection (
                              in Connection connection)
    raises (CosNotifyChannelAdmin::ChannelNotFound,
           CosEventChannelAdmin::TypeError,
            AlreadyExists,
            CycleCreationForbidden,
            DiamondCreationForbidden);

  ConnectionIDSeq get_all_connections ();

  Connection get_connection (
                             in ConnectionID connection )
    raises ( ConnectionNotFound );

  void remove_connection (
                          in ConnectionID connection )
    raises ( ConnectionNotFound );

  CosNotifyChannelAdmin::ChannelIDSeq get_offer_channels (
                                                    in MemberID channel )
    raises ( CosNotifyChannelAdmin::ChannelNotFound );

  CosNotifyChannelAdmin::ChannelIDSeq get_subscription_channels (
                                                             in MemberID channel )
    raises ( CosNotifyChannelAdmin::ChannelNotFound );

  void destroy();

  // Cycle and diamond configurations listing
  CycleSeq get_cycles();

  DiamondSeq get_diamonds();

  // Connection of clients to the domain
  // - using no specific information
  //   - for any clients
  void set_default_consumer_channel(in MemberID channel)
    raises (CosNotifyChannelAdmin::ChannelNotFound);

  void set_default_supplier_channel(in MemberID channel)
    raises (CosNotifyChannelAdmin::ChannelNotFound);

  CosNotifyChannelAdmin::ProxyPushSupplier
  connect_push_consumer(in CosEventComm::PushConsumer client)
    raises (CosNotifyChannelAdmin::ChannelNotFound);

  CosNotifyChannelAdmin::ProxyPullSupplier
  connect_pull_consumer(in CosEventComm::PullConsumer client)
    raises (CosNotifyChannelAdmin::ChannelNotFound);
```

```
    CosNotifyChannelAdmin::ProxyPushConsumer
    connect_push_supplier(in CosEventComm::PushSupplier client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::ProxyPullConsumer
    connect_pull_supplier(in CosEventComm::PullSupplier client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    //   - for structured clients
    CosNotifyChannelAdmin::StructuredProxyPushSupplier
    connect_structured_push_consumer(in CosNotifyComm::StructuredPushConsumer client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::StructuredProxyPullSupplier
    connect_structured_pull_consumer(in CosNotifyComm::StructuredPullConsumer client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::StructuredProxyPushConsumer
    connect_structured_push_supplier(in CosNotifyComm::StructuredPushSupplier client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::StructuredProxyPullConsumer
    connect_structured_pull_supplier(in CosNotifyComm::StructuredPullSupplier client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    //   - for sequence clients
    CosNotifyChannelAdmin::SequenceProxyPushSupplier
    connect_sequence_push_consumer(in CosNotifyComm::SequencePushConsumer client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::SequenceProxyPullSupplier
    connect_sequence_pull_consumer(in CosNotifyComm::SequencePullConsumer client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::SequenceProxyPushConsumer
    connect_sequence_push_supplier(in CosNotifyComm::SequencePushSupplier client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::SequenceProxyPullConsumer
    connect_sequence_pull_supplier(in CosNotifyComm::SequencePullSupplier client)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    // - using a channel id
    //   - for any clients
    CosNotifyChannelAdmin::ProxyPushSupplier
    connect_push_consumer_with_id(in CosEventComm::PushConsumer client,
                                  in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::ProxyPullSupplier
    connect_pull_consumer_with_id(in CosEventComm::PullConsumer client,
                                  in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::ProxyPushConsumer
    connect_push_supplier_with_id(in CosEventComm::PushSupplier client,
                                  in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::ProxyPullConsumer
    connect_pull_supplier_with_id(in CosEventComm::PullSupplier client,
                                  in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    //   - for structured clients
    CosNotifyChannelAdmin::StructuredProxyPushSupplier
    connect_structured_push_consumer_with_id(in CosNotifyComm::StructuredPushConsumer client,
                                             in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::StructuredProxyPullSupplier
    connect_structured_pull_consumer_with_id(in CosNotifyComm::StructuredPullConsumer client,
                                             in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::StructuredProxyPushConsumer
    connect_structured_push_supplier_with_id(in CosNotifyComm::StructuredPushSupplier client,
                                             in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::StructuredProxyPullConsumer
    connect_structured_pull_supplier_with_id(in CosNotifyComm::StructuredPullSupplier client,
                                             in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);
```

```
    //   - for sequence clients
    CosNotifyChannelAdmin::SequenceProxyPushSupplier
    connect_sequence_push_consumer_with_id(in CosNotifyComm::SequencePushConsumer client,
                                           in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::SequenceProxyPullSupplier
    connect_sequence_pull_consumer_with_id(in CosNotifyComm::SequencePullConsumer client,
                                           in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::SequenceProxyPushConsumer
    connect_sequence_push_supplier_with_id(in CosNotifyComm::SequencePushSupplier client,
                                           in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    CosNotifyChannelAdmin::SequenceProxyPullConsumer
    connect_sequence_pull_supplier_with_id(in CosNotifyComm::SequencePullSupplier client,
                                           in MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);
  };


  exception DomainNotFound {};


  interface EventDomainFactory {

    EventDomain create_event_domain(
                                    in CosNotification::QoSProperties initialQoS ,
                                    in CosNotification::AdminProperties initialAdmin,
                                    out DomainID id )
      raises ( CosNotification::UnsupportedQoS,
               CosNotification::UnsupportedAdmin );

    DomainIDSeq get_all_domains ();

    EventDomain get_event_domain (
                                  in DomainID id )
      raises ( DomainNotFound );
  };
};

#endif // _COS_EVENT_DOMAIN_ADMIN_IDL_
```

Table 3-1: The CosEventDomainAdmin Module.

3.1. The EventDomain Interface

The **EventDomain** interface encapsulates all behaviors supported by event domain objects. Event domain objects are capable of managing one or more topologies of interconnected untyped Notification Service channels.

The **EventDomain** interface inherits from the **QoSAdmin** and **AdminPropertiesAdmin** interfaces defined in the **CosNotifyChannelAdmin** module. Inheritance of these interfaces enables event domains to be configured to support certain QoS and Admin property settings at the domain level. Following the conventions established by the Notification Service, settings for properties at the domain level that are also settable at the channel level would become the default settings for these properties for all channels within the domain. However, awareness of the new, higher level in the QoS hierarchy could only be supported by new implementations of the Notification Service, since existing implementations are not aware of the domain concept.

The **EventDomain** interface defines operations for adding new channels to a domain, for retrieving a particular channel within the domain by unique ID, and for retrieving a list of all channels that exist within the domain, and for removing a particular channel from a domain.

The **EventDomain** interface also defines operations for forming connections between two channels within its domain, for retrieving a particular connection by unique ID, for retrieving a list of all connections that exist within the domain, and for removing a particular connection from the domain.

In addition, the **EventDomain** interface supports an operation that, given the unique ID of a channel within

the target event domain, will return the list of all supplier channels that are upstream within the same topology of interconnected channels as the input channel. Likewise, the **EventDomain** interface supports an operation that, given the unique ID of a channel within the target event domain, will return the list of all consumer channels that are downstream within the same topology of interconnected channels as the input channel.

A set of operations is also provided to allow the connection of clients to the Event Domain, either using a default channel or specifying the identifier of a channel.

Lastly, the **EventDomain** interface supports an operation that can be invoked to destroy the target domain.

3.1.1. add _channel

The *add_ channel* operation adds a channel to the target domain. This operation takes the reference of a channel as input, and returns an identifier for the channel that is unique among all channels contained within the domain. Note that this identifier specifically represents the channel's membership within the domain, and is not the same as the identifier assigned to the channel by the factory that created it. Having the domain assign its own identifiers to member channels enables channels created by different factories to be added to the same domain, still guaranteeing uniqueness among the identifiers assigned to channels within a domain. Thus, a particular channel may belong to multiple domains, and have a different identifier assigned to it within each domain to which it belongs.

3.1.2. get_all_channels

The *get_all_channels* operation returns a sequence of all of the unique identifiers corresponding to all channels that currently exist within the target domain. Note these are the identifiers that were assigned to the member channels when they were added to the domain.

3.1.3. get_channel

The *get_channel* operation accepts as input a numeric value that is supposed to be the unique identifier of a channel that currently exists within the target domain. 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 channel corresponding to the input identifier. Note the identifier supplied as input should be the identifier assigned to the channel when it was added to the domain.

3.1.4. remove_channel

The *remove_channel* operation removes a channel from the target domain. The operation takes as input the unique identifier of a channel within the domain. If the supplied input parameter is not the unique identifier of a channel within the domain, the **ChannelNotFound** exception is raised. Otherwise, the channel corresponding to the supplied reference is removed from the target domain. Note the identifier supplied as input should be the identifier assigned to the channel when it was added to the domain.

3.1.5. add_connection

The *add_connection* operation is invoked to cause a connection to be formed between two channels in the target domain.

The operation takes as input a data structure that describes the connection to be formed. This structure contains the ID of the channel that is intended to be the supplier in the relationship between the two channels, and the ID of the channel that is intended to be the consumer in the relationship between the two channels. If either of these IDs does not correspond to the ID of a channel that currently exists within the target domain, the **ChannelNotFound** exception is raised. The input structure also contains a flag that indicates the form of events that will be communicated between the two channels over the connection, and another flag that indicates whether push or pull style communication should be used.

If the two channels indicated by the ID fields of the connection structure exist within the target domain, but a connection between them already exists in the same direction as indicated by their place in the structure (i.e., the existing connection is such that the same channel that is the supplier in the relationship would also be the supplier as a result of forming the new connection), the **AlreadyExists** exception is raised.

If the **CycleDetection** QoS property of the target event domain is set to the value of **ForbidCycle,** and the creation of the requested connection would result in a cycle being created within the topology of channels to which the connection is being added, then the **CycleCreationForbidden** exception is raised. This exception contains as data the sequence of channel member identifiers that would have formed the cycle.

Likewise, if the **DiamondDetection** QoS property of the target event domain is set to the value of **ForbidDiamond,** and the creation of the requested connection would result in a diamond being created within the topology of channels to which the connection is being added, then the **DiamondCreationForbidden** exception is raised. This exception contains as data a sequence of conflicting paths, each path being a sequence of channel member identifiers.

Otherwise, the appropriate operations are invoked by the target event domain upon the two channels involved in the connection in order to form the desired connection. Upon successfully doing this, the operation assigns a numeric identifier to correspond to the connection that is unique among all connection identifiers in the domain, and returns this identifier as the result of the operation.

3.1.6. get_all_connections

The *get_all_connections* operation returns a sequence of all of the unique numeric identifiers corresponding to all connections that currently exist within the target domain.

3.1.7. get_connection

The *get_connection* operation accepts as input a numeric value that is supposed to be the unique identifier of connection that currently exists within the target domain. If this input value does not correspond to such a unique identifier, the **ConnectionNotFound** exception is raised. Otherwise, the operation returns a data structure that describes the connection corresponding to the input ID.

3.1.8. remove_connection

The *remove_connection* operation is invoked to remove an existing connection between two channels in the target domain. The operation takes as input the unique identifier of a connection that exists within the domain. If the input parameter does not correspond to an existing connection within the domain, the **ConnectionNotFound** exception is raised. Otherwise, the necessary operations are invoked upon the target channels to remove the connection described by the corresponding connection structure. Note that this operation does not remove the two channels involved. To remove channels completely, their destroy operation must be invoked.

3.1.9. get_offer_channels

The *get_offer_channels* operation is invoked to obtain a list of all channels that exist upstream in the target domain with respect to a given channel that exists within the domain. The operation takes as input the unique ID of a channel that exists within the domain. If the supplied ID does not correspond to a channel that exists within the domain, the **ChannelNotFound** exception is raised. Otherwise, the sequence of IDs of all channels that exist upstream with respect to the input channel is returned.

3.1.10. get_subscription_channels

The *get_subscription_channels* operation is invoked to obtain a list of all channels that exist downstream in the target domain with respect to a given channel that exists within the domain. The operation takes as input the unique ID of a channel that exists within the domain. If the supplied ID does not correspond to a channel that exists within the domain, the **ChannelNotFound** exception is raised. Otherwise, the sequence of IDs of

all channels that exist downstream with respect to the input channel is returned.

3.1.11. destroy

The destroy operation is invoked to destroy the target domain. If connections between channels within the domain exist that were established by the target domain, these connections will be removed prior to destruction of the target domain.

3.1.12. get_cycles

The *get_cycles* operation is invoked to retrieve a list of cycles that exist within any topology of channels formed by connections that were established between these channels by the target domain. The operation accepts no input parameters, and returns as a result a sequence of cycles, whereas each cycle is itself a sequence of member identifiers that identify channels within the domain that are involved in the cycle.

3.1.13. get_diamonds

The *get_diamonds* operation is invoked to retrieve a list of diamonds that exist within any topology of channels formed by connections that were established between these channels by the target domain. The operation accepts no input parameters, and returns as a result a sequence of diamonds, whereas each diamond is itself a sequence of member identifiers of channels within the domain that are involved in the diamond.

3.1.14. set_default_consumer_channel

The *set_default_consumer_channel* operation is invoked to specify a particular channel within the target domain as the default channel to which consumers will be connected when they invoke one of the *connect_*_consumer* operations supported by the EventDomain interface, requesting in a single invocation that they be connected to the domain.

The operation accepts as input a number value that should be the unique member ID of one of the channels within the target event domain. If the input value does correspond to the member ID of one of the channels within the target domain, that channel becomes the default channel within the domain for consumer connections. If, however, there is no channel within the domain that has the input value as its member ID, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised.

Note that before this operation is invoked, the domain's default channel for consumers is set to the first channel added to the target domain.

3.1.15. set_default_supplier_channel

The *set_default_supplier_channel* operation is invoked to specify a particular channel within the target domain as the default channel to which suppliers will be connected when they invoke one of the *connect_*_supplier* operations supported by the EventDomain interface, requesting in a single invocation that they be connected to the domain.

The operation accepts as input a number value that should be the unique member ID of one of the channels within the target event domain. If the input value does correspond to the member ID of one of the channels within the target domain, that channel becomes the default channel within the domain for supplier connections. If, however, there is no channel within the domain that has the input value as its member ID, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised.

Note that before this operation is invoked, the domain's default channel for suppliers is set to the first channel added to the target domain.

3.1.16. connect_push_consumer

The *connect_push_consumer* operation is invoked to connect a push style consumer of events in the form of

CORBA::Anys to the event domain. The operation accepts as input the reference to either an Event or Notification Service style push consumer of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PushConsumer** (an Event Service style push consumer), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PushConsumer** interface (i.e., a Notification Service style push consumer).

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for consumers within the domain to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.17. connect_pull_consumer

The *connect_pull_consumer* operation is invoked to connect a pull style consumer of events in the form of CORBA::Anys to the event domain. The operation accepts as input the reference to either an Event or Notification Service style pull consumer of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PullConsumer** (an Event Service style pull consumer), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PullConsumer** interface (i.e., a Notification Service style pull consumer).

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for consumers within the domain to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.18. connect_push_supplier

The *connect_push_supplier* operation is invoked to connect a push style supplier of events in the form of CORBA::Anys to the event domain. The operation accepts as input the reference to either an Event or Notification Service style push supplier of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PushSupplier** (an Event Service style push supplier), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PushSupplier** interface (i.e., a Notification Service style push supplier).

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for suppliers within the domain to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.19. connect_pull_supplier

The *connect_pull_supplier* operation is invoked to connect a pull style supplier of events in the form of CORBA::Anys to the event domain. The operation accepts as input the reference to either an Event or Notification Service style pull supplier of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PullSupplier** (an Event Service style pull supplier), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PullSupplier** interface (i.e., a Notification Service style pull supplier).

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for suppliers within the domain to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's

connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.20. connect_structured_push_consumer

The *connect_structured_push_consumer* operation is invoked to connect a push style consumer of events in the form of structured events to the event domain. The operation accepts as input the reference to a Notification Service style consumer of events in the form of structured events that uses push mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for consumers within the domain to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.21. connect_structured_pull_consumer

The *connect_structured_pull_consumer* operation is invoked to connect a pull style consumer of events in the form of structured events to the event domain. The operation accepts as input the reference to a Notification Service style consumer of events in the form of structured events that uses pull mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for consumers within the domain to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.22. connect_structured_push_supplier

The *connect_structured_push_supplier* operation is invoked to connect a push style supplier of events in the form of structured events to the event domain. The operation accepts as input the reference to a Notification Service style supplier of events in the form of structured events that uses push mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for suppliers within the domain to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.23. connect_structured_pull_supplier

The *connect_structured_pull_supplier* operation is invoked to connect a pull style supplier of events in the form of structured events to the event domain. The operation accepts as input the reference to a Notification Service style supplier of events in the form of structured events that uses pull mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for suppliers within the domain to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.24. connect_sequence_push_consumer

The *connect_sequence_push_consumer* operation is invoked to connect a push style consumer of events in the form of sequence events to the event domain. The operation accepts as input the reference to a Notification Service style consumer of events in the form of sequence events that uses push mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for consumers within the domain to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.25. connect_sequence_pull_consumer

The *connect_sequence_pull_consumer* operation is invoked to connect a pull style consumer of events in the form of sequence events to the event domain. The operation accepts as input the reference to a Notification Service style consumer of events in the form of sequence events that uses pull mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for consumers within the domain to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.26. connect_sequence_push_supplier

The *connect_sequence_push_supplier* operation is invoked to connect a push style supplier of events in the form of sequence events to the event domain. The operation accepts as input the reference to a Notification Service style supplier of events in the form of sequence events that uses push mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for suppliers within the domain to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.27. connect_sequence_pull_supplier

The *connect_sequence_pull_supplier* operation is invoked to connect a pull style supplier of events in the form of sequence events to the event domain. The operation accepts as input the reference to a Notification Service style supplier of events in the form of sequence events that uses pull mode of interaction with its channel.

If the target domain contains no channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the default channel for suppliers within the domain to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.28. connect_push_consumer_with_id

The *connect_push_consumer_with_id* operation is invoked to connect a push style consumer of events in the form of CORBA::Anys to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to either an Event or Notification Service style push consumer of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PushConsumer** (an Event Service style push consumer), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PushConsumer** interface (i.e., a Notification Service style push consumer). The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.29. connect_pull_consumer_with_id

The *connect_pull_consumer_with_id* operation is invoked to connect a pull style consumer of events in the form of CORBA::Anys to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to either an Event or Notification Service style pull consumer of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PullConsumer** (an Event Service style pull consumer), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PullConsumer** interface (i.e., a Notification Service style pull consumer). The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.30. connect_push_supplier_with_id

The *connect_push_supplier_with_id* operation is invoked to connect a push style supplier of events in the form of CORBA::Anys to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to either an Event or Notification Service style push supplier of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PushSupplier** (an Event Service style push supplier), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PushSupplier** interface (i.e., a Notification Service style push supplier). The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.31. connect_pull_supplier_with_id

The *connect_pull_supplier_with_id* operation is invoked to connect a pull style supplier of events in the form of CORBA::Anys to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to either an Event or Notification Service style pull supplier of events in the form of CORBA::Anys. The type of the input parameter is **CosEventComm::PullSupplier** (an Event Service style pull supplier), but due to interface inheritance this could also be a reference to an object supporting the **CosNotifyComm::PullSupplier** interface (i.e., a Notification Service style pull supplier). The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.32. connect_structured_push_consumer_with_id

The *connect_structured_push_consumer_with_id* operation is invoked to connect a push style consumer of events in the form of structured events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style push consumer of events in the form of structured events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.33. connect_structured_pull_consumer_with_id

The *connect_structured_pull_consumer_with_id* operation is invoked to connect a pull style consumer of events in the form of structured events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style pull consumer of events in the form of structured events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.34. connect_structured_push_supplier_with_id

The *connect_structured_push_supplier_with_id* operation is invoked to connect a push style supplier of events in the form of structured events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style push supplier of events in the form of structured events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.35. connect_structured_pull_supplier_with_id

The *connect_structured_pull_supplier_with_id* operation is invoked to connect a pull style supplier of events in the form of structured events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style pull supplier of events in the form of structured events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.36. connect_sequence_push_consumer_with_id

The *connect_sequence_push_consumer_with_id* operation is invoked to connect a push style consumer of events in the form of sequence events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style push consumer of events in the form of sequence events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.37. connect_sequence_pull_consumer_with_id

The *connect_sequence_pull_consumer_with_id* operation is invoked to connect a pull style consumer of events in the form of sequence events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style pull consumer of events in the form of sequence events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input consumer to this channel. The channel's default ConsumerAdmin will be used to create the appropriate proxy supplier instance, and the proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the proxy supplier created on behalf of the client is returned as the result of the operation.

3.1.38. connect_sequence_push_supplier_with_id

The *connect_sequence_push_supplier_with_id* operation is invoked to connect a push style supplier of events in the form of sequence events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style push supplier of events in the form of sequence events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.1.39. connect_sequence_pull_supplier_with_id

The *connect_sequence_pull_supplier_with_id* operation is invoked to connect a pull style supplier of events in the form of sequence events to a specific channel within the event domain. The operation accepts two input parameters. The first is as the reference to a Notification Service style pull supplier of events in the form of sequence events. The second input parameter is an integer value that should correspond to the unique member ID of one of the channels within the target domain.

If the target domain does not contain a channel whose member ID is equivalent the second input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. Otherwise, the event domain proceeds to invoke the appropriate operations upon the channel corresponding to the member ID passed as the second input parameter to connect the input supplier to this channel. The channel's default SupplierAdmin will be used to create the appropriate proxy consumer instance, and the proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the proxy consumer created on behalf of the client is returned as the result of the operation.

3.2.The EventDomainFactory Interface

The **EventDomainFactory** interface defines operations for creating and managing event domains. It supports a routine that creates new instances of event domains and assigns unique numeric identifiers to them. In addition, the **EventDomainFactory** interface supports a routine which can return the unique identifiers assigned to all event domains created by a given instance of **EventDomainFactory**, and another routine which, given the unique identifier of an event domain created by a target **EventDomainFactory** instance, returns the object reference of that event domain.

3.2.1. create_event_domain

The *create_event_domain* operation is invoked to create a new instance of event domain. This operation accepts two input parameters. The first input parameter is a list of name-value pairs that specify the initial QoS property settings for the new event domain. The second input parameter is a list of name-value pairs that specify the initial administrative property settings for the new event domain. If no implementation of the **EventDomain** 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 which could be satisfied.

Likewise, if no implementation of the **EventDomain** 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.

If neither of these exceptions is raised, the *create_event_domain* operation will return a reference to an event domain. In addition, the operation assigns to this new event domain a numeric identifier that is unique among all event domains created by the target object. This numeric identifier is returned as an output parameter.

3.2.2. get_all_domains

The *get_all_domains* operation returns a sequence of all of the unique numeric identifiers corresponding to event domains that have been created by the target object.

3.2.3. get_event_domain

The *get_event_domain* operation accepts as input a numeric value that is supposed to be the unique identifier of an event domain that has been created by the target object. If this input value does not correspond to such a unique identifier, the **DomainNotFound** exception is raised. Otherwise, the operation returns the object reference of the event domain corresponding to the input identifier.

4. Typed Event Domain Interfaces

This section describes the semantic behavior of the interfaces that comprise the Typed Event Domain. The Typed Event Domain IDL is defined within the **CosTypedEventDomainAdmin** 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.

A Typed Event Domain is essentially equivalent to the Event Domain described in the previous section, with the exception that it may contain a combination of untyped and typed Notification Service channels. A more detailed explanation of the behavior of each functional aspect of the Event Domain is given in section 2 of this document.

```
#ifndef _COS_TYPED_EVENT_DOMAIN_ADMIN_IDL_
#define _COS_TYPED_EVENT_DOMAIN_ADMIN_IDL_

// Typed Event Domain  Interface
#include "CosTypedEventComm.idl"
#include "CosTypedEventChannelAdmin.idl"
#include "CosTypedNotifyChannelAdmin.idl"
#include "CosEventDomainAdmin.idl"


module CosTypedEventDomainAdmin {

  struct TypedConnection {
      CosEventDomainAdmin::MemberID supplier_id;
      CosEventDomainAdmin::MemberID consumer_id;
      CosTypedEventChannelAdmin::Key typed_interface;
      CosEventDomainAdmin::NotificationStyle notification_style;
  };

  interface TypedEventDomain :
    CosEventDomainAdmin::EventDomain {

    CosEventDomainAdmin::MemberID add_typed_channel (
                                                      in CosTypedNotifyChannelAdmin::TypedEventChannel
channel );

    CosTypedNotifyChannelAdmin::TypedEventChannel get_typed_channel (
                                              in CosEventDomainAdmin::MemberID channel )
      raises ( CosNotifyChannelAdmin::ChannelNotFound );

    // Form typed connection between two channels
      CosEventDomainAdmin::ConnectionID add_typed_connection ( in TypedConnection connection )
            raises ( CosNotifyChannelAdmin::ChannelNotFound,
                     CosEventChannelAdmin::TypeError,
                     CosEventDomainAdmin::AlreadyExists,
                     CosEventDomainAdmin::CycleCreationForbidden,
                     CosEventDomainAdmin::DiamondCreationForbidden );

    // Set default channels for typed clients
    void set_default_typed_consumer_channel (in CosEventDomainAdmin::MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    void set_default_typed_supplier_channel (in CosEventDomainAdmin::MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound);

    // Connection of clients to the domain
    // - using no specific information
    //   - for typed clients
    CosTypedNotifyChannelAdmin::TypedProxyPushSupplier
    connect_typed_push_consumer(in CosTypedEventComm::TypedPushConsumer client,
                                in CosTypedNotifyChannelAdmin::Key uses_interface)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::NoSuchImplementation,
              CosEventChannelAdmin::TypeError);

    CosTypedNotifyChannelAdmin::TypedProxyPullSupplier
    connect_typed_pull_consumer(in CosEventComm::PullConsumer client,
                                in CosTypedNotifyChannelAdmin::Key supported_interface)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::InterfaceNotSupported);

    CosTypedNotifyChannelAdmin::TypedProxyPushConsumer
    connect_typed_push_supplier(in CosEventComm::PushSupplier client,
                                in CosTypedNotifyChannelAdmin::Key supported_interface)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::InterfaceNotSupported

    CosTypedNotifyChannelAdmin::TypedProxyPullConsumer
    connect_typed_pull_supplier(in CosTypedEventComm::TypedPullSupplier client,
```

```
                                in CosTypedNotifyChannelAdmin::Key uses_interface)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::NoSuchImplementation,
              CosEventChannelAdmin::TypeError);


    // - using a channel id
    //   - for typed clients
    CosTypedNotifyChannelAdmin::TypedProxyPushSupplier
    connect_typed_push_consumer_with_id(in CosTypedEventComm::TypedPushConsumer client,
                                        in CosTypedNotifyChannelAdmin::Key uses_interface,
                                        in CosEventDomainAdmin::MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::NoSuchImplementation,
              CosEventChannelAdmin::TypeError);

    CosTypedNotifyChannelAdmin::TypedProxyPullSupplier
    connect_typed_pull_consumer_with_id(in CosEventComm::PullConsumer client,
                                        in CosTypedNotifyChannelAdmin::Key supported_interface,
                                        in CosEventDomainAdmin::MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::InterfaceNotSupported);


    CosTypedNotifyChannelAdmin::TypedProxyPushConsumer
    connect_typed_push_supplier_with_id(in CosEventComm::PushSupplier client,
                                         in CosTypedNotifyChannelAdmin::Key supported_interface,
                                         in CosEventDomainAdmin::MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::InterfaceNotSupported);

    CosTypedNotifyChannelAdmin::TypedProxyPullConsumer
    connect_typed_pull_supplier_with_id(in CosTypedEventComm::TypedPullSupplier client,
                                in CosTypedNotifyChannelAdmin::Key uses_interface,
                                in CosEventDomainAdmin::MemberID channel)
      raises (CosNotifyChannelAdmin::ChannelNotFound, CosTypedEventChannelAdmin::NoSuchImplementation,
              CosEventChannelAdmin::TypeError);

  interface TypedEventDomainFactory {

    TypedEventDomain create_typed_event_domain(
                                             in CosNotification::QoSProperties initialQoS ,
                                             in CosNotification::AdminProperties initialAdmin,
                                             out CosEventDomainAdmin::DomainID id )
      raises ( CosNotification::UnsupportedQoS,
               CosNotification::UnsupportedAdmin );

    CosEventDomainAdmin::DomainIDSeq get_all_typed_domains ();

    TypedEventDomain get_typed_event_domain (
                                            in CosEventDomainAdmin::DomainID id )
      raises ( CosEventDomainAdmin::DomainNotFound );
  };
};

#endif // _COS_TYPED_EVENT_DOMAIN_ADMIN_IDL_
```

Table 4-1: The CosTypedEventDomainAdmin Module.

4.1.The TypedEventDomain Interface

The **TypedEventDomain** interface encapsulates all behaviors supported by typed event domain objects. Typed event domain objects are capable of managing one or more topologies of interconnected channels, where each channel may be capable of supporting both typed and untyped communication.

The **TypedEventDomain** interface inherits from the **EventDomain** interface defined in the **CosEventDomainAdmin** module. This interface in turn inherits from the **QoSAdmin** and **AdminPropertiesAdmin** interfaces defined in the **CosNotifyChannelAdmin** module. Due to the inheritance of these latter two interfaces, typed event domains can be configured to support certain QoS and Admin property settings.

In addition, inheritance of the **EventDomain** interface implies that an instance of the **TypedEventDomain** interface supports the following capabilities:

- Can add an untyped channel to its domain
- Can remove a typed or untyped channel from its domain (note that the same *remove _channel* operation can be used for both typed and untyped channels)

- Can retrieve the reference of an untyped channel that exists within its domain by unique ID
- Can retrieve a list of all channels (both untyped and typed) that exist within the domain
- Can form connections between channels within its domain
- Can retrieve a structure describing a connection by unique ID
- Can retrieve a list of all connections that exist within the domain
- Can remove a particular connection from the domain
- Given the unique ID of a channel within the domain, can return the list of all supplier channels that are upstream within the same topology of interconnected channels as the input channel
- Given the unique ID of a channel within the domain, can return the list of all consumer channels that are downstream within the same topology of interconnected channels as the input channel
- Can be destroyed

In addition, as described below the **TypedEventDomain** interface defines an operation for adding a new instance of a typed channel to a domain, and an operation for returning the reference of an existing typed channel given that channel's unique ID. It also supports an operation for forming a typed connection between two typed channels belonging to the domain, and operations enabling clients of typed event channels to request connections to the domain.

4.1.1. add_typed_channel

The *add_typed_channel* operation adds a typed channel to the target domain. This operation takes the reference to a typed channel as input, and returns an identifier for the channel that is unique among all channels (both typed and untyped) contained within the domain. Note that this identifier specifically represents the channel's membership within the domain, and is not the same as the identifier assigned to the channel by the factory that created it. Having the domain assign its own identifiers to member channels enables channels created by different factories to be added to the same domain, still guaranteeing uniqueness among the identifiers assigned to channels within a domain. Thus, a particular channel may belong to multiple domains, and have a different identifier assigned to it within each domain to which it belongs.

4.1.2. get_typed_channel

The *get_typed_channel* operation accepts as input a numeric value that is supposed to be the unique identifier of a typed channel that currently exists within the target domain. 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 channel corresponding to the input identifier. Note that the identifier supplied should be one that was assigned to a channel as a result of invoking *add_typed _channel* on the target domain.

4.1.3. add_typed_connection

The *add_typed_connection* operation is used to form a typed connection between two typed channels that belong to the target domain. The operation accepts as input a data structure that describes the desired connection. This data structure includes the member IDs of the supplier and consumer channels involved in the new connection, the fully qualified name of the strongly typed interface the two channels will use to interact (this is required to connect two typed channels), and a flag indicating whether push or pull style communication will be used.

If either of the two member IDs passed as input does not correspond to the ID of a channel that is a member of the target domain, the **ChannelNotFound** exception is raised. If either of the two channels involved in the connection are not designed to use the supplied interface (e.g., if this is push-style communication, the consumer channel must support this interface, and the supplier channel must be designed to use this

interface), the **TypeError** exception is raised. If a connection between the two channels involved in the operation already exists within the domain in the same direction that would result in this new connection being formed, the **AlreadyExists** exception is raised. If cycle detection on the target domain is enabled, and the addition of the desired connection would create a cycle in the topology that this connection would be part of, the **CycleCreationForbidden** exception is raised. Likewise, if diamond detection on the target domain is enabled, and the addition of the desired connection would create a diamond in the topology that this connection would be part of, the **DiamondCreationForbidden** exception is raised.

If none of these exception conditions exist, the target domain invokes the appropriate operations on the two channels involved in the connection, in order to create the desired connection. A ID is assigned to represent the connection that is unique among all other connection IDs within the target domain, and this value is returned as the result of the operation.

4.1.4. set_default_typed_consumer_channel

The *set_default_typed_consumer_channel* operation is invoked to specify a particular channel within the target domain as the default channel to which typed consumers will be connected when they invoke one of the *connect_typed_*_consumer* operations supported by the TypedEventDomain interface, requesting in a single invocation that they be connected to the domain.

The operation accepts as input a number value that should be the unique member ID of one of the channels within the target event domain. If the input value does correspond to the member ID of one of the channels within the target domain, that channel becomes the default channel within the domain for typed consumer connections. If, however, there is no channel within the domain that has the input value as its member ID, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised.

Note that before this operation is invoked, the domain's default channel for typed consumers is set to the first typed channel added to the target domain.

4.1.5. set_default_supplier_channel

The *set_default_typed_supplier_channel* operation is invoked to specify a particular channel within the target domain as the default channel to which typed suppliers will be connected when they invoke one of the *connect_typed_*_supplier* operations supported by the TypedEventDomain interface, requesting in a single invocation that they be connected to the domain.

The operation accepts as input a number value that should be the unique member ID of one of the channels within the target event domain. If the input value does correspond to the member ID of one of the channels within the target domain, that channel becomes the default channel within the domain for typed supplier connections. If, however, there is no channel within the domain that has the input value as its member ID, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised.

Note that before this operation is invoked, the domain's default channel for typed suppliers is set to the first typed channel added to the target domain.

4.1.6. connect_typed_push_consumer

The *connect_typed_push_consumer* operation is invoked to connect a push style consumer of typed events to the target domain. The operation accepts two input parameters. The first is the reference to a typed push consumer. The second input parameter is the fully qualified name of the interface supported by the input typed push consumer that the domain should use to send typed events to it.

If the target domain contains no typed channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the default typed consumer channel within the domain does not support the ability to push events to the specified typed interface, the **CosTypedNotifyChannelAdmin::NoSuchImplementation** exception is raised. If the input typed consumer does not actually support the interface passed as the second input parameter, the **CosEventChannelAdmin::TypeError** exception is raised.

If none of these exception conditions exist, the target domain proceeds to invoke the appropriate operations

upon the default channel for typed consumers within the domain to connect the input consumer to this channel. The channel's default TypedConsumerAdmin will be used to create the appropriate typed proxy supplier instance, and the typed proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the typed proxy supplier created on behalf of the client is returned as the result of the operation.

4.1.7. connect_typed_pull_consumer

The *connect_typed_pull_consumer* operation is invoked to connect a pull style consumer of typed events to the target domain. The operation accepts two input parameters. The first is the reference to a typed pull consumer. The second input parameter is the fully qualified name of the interface the client expects the default channel to support, which it will use to pull typed events.

If the target domain contains no typed channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the default typed consumer channel within the domain is not capable of creating any typed proxy supplier that supports that interface specified by the second input parameter, the **CosTypedNotifyChannelAdmin::InterfaceNotSupported** exception is raised.

If neither of these exception conditions exist, the target domain proceeds to invoke the appropriate operations upon the default channel for typed consumers within the domain to connect the input consumer to this channel. The channel's default TypedConsumerAdmin will be used to create the appropriate typed proxy supplier instance, and the typed proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the typed proxy supplier created on behalf of the client is returned as the result of the operation.

4.1.8. connect_typed_push_supplier

The *connect_typed_push_supplier* operation is invoked to connect a push style supplier of typed events to the target domain. The operation accepts two input parameters. The first is the reference to a typed push supplier. The second input parameter is the fully qualified name of the interface the client expects the default channel to support, which it will use to push typed events.

If the target domain contains no typed channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the default typed supplier channel within the domain is not capable of creating any typed proxy consumer that supports that interface specified by the second input parameter, the **CosTypedNotifyChannelAdmin::InterfaceNotSupported** exception is raised.

If neither of these exception conditions exist, the target domain proceeds to invoke the appropriate operations upon the default channel for typed suppliers within the domain to connect the input supplier to this channel. The channel's default TypedSupplierAdmin will be used to create the appropriate typed proxy consumer instance, and the typed proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the typed proxy consumer created on behalf of the client is returned as the result of the operation.

4.1.9. connect_typed_pull_supplier

The *connect_typed_pull_supplier* operation is invoked to connect a pull style supplier of typed events to the target domain. The operation accepts two input parameters. The first is the reference to a typed pull supplier. The second input parameter is the fully qualified name of the interface supported by the input typed pull supplier that the domain should use to pull typed events from it.

If the target domain contains no typed channels, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the default typed supplier channel within the domain does not support the ability to pull events from the specified typed interface, the **CosTypedNotifyChannelAdmin::NoSuchImplementation** exception is raised. If the input typed supplier does not actually support the interface passed as the second input parameter, the **CosEventChannelAdmin::TypeError** exception is raised.

If none of these exception conditions exist, the target domain proceeds to invoke the appropriate operations

upon the default channel for typed suppliers within the domain to connect the input supplier to this channel. The channel's default TypedSupplierAdmin will be used to create the appropriate typed proxy consumer instance, and the typed proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the typed proxy consumer created on behalf of the client is returned as the result of the operation.

4.1.10. connect_typed_push_consumer_with_id

The *connect_typed_push_consumer_with_id* operation is invoked to connect a push style consumer of typed events to a specific channel within the target domain. The operation accepts three input parameters. The first is the reference to a typed push consumer. The second input parameter is the fully qualified name of the interface supported by the input typed push consumer that the channel to which the consumer will be connected should use to send typed events to it. The third input parameter is the member ID of the channel within the target domain to which the consumer should be connected.

If no channel exists within the target domain that corresponds to the unique ID supplied as the third input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the channel specified by the third input parameter does not support the ability to push events to the specified typed interface, the **CosTypedNotifyChannelAdmin::NoSuchImplementation** exception is raised. If the input typed consumer does not actually support the interface passed as the second input parameter, the **CosEventChannelAdmin::TypeError** exception is raised.

If none of these exception conditions exist, the target domain proceeds to invoke the appropriate operations upon the channel specified by the third input parameter to connect the input consumer to this channel. The channel's default TypedConsumerAdmin will be used to create the appropriate typed proxy supplier instance, and the typed proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the typed proxy supplier created on behalf of the client is returned as the result of the operation.

4.1.11. connect_typed_pull_consumer_with_id

The *connect_typed_pull_consumer_with_id* operation is invoked to connect a pull style consumer of typed events to a specific channel within the target domain. The operation accepts three input parameters. The first is the reference to a typed pull consumer. The second input parameter is the fully qualified name of the interface the client expects the channel to which it is attempting to connect to support, which it will use to pull typed events. The third input parameter is the member ID of the channel within the target domain to which the consumer should be connected.

If no channel exists within the target domain that corresponds to the unique ID supplied as the third input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the channel specified by the third input parameter is not capable of creating any typed proxy supplier that supports that interface specified by the second input parameter, the **CosTypedNotifyChannelAdmin::InterfaceNotSupported** exception is raised.

If neither of these exception conditions exist, the target domain proceeds to invoke the appropriate operations upon the channel specified by the third input parameter to connect the input consumer to this channel. The channel's default TypedConsumerAdmin will be used to create the appropriate typed proxy supplier instance, and the typed proxy supplier's connect operation will be invoked to connect the consumer to the channel. The reference of the typed proxy supplier created on behalf of the client is returned as the result of the operation.

4.1.12. connect_typed_push_supplier_with_id

The *connect_typed_push_supplier_with_id* operation is invoked to connect a push style supplier of typed events to a specific channel within the target domain. The operation accepts three input parameters. The first is the reference to a typed push supplier. The second input parameter is the fully qualified name of the interface the client expects the channel to which it is attempting to connect to support, which it will use to push typed events. The third input parameter is the member ID of the channel within the target domain to which the supplier should be connected.

If no channel exists within the target domain that corresponds to the unique ID supplied as the third input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the channel specified by the third input parameter is not capable of creating any typed proxy consumer that supports that interface specified by the second input parameter, the **CosTypedNotifyChannelAdmin::InterfaceNotSupported** exception is raised.

If neither of these exception conditions exist, the target domain proceeds to invoke the appropriate operations upon the channel specified by the third input parameter to connect the input supplier to this channel. The channel's default TypedSupplierAdmin will be used to create the appropriate typed proxy consumer instance, and the typed proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the typed proxy consumer created on behalf of the client is returned as the result of the operation.

4.1.13. connect_typed_pull_supplier_with_id

The *connect_typed_pull_supplier_with_id* operation is invoked to connect a pull style supplier of typed events to a specific channel within the target domain. The operation accepts three input parameters. The first is the reference to a typed pull supplier. The second input parameter is the fully qualified name of the interface supported by the input typed pull supplier that the channel to which the consumer will be connected should use to pull typed events from it. The third input parameter is the member ID of the channel within the target domain to which the supplier should be connected.

If no channel exists within the target domain that corresponds to the unique ID supplied as the third input parameter, the **CosNotifyChannelAdmin::ChannelNotFound** exception is raised. If the channel specified by the third input parameter does not support the ability to pull events from the specified typed interface, the **CosTypedNotifyChannelAdmin::NoSuchImplementation** exception is raised. If the input typed supplier does not actually support the interface passed as the second input parameter, the **CosEventChannelAdmin::TypeError** exception is raised.

If none of these exception conditions exist, the target domain proceeds to invoke the appropriate operations upon the channel specified by the third input parameter to connect the input supplier to this channel. The channel's default TypedSupplierAdmin will be used to create the appropriate typed proxy consumer instance, and the typed proxy consumer's connect operation will be invoked to connect the supplier to the channel. The reference of the typed proxy consumer created on behalf of the client is returned as the result of the operation.

4.2.The TypedEventDomainFactory Interface

The **TypedEventDomainFactory** interface defines operations for creating and managing typed event domains. It supports a routine that creates new instances of typed event domains and assigns unique numeric identifiers to them. In addition, the **TypedEventDomainFactory** interface supports a routine which can return the unique identifiers assigned to all event domains created by a given instance of **TypedEventDomainFactory**, and another routine which, given the unique identifier of a typed event domain created by a target **TypedEventDomainFactory** instance, returns the object reference of that domain.

4.2.1. create_typed_event_domain

The *create_typed_event_domain* operation is invoked to create a new instance of typed event domain. This operation accepts two input parameters. The first input parameter is a list of name-value that specify the initial QoS property settings for the new typed event domain. The second input parameter is a list of name-value pairs that specify the initial administrative property settings for the new typed event domain. If no implementation of the **TypedEventDomain** 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 which could be satisfied.

Likewise, if no implementation of the **TypedEventDomain** 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.

If neither of these exceptions is raised, the *create_typed_event_domain* operation will return a reference to a typed event domain. In addition, the operation assigns to this new typed event domain a numeric identifier that is unique among all typed event domains created by the target object. This numeric identifier is returned as an output parameter.

4.2.2. get_all_typed_domains

The *get_all_typed_domains* operation returns a sequence of all of the unique numeric identifiers corresponding to typed event domains that have been created by the target object.

4.2.3. get_typed_event_domain

The *get_typed_event_domain* operation accepts as input a numeric value that is supposed to be the unique identifier of a typed event domain that has been created by the target object. If this input value does not correspond to such a unique identifier, the **DomainNotFound** exception is raised. Otherwise, the operation returns the object reference of the typed event domain corresponding to the input identifier.

5. Log Domain Interfaces

This section describes the semantic behavior of the interfaces that make up the Log Domain. Log domains are a specialized type of event domain that may contain regular Notification Service channels, typed Notification Service channels, and either regular or typed notification style logs as defined by the Telecom Log Service.

Log Domain IDL defines a single **DsLogDomainAdmin** 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. A more detailed explanation of the behavior of each functional aspect of the Event Domain is given in section 2 of this document.

```
#ifndef _DS_LOG_DOMAIN_ADMIN_IDL_
#define _DS_LOG_DOMAIN_ADMIN_IDL_

// Event Log Domain  Interface
#include "CosNotification.idl"
#include "CosNotifyChannelAdmin.idl"
#include "CosEventDomainAdmin.idl"
#include "CosTypedEventDomainAdmin.idl"
#include "DsNotifyLogAdmin.idl"
#include "DsTypedNotifyLogAdmin.idl"

module DsLogDomainAdmin {


  interface EventLogDomain :
    CosTypedEventDomainAdmin::TypedEventDomain {

    CosEventDomainAdmin::MemberID add_log (
                                              in DsNotifyLogAdmin::NotifyLog log );

    DsNotifyLogAdmin::NotifyLog get_log (
                                    in CosEventDomainAdmin::MemberID log )
      raises ( CosNotifyChannelAdmin::ChannelNotFound );

    CosEventDomainAdmin::MemberID add_typed_log (
                                                    in DsTypedNotifyLogAdmin::TypedNotifyLog log );

    DsTypedNotifyLogAdmin::TypedNotifyLog get_typed_log (
                                               in CosEventDomainAdmin::MemberID log )
      raises ( CosNotifyChannelAdmin::ChannelNotFound );
  };


  interface EventLogDomainFactory {

    EventLogDomain create_event_log_domain(
                                         in CosNotification::QoSProperties initialQoS ,
                                         in CosNotification::AdminProperties initialAdmin,
                                         out CosEventDomainAdmin::DomainID id )
      raises ( CosNotification::UnsupportedQoS,
               CosNotification::UnsupportedAdmin );

    CosEventDomainAdmin::DomainIDSeq get_all_event_log_domains ();

    EventLogDomain get_event_log_domain (
                                        in CosEventDomainAdmin::DomainID id )
      raises ( CosEventDomainAdmin::DomainNotFound );
  };
};

#endif // _DS_LOG_DOMAIN_ADMIN_IDL_
```

Table 5-1: The DsLogDomainAdmin Module.

5.1. The EventLogDomain Interface

The **EventLogDomain** interface encapsulates all behaviors supported by event log domain objects. Event log domain objects are capable of managing one or more topologies of interconnected channels and logs, where each channel and log may be capable of supporting both typed and untyped communication.

The **EventLogDomain** interface inherits from the **TypedEventDomain** interface defined in the **CosTypedEventDomainAdmin** module. This inheritance enables an **EventLogDomain** to maintain

topologies of both typed and untyped channels, as well as typed and untyped logs. The **TypedEventDomain** interface inherits from the **EventDomain** interface defined in the **CosEventDomainAdmin** module. This interface in turn inherits from the **QoSAdmin** and **AdminPropertiesAdmin** interfaces defined in the **CosNotifyChannelAdmin** module. Due to the inheritance of these latter two interfaces, event log domains can be configured to support certain QoS and Admin property settings.

In addition, inheritance of the **TypedEventDomain** interface implies that an instance of the **EventLogDomain** interface supports the following capabilities:

- Can add typed and untyped channels to its domain
- Can remove a typed or untyped channel from its domain (note that the same *remove _channel* operation can be used for typed and untyped channels and logs)
- Can retrieve the reference of an untyped or typed channel that exists within its domain by unique ID
- Can retrieve a list of all channels and logs (untyped and typed) that exist within the domain
- Can form connections between channels and/or logs within its domain
- Can retrieve a structured describing a connection by unique ID
- Can retrieve a list of all connections that exist within the domain
- Can remove a particular connection from the domain
- Given the unique ID of a channel or log within the domain, can return the list of all supplier channels and logs that are upstream within the same topology of interconnected channels and logs as the input channel or log
- Given the unique ID of a channel or log within the domain, can return the list of all consumer channels and logs that are downstream within the same topology of interconnected channels and logs as the input channel or log
- Can be destroyed

In addition, as described below the **EventLogDomain** interface defines operations for adding new instances of typed and untyped logs to a domain, and operations for returning the reference of an existing typed or untyped log given that log's unique ID.

5.1.1. add _log

The *add_log* operation adds an untyped Notification log to the target domain. This operation takes the reference to an untyped Notification log as input, and returns an identifier for the log that is unique among all channels and logs (typed and untyped) contained within the domain. Note that this identifier specifically represents the log's membership within the domain, and is not the same as the identifier assigned to the log by the factory that created it. Having the domain assign its own identifiers to member logs enables logs created by different factories to be added to the same domain, still guaranteeing uniqueness among the identifiers assigned to logs within a domain. Thus, a particular log may belong to multiple domains, and have a different identifier assigned to it within each domain to which it belongs.

5.1.2. get_log

The *get_log* operation accepts as input a numeric value that is supposed to be the unique identifier of an untyped log that currently exists within the target domain. 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 log corresponding to the input identifier. Note that the identifier supplied should be one that was assigned to a log as a result of invoking *add_log* on the target domain.

5.1.3. *add_typed _log*

The *add_typed_log* operation adds a typed log to the target domain. This operation takes the reference to a typed log as input, and returns an identifier for the log that is unique among all channels and logs (typed and untyped) contained within the domain. Note that this identifier specifically represents the log's membership within the domain, and is not the same as the identifier assigned to the log by the factory that created it. Having the domain assign its own identifiers to member logs enables logs created by different factories to be added to the same domain, still guaranteeing uniqueness among the identifiers assigned to logs within a domain. Thus, a particular log may belong to multiple domains, and have a different identifier assigned to it within each domain to which it belongs.

5.1.4. *get_typed_log*

The *get_typed_log* operation accepts as input a numeric value that is supposed to be the unique identifier of a typed log that currently exists within the target domain. 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 log corresponding to the input identifier. Note that the identifier supplied should be one that was assigned to a log as a result of invoking *add_typed_log* on the target domain.

5.2. The EventLogDomainFactory Interface

The **EventLogDomainFactory** interface defines operations for creating and managing event log domains. It supports a routine that creates new instances of event log domains and assigns unique numeric identifiers to them. In addition, the **EventLogDomainFactory** interface supports a routine which can return the unique identifiers assigned to all event log domains created by a given instance of **EventLogDomainFactory**, and another routine which, given the unique identifier of an event log domain created by a target **EventLogDomainFactory** instance, returns the object reference of that domain.

5.2.1. *create_event_log_domain*

The *create_event_log_domain* operation is invoked to create a new instance of event log domain. This operation accepts two input parameters. The first input parameter is a list of name-value pairs that specify the initial QoS property settings for the new event log domain. The second input parameter is a list of name-value pairs that specify the initial administrative property settings for the new event log domain. If no implementation of the **EventLogDomain** 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 which could be satisfied.

Likewise, if no implementation of the **EventLogDomain** 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.

If neither of these exceptions is raised, the *create_event_log_domain* operation will return a reference to an event log domain. In addition, the operation assigns to this new event log domain a numeric identifier that is unique among all event log domains created by the target object. This numeric identifier is returned as an output parameter.

5.2.2. *get_all_event_log_domains*

The *get_all_event_log_domains* operation returns a sequence of all of the unique numeric identifiers corresponding to event log domains that have been created by the target object.

5.2.3. get_event_log_domain

The *get_event_log_domain* operation accepts as input a numeric value that is supposed to be the unique identifier of an event log domain that has been created by the target object. If this input value does not correspond to such a unique identifier, the **DomainNotFound** exception is raised. Otherwise, the operation returns the object reference of the event log domain corresponding to the input identifier.

Tables of illustrations

Figures:

Tables: