Air Traffic Control Specification

New Edition: May 2000

PATENT

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

NOTICE

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

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

ISSUE REPORTING

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

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

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

What is CORBA?

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

Associated OMG Documents

In addition to the CORBA Transportation specifications, the CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Languages*, a collection of language mapping specifications. See the individual language emapping specifications.
- *CORBAservices: Common Object Services Specification*, a collection of OMG's Object Services specifications.
- *CORBAfacilities: Common Facilities Specification*, a collection of OMG's Common Facility specifications.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Med*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

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

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

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

Acknowledgments

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

- Compaq Computer Corporation
- Orthogon GmbH

Interface Description 1

1.1 Purpose of the Interface

This chapter defines the CORBA interface used for implementing ATC systems that follow the Model-View-Controller paradigm. The purpose of this paradigm is to separate ATC conceptual objects (Model) from the presentation objects (View) and from the input processing (Controller). It provides maximum independence between the involved objects. It is a generic interface designed to an event model which covers the typical requirements for ATC systems. The term 'generic' means that the interface is not based on ATC application specific objects – its IDL definition is completely independent from any particular ATC system.

The following list shows the most important benefits that can be achieved by using such an architecture in connection with CORBA:

- Capability of integration of different components (e.g., control tower subsystems[1]) under a common user interface even when these are developed by different vendors.
- A change of the user interface (i.e., changing the layout or changing the kind of widgets) does not lead to changes in the ATC application code. Experience shows that most of change requests are pure HMI requirements – hence the above constitutes an important economical achievement.
- Single source principle for ATC application objects is supported. No appropriate counterpart object in the HMI component has to be programmed for adapting the HMI component to a certain ATC application object.

1. This example is related to a current commercial project for the DFS (German ATC authority) in which the proposed interface will be used for the Integrated Tower System.

- Development of HMI and ATC application software can be done separately. Since IDL scripts are the only common part of the HMI and ATC application the application programmer does not need any library from the vendor of the HMI component. The development environment can be completely different. There are no needs for upgrades in the application if the HMI component changes.
- Independence of the ATC application from the HMI product. No HMI features are visible at the interface between application and HMI. This leads to 'clear semantics' and 'clean contexts' in the development of software: a source of an ATC application will contain code that handles application specific procedures only because developers of applications have no access to presentation objects and no possibility to create presentation objects anyway.
- In most of the cases of changes less tests and re-tests are required. Changes are located in one component only (mostly HMI). That means critical sequences of code in other components of the entireATC system will not be touched. In contrast when HMI-procedures are part of an ATC application software changing would bear much more risks.
- Maintenance of HMI and ATC applications can be done separately (e.g., maintenance can be done by different teams) typically a HMI expert has to have other skills than an expert for a weather information system.
- Independence from hardware, the operational system and programming languages. HMI and application can be coded in different programming languages and may run on different architectures under different OSes.

1.2 Architectural Context

This interface is not designed specifically for a certain ATC application. It covers the communication between the Model and View part and between the Model and the Controller part in a generic way suitable for the ATC domain.

An interface between Model and Controller and View can be split in two parts because two different kinds of communication mechanisms are needed: the Controller interacts with a CO application object on a request basis whereas the View part needs an event driven mechanism to get up-to-date.

Since the CO application object is an entity which implements a CORBA interface its methods can be invoked by any client object. This also covers the needs of a Controller part completely. Moreover it provides a maximum of independence according to the wanted benefits mentioned above: the application objects implementation is independent of the way the Controller works. This CORBA interface is specific to the object it implements and hence not part of this definition (except those additions which will be needed in connection with the interface providing communication with the View part).

In contrast communication between Model and View is more complex. The View part has the need to observe changes of the application objects attributes. The way to present the information represented by the attribute values will not affect the design of the application object – it will be hidden to the object. Therefore a publish-and-subscribe notification service is put in between Model and View part.

In the following section this notification service and its interfaces are described. Also the conditions and prerequisites for the application objects and the connected clients are described.

1.3 Models And Design Patterns

1.3.1 Communication Models And Design Patterns Applied

The basis for the publish-and-subscribe notification service is the behavioral design pattern 'Observer' [Gam95]. Notification of the registered observers is handled separately by a **RealPublisher** (in [Gam95] called 'Change Manager'). This means a client ([Gam95]: an Observer) subscribes to a concrete application object ([Gam95]: a subject). It does not communicate with an event channel which hides the object that supplies the events. In contrast to the original design pattern 'observer' this notification service connects a 'personal' **RealPublisher** to each application object that wants to publish events (i.e., each application object has its own 'Change Manager'). This solution has advantages in flexibility, the publishers can be distributed independent from each other. For instance performance of distributing events is best when the **RealPublisher** is located 'close to' its subscribers (e.g., **RealPublisher** and most of the subscribers located on the same machine instead of communicating over a network).

The publish-and-subscribe service knows three event types: 'object creation,' 'object deletion,' and 'attribute change.' A special additional well-known object (CO Administrator - described below) receives and distributes the event type 'object creation.' The CO Administrator can be seen as a publisher proxy for factory objects. Clients can get informed about creation of objects – this is useful when a larger amount of temporary objects is involved which is typical for ATC systems. Interested clients can trigger their own rules to decide whether to subscribe to a created object or not. In the case of well-known objects the notification service can also be used without attaching to the CO Administrator. An object reference (e.g., returned by a naming service) can be used to directly subscribe to an application object that makes use of the publish-and-subscribe service.

The proposed interface consists of several groups of entities and knows different participants, which are:

- Conceptual Objects (COs)
- Real Publishers
- CO Subscribers
- CO Administrator
- The subscribers of the CO Administrator
- The factories for creating Real Publishers

1.3.2 Conceptual Objects

Conceptual Objects (COs) are ATC application objects that have no knowledge about their presentation. A CO has no methods whose purpose is to create an external representation of the object or the object state. However, a CO is responsible for making its creation and deletion known to the CO administrator (see below), as well as notifying certain objects (subscribers) about attribute value changes. Each CO may assign itself an object tag – a name by which it can be identified by others, meaning subscribers. Clients of the publish-and-subscribe notification service (clients of the CO Administrator) can ask for particular objects or sets of objects with a certain tag. A tag is an arbitrary string not under further control by the notification service. Object tags can be identical with the objects entry at a naming service but they need not. COs implement the **COpublisher** interface (and the **COpublisher2** interface which is a specialization of **COpublisher** for communication between the CO and its Real Publisher).

1.3.3 Real Publishers

A Real Publisher distributes a CO's events 'attribute change' and 'object deletion.' A CO delegates processing of distribution to its 'personal' Real Publisher. The assignment of a Real Publisher to a CO happens when a CO notifies the CO Administrator about its creation. Real Publishers implement the **RealPublisher** interface.

1.3.4 CO Subscribers

These are the event consumers of the CO. CO subscribers get informed about a CO's 'attribute change' and 'object deletion.' CO Subscribers implement the **COsubscriber** interface.

1.3.5 The CO Administrator

The CO Administrator is a well-known object that is notified about the creation and deletion of COs. It forwards this notification to interested parties. Within this context 'interested party' means any object which fulfills a certain interface, and explicitly makes its 'interest' known to the CO Administrator. Furthermore, the CO Administrator provides methods which allows any object to query information about the number and identity of COs which are known to the CO Administrator. With the view of a CO the CO Administrator also serves as a factory for '**RealPublisher** objects.' A CO Administrator implements three interfaces for different purposes:

1. the **COadmin** interface for communication with COs,
2. the **COadminPublisher** interface providing methods for the CO Administrator's subscribers, and
3. the **COadminControl** interface which is used only internally (communication between CO Administrator and **RealPublisher**s).

1.3.6 The subscribers of the CO Administrator

These are the event consumers of the CO Administrator. The subscribers implement the **COadminSubscriber** interface.

1.3.7 The factories for creating Real Publishers

Factories for **RealPublishers** are only known to the CO Administrator. The main purpose to explicitly define interfaces for these factories is that factories can be distributed on different machines. Factories implement the **RPfactory** interface.

1.4 Using Publish-and-Subscribe

1.4.1 How Implementations Use the Publish-and-Subscribe Service

An ATC application object (CO) delegates distribution of its events (i.e., changes of values of its attributes and its life cycle events - creation/deletion) to the publish-and-subscribe service. The life cycle events have to be forwarded to a CO Administrator object whereas the changes in values of the attributes have to be sent to a 'personal' proxy - the **RealPublisher**. The CO gets its **RealPublisher** with return of the call that notifies the CO Administrator about creation (via the **COadmin** interface). Besides distribution of events a CO delegates subscription and unsubscription to that object. The subscribe and unsubscribe methods thus become easy to implement. All the CO needs to do is to pass those calls on to the **RealPublisher** object. So a CO programmer need not implement this interface's functionality by himself. Also, such an approach would be uneconomic and error-prone. Because of platform independence, and for legal and management reasons, it is not practicable to provide CO programmers with a library which they'd be apt to link to their CO implementations. Therefore, the only design restriction is that every CO must implement the **COpublisher** interface.

In the configuration described so far, the CO Administrator is also a factory for **RealPublisher** objects. Proper de-coupling of interfaces would usually dictate a separation of these two functionalities. However, the high availability (ability to quickly recover from crash or failure) required of COs and the CO Administrator makes integrating the factory with the **COadmin** an attractive option: the CO Administrator can record the association between COs and their **RealPublishers**, and use this information during recovery.

1.5 Failure and Recovery

The failure and recovery scenarios presented in the next 3 subsections are based on the following assumptions:

- There is a mechanism which detects crashed processes and restarts them.
- The processes are able to recover their previous state after a crash and restart (e.g., by saving their state in a RAID disk array.

- COs, the CO Administrator, and the **COsubscribers/COadminSubscribers** each reside in their own process.
- The assignment of tags to objects by the publish-and-subscribe notification service may be used to group objects residing in the same address space (i.e., objects in the same address space are labeled with the same tag or tagged with the same common names).

The failure and recovery scenarios shall be supported by the standard implementation at least to the extent described below. Additional functionalities are optional.

1.5.1 Scenario 1: Crash of a CO process

When a CO process crashes, it is restarted. Upon restart, the CO process issues a call to **COadmin::delete_objs_by_name(p)**, passing its object tag 'p' as argument. This results in the following:

- All **COsubscribers** subscribed to the COs of the process are notified of the destruction by the COs **RealPublishers**.
- All associated **RealPublisher** objects are deleted. The **RealPublishers** of the previously registered COs that have the 'p' as a common part in its tag (i.e., objects which were located in the address space of the crashed process) are removed from the CO Administrator´s 'internal memory.'

Next, the CO process (re)creates its COs. Each of these COs notifies the CO Administrator of its creation, resulting in the following:

- The new CO is added to the **COadmin**´s 'internal memory' and a new **RealPublisher** is created for the new CO.
- All **COadminSubscribers** currently subscribed to the CO Administrator are notified of the creation of the new CO, resulting in the proper actions (most probably creation of a **COsubscriber** object which then subscribes to the new CO).

1.5.2 Scenario 2: Crash of the CO Administrator

When the CO Administrator crashes, it is restarted. Upon restart, the CO Administrator restores all object references previously stored in its 'internal memory. For every CO reference, this results in the following:

- A new **RealPublisher** object is created (and its **masterCO** attribute is initialized with CO's reference). The references to the subscribers of **RealPublisher** are restored. CO is notified of the fact that it must use a new **RealPublisher** (by a call to **COpublisher2::reset_real_publisher()**, with the new **RealPublisher** as its argument).
- The **COsubscribers** are not affected by the recovery of CO Administrator in any way (except that information flow from the COs had stopped for the duration of the recovery).

1.5.3 Scenario 3: Crash of a HMI application

When an HMI application crashes, it is basically up to the application what to do. Without further information at hand, two scenarios seem likely:

1. The HMI application has not saved its state

In this case, the HMI application needs to do the following (actions to be taken outside its CORBA interface not listed):

- Create a new **COadminSubscriber** object and subscribe to the **CoadminPublisher**.
- Get hold of the COs it is “interested” in. The **COadminPublisher**´s query methods are intended for just that purpose.
- Create a new **COsubscriber** object for each CO and subscribe it to that CO.

2. The application has saved its state

The HMI application may have saved its state, including all references to COs, prior to the crash. In this case, it can be restored without having to create new **COadminSubscriber** and **COsubscriber** objects. However, all present **COsubscribers** and **COadminSubscribers** must be re-subscribed to their respective COs/the **COadminPublisher**. During recovery, the CO Administrator or any **COpublisher** may have tried to notify the HMI application of a CO creation/destruction or an attribute change. During recovery, this would have resulted in an exception and subsequent removal of the **COadminSubscriber** or **COsubscriber** from the **COadminiPublisher**´s or the **COpublisher**´s subscription list.

Interface Definitions 2

2.1 IDL Type Definitions

2.1.1 Attr Structure

```
#pragma prefix “org.omg”
module ODS
{
    typedef stringAttrName;
    typedef sequence<AttrName>NameSeq;

    struct Attr
    {
        AttrName name;
        any      value;
    };

    #pragma version AttrName     1.0
    #pragma version NameSeq      1.0
    #pragma version Attr         1.0
};
```

The **Attr** structure is a generic representation of an IDL attribute. The **name** string contains the name of the attribute, as given in the IDL interface containing the attribute. The **value** contains the value of the attribute. Its actual type must be the exact type of the attribute, as given in the IDL interface containing the attribute.

2.1.2 Sequences of attributes

```
#pragma prefix “org.omg”
module ODS
{
```

```
    typedef sequence<Attr>      AttrSeq;
    typedef sequence<long>      LongSeq;
    typedef sequence<float>     FloatSeq;
    typedef sequence<string>    StringSeq;
    typedef sequence<Object>    ObjSeq;

    #pragma version AttrSeq     1.0
    #pragma version LongSeq     1.0
    #pragma version FloatSeq    1.0
    #pragma version StringSeq   1.0
    #pragma version ObjSeq      1.0
};
```

The sequences above are defined as a convenience for methods of the interfaces **RealPublisher** and **COsubscriber**.

2.1.3 Object Tag

```
#pragma prefix “org.omg”
module ODS
{
    typedef string              ObjTag;
    typedef sequence<ObjTag>    TagSeq;

    #pragma version ObjTag      1.0
    #pragma version TagSeq      1.0
};
```

An object tag identifies a CO or a whole set of COs. An object tag is an arbitrary string. The publish-and-subscribe notification service does not take care about object tags except the few simple syntax rules defined below. Since object tags do not control or influence the behavior of the notification service it is the responsibility of the applications to assign unique tags to their objects, if necessary. In general a subscriber needs to know the naming conventions of an application, there is no further regulation defined here.

The syntax of an **ObjTag** is as follows:

- A name shall be a sequence of characters taken from the set of 'printable characters' (but no spaces or tabs, etc.).
- The minimum length of a tag will be at least 5 characters.
- The first character of a tag will be taken from the set [A-Za-z0-9].

An object tag pattern describes a subset of objects whose sequence of the first N characters of its object tag are identical with the pattern where N denotes the length of the tag pattern. A given **ObjTag** pattern 'A' identifies (or matches) the set of objects labeled with object tags of which the beginning sub-string in the tag are identical with 'A'. A pattern follows the same syntax rules as defined for tags.

Example: if 'A' is an **ObjTag** pattern with the value "/example/subset" and a set of objects is labeled with "/example/subset/apple," "/example/subset/peach," "/example/subset/banana," "/example/subset/cranberry," "/example/setOfTrees/apple," then 'A' matches the first four object tags except the last one ("/example/setOfTrees/apple").

2.1.4 Association of Object and Tag

```
#pragma prefix “org.omg”
module ODS
{
    struct COwithTags
    {
        COpublisher    co;
        ObjTag         tag;
    };
    typedef sequence<COwithTags>   COseq;

    #pragma version COwithTags     1.0
    #pragma version COseq          1.0
};
```

The **COwithTags** structure is defined as a convenience for **get_objs_by_name()** and **get_all_objects()** methods of the interface **COadminPublisher**. The member **co** is an object reference to a CO. The **tag** gives the referenced CO's object tag.

2.2 ODS Module Interface Definitions

2.2.1 COpublisher

2.2.1.1 Purpose

The interface **COpublisher** must be implemented by every CO. This is the interface relevant for **COsubscribers** if they want to subscribe to an application object. It enables **COsubscriber** objects to

- subscribe to and unsubscribe from notification about CO attribute changes, and to
- inform a **COpublisher** object which subset of its attributes they are interested in.

Since a CO (ATC application object) makes use of the publish-and-subscribe notification service to publish its attribute changes all these method calls are passed through to the CO's personal publisher (which implements the **RealPublisher**) directly. This personal publisher notifies the attached subscribers. The subscribers do not know from which entity they are called.

2.2.1.2 Formal Description

```
#pragma prefix “org.omg”
module ODS
{
    exception BadAttributeName
    {
        AttrName name;
    };

    exception UnknownID {};

    interface COpublisher : BasicPublisher::Publisher
    {
        BasicPublisher::UID subscribe_co_subscriber(
            in COsubscriber sub)
        raises(BasicPublisher::SubscribeError);

        BasicPublisher::UID subscribe_co_selective(
            in COsubsciber sub,
                in NameSeq attr_names)
        raises(BasicPublisher::SubscribeError,
            BadAttributeName);

        void reset_selection (in BasicPublisher::UID sub,
                        in NameSeq attr_names)
        raises(UnknownID, BadAttributeName);

        oneway round_trip(BasicPublisher::UID initiator);
    };

    #pragma version BadAttributeName    1.0
    #pragma version UnknownID           1.0
    #pragma version COpublisher         1.0
};
```

2.2.1.3 Attributes And Methods

BasicPublisher::UID subscribe_co_subscriber (in COsubscriber sub);

This method subscribes the **COsubscriber** object **sub** to a notification about the **COpublisher**’s attribute changes.

A call to the **subscribe_co_subscriber()** method on an object A with the arguments sub registers the object **sub** with the object A (on which the method is invoked), meaning that:

- If with completion of **subscribe_co_subscriber()** the number of objects registered with the CO would exceed a maximum system parameter for Subscribers, the method **subscribe_co_subscriber()** shall throw an exception

BasicPublisher::SubscribeError{SUB_TOO_MANY} and the object referenced by the argument **sub** shall not be registered. Otherwise if the **subscribe_co_subscriber()** was successful, no exception is returned.

- After this method has been called, and until the **unsubscribe()** method of CO object A has been called with the **COsubscriber**'s **uid** as its argument, every value change in one of **COpublisher**'s IDL attributes results in an appropriate call to one of **sub**'s **COsubscriber** methods.

If any such call to a **COsubscriber sub** results in the raising of an exception, A will not invoke these methods of the object **sub**, until the **subscribe_co_subscriber()** method is again invoked successfully with **sub** as its first parameter. Therefore if notification of **sub** raises an exception, **sub** is automatically unsubscribed from object A.

After invocation and successful completion the method returns a unique ID (UID), which can be used for unsubscription at a later time.

BasicPublisher::UID subscribe_co_selective (in COsubscriber sub, in NameSeq attr_names);

This method is similar to **subscribe_co_subscriber()**. The difference is that notification of subscribers is restricted to a subset of attributes. The method subscribes the **COsubscriber** object **sub** to notification about value changes of those of **Copublisher**'s attributes whose names are given in the **attr_names** argument.

A call to the **subscribe_co_selective()** method on an object A with the arguments sub registers the object **sub** with the object A (on which the method is invoked), meaning that:

- If with completion of **subscribe_co_selective()** the number of objects registered with the CO would exceed a maximum system parameter for subscribers, the method **subscribe_co_selective()** throws an exception BasicPublisher::SubscribeError{SUB_TOO_MANY} and the object referenced by the argument **sub** will not be registered. Otherwise if the **subscribe_co_selective()** was successful, no exception is returned.
- If one of the attribute names contained in an element of parameter **attr_names** is not compliant with the rules defined for CORBA attribute names, an exception BadAttributeName{<badName>} will be raised - where <badName> means the first erroneous attribute name in the sequence **attr_names**.
- After this method has been called, and until the **unsubscribe()** method of object A has been called with the **COsubscriber's uid** (see **BasicPublisher:: Publisher**) as its argument, every value change in one of **COpublisher**'s IDL attributes results in an appropriate call to one of **sub**'s **COsubscriber** methods, if and only if the name of the changed attribute is equal to one of the names given in **attr_names** argument.

If any such call to a **COsubscriber sub** results in the raising of an exception, A will not invoke these methods of the object **sub** until the **subscribe_co_subscriber()** method is again invoked successfully with **sub** as its first parameter (if notification of **sub** raises an exception, **sub** is automatically unsubscribed from object A).

After invocation and successful completion the method returns a unique ID (UID), which can be used for unsubscription at a later time.

void reset_selection (in BasicPublisher::UID sub,
in NameSeq attr_names);

This method is used to change the subset of the **COpublisher**'s attributes whose changes are relevant to the **COsubscriber** object identified by **sub**. The result of this method is independent from the method the caller used previously to get subscribed to (i.e., the caller may get subscribed either by calling **subscribe_co_subscriber()** or by calling **subscribe_co_selective()**.

If the **COsubscriber** object (with the identification UID) is not already subscribed to, the called **COpublisher** object raises the exception UnknownID{}.

If one of the attribute names contained in an element of parameter **attr_names** is not compliant with the rules defined for CORBA attribute names, an exception BadAttributeName{<badName>} is raised – where <badName> means the first erroneous name in the sequence **attr_names**.

After invocation and successful completion of **reset_selection()** the **COpublisher** object will initiate its **RealPublisher** object to call the appropriate method of **sub**'s **COsubscriber** interface only for changes in value of those attributes whose name is equal to one of the names given in **attr_names** argument.

If the number of **attr_names** elements is 0 (zero), the **COpublisher** object will call the appropriate method of **sub**'s **COsubscriber** interface for changes in value of any attribute of the **COpublisher** object (this is like getting subscribed by calling **subscribe_co_subscriber()**).

oneway round_trip (in BasicPublisher:UID initiator);

This method requests sending a 'round trip message.' A round trip message is forwarded to the **RealPublisher**, which forwards the call again back to the originator (and only to the originator) of the round trip. In cases when it is absolutely not predictable when the next attribute change will happen, the round trip is a feature that gives a subscriber the chance to verify whether the whole chain of information distribution is still alive.

The called CO will forward this call 1-to-1 to its **RealPublisher** object.

2.2.1.4 Related Interfaces

COpublisher2: the **COpublisher2** interface is an extension of **COpublisher**, which is needed for communication between the publish-and-subscribe notification service and a CO.

RealPublisher: an application object that implements **COpublisher2** and **COpublisher** uses the **RealPublisher** interface to delegate publishing.

2.2.2 COpublisher2

2.2.2.1 Purpose

The **COpublisher2** interface is a **COpublisher** interface enhanced with a method that allows reassigning a **RealPublisher** to a CO. This only is necessary when a **RealPublisher** fails and cannot be replaced by an object with the same object reference (which is the usual case). In this case the entity of the notification service that runs the **RealPublishers** has to update the attached CO. Therefore this is the interface a CO presents to its **RealPublisher** – any other subscriber needs not to know this kind of interface.

2.2.2.2 Formal Description

```
#pragma prefix “org.omg”
module ODS
{
    interface COpublisher2 : COpublisher
    {
        void reset_real_publisher(
            in RealPublisher real_publisher);
    };

    #pragma version COpublisher2 1.0
};
```

2.2.2.3 Attributes And Methods

void reset_real_publisher (in RealPublisher real_publisher);

A call to this method tells a **COpublisher2** object that its **RealPublisher** object is no longer valid. A reference to a new, replacement **RealPublisher** object is passed as a parameter.

If with invocation of **reset_real_publisher()** the publish-and-subscribe notification service gets an exception, the previously created **RealPublisher** remains active and will not be deleted.

This implies that the data necessary for recovery of **RealPublishers** has to be kept in a non-volatile memory. The entity that runs the **RealPublishers** will make sure that for each known CO the reference with its corresponding tag, UID and the subscriber references will not be lost or corrupted.

2.2.3 COsubscriber

2.2.3.1 Purpose

The **COsubscriber** interface is the counterpart to **COpublisher**. An object that implements **COsubscriber** can:

- register (subscribe) for notification about CO attribute changes with a **COpublisher** object,
- receive notification about CO attribute changes from a **COpublisher** object

The methods of **COsubscriber** are not discussed one by one in detail here, since they all follow the same pattern. Instead, a general description is given.

Invoking these methods on **COsubscriber** objects is exclusively handled by **RealPublisher** objects - see definition of **RealPublisher**.

2.2.3.2 Formal Description

```
#pragma prefix “org.omg”
module ODS
{
    interface COsubscriber : BasicPublisher::Subscriber
    {
        void set_long (in ObjTag co,
            in AttrName name,
            in long value);
        void set_float (in ObjTag co,
            in AttrName name,
            in float value);
        void set_string (in ObjTag co,
            in AttrName name,
            in string value);
        void set_object (in ObjTag co,
            in AttrName name,
            in Object value);
        void set_any (in ObjTag co,
            in AttrName name,
            in any value);

        void set_long_seq (in ObjTag co,
            in AttrName name,
            in LongSeq value);
        void set_float_seq (in ObjTag co,
            in AttrName name,
            in FloatSeq value);
        void set_string_seq (in ObjTag co,
            in AttrName name,
            in StringSeq Value);
```

```
        void set_object_seq (in ObjTag co,
            in AttrName name,
            in ObjectSeq value);

        void set_attributes (in ObjTag co,
            in AttrSeq attrs);

        void obj_deleted (in ObjTag co);

        oneway round_trip (in ObjTag called_co);
    };

    #pragma version COsubscriber 1.0
};
```

2.2.3.3 *Attributes And Methods*

Most of the `COsubscriber` methods have one of the following forms:

```
void set_<type of attribute> (in ObjTag co,
                              in AttrName attr_name,
                              in <type of attribute> value);
```

and

```
void set_<type of attribute>_seg (in ObjTag co,
                              in AttrName attr_name,
                              in <type of attribute> value);
```

A call to this method notifies a **COsubscriber** object of an attribute value change in one of the **COpublisher** objects it is subscribed to.

The first parameter is the object tag of the **COpublisher** object. This parameter enables the **COsubscriber** to know which of the **COpublisher**'s it is subscribed to has sent the notification.

The second parameter is the name of the attribute whose value has changed, as given in the IDL definition of the object in question (usually an object that inherits and augments the **COpublisher** interface).

The third parameter is the new value of the attribute. Types supported are:

- long
- float
- string
- any
- Object

Type short is supported by passing it as a long type. Also, sequences of values of these types are supported. They are provided to support indexed attributes like (e.g., eto[]).

If the subscriber that has implemented a **COsubscriber** interface raises any exception, the notification service unsubscribes this subscriber immediately.

In the case where a subscribed **COsubscriber** is (temporarily) not reachable by the notification service (e.g., due to data flow control when the server object is overloaded), the notification service is waiting for the server object's readiness and tries to send the request as long as no further notification of the same attribute is indicated; otherwise, this first notification is aborted and notification is resumed with the new (current) attribute value. This means the newest notification may kill processing of a yet uncompleted notification and a subscriber may not assume that each notification is delivered.

void set_attributes (in ObjTag co, in AttrSeq attrs);

A call to this method notifies the **COsubscriber** object of a change to several attributes, whose names and values are given in the sequence **attrs**.

In the case where a subscribed **COsubscriber** is (temporarily) not reachable by the notification service (e.g., due to data flow control when the server object is overloaded), the notification service is waiting for the server object's readiness and tries to send the request as long as no further notification is indicated; otherwise, this first notification is aborted and notification is resumed with the new (current) attribute values. This means the newest notification kills processing of a yet uncompleted notification and a subscriber may not assume that each notification is delivered.

void obj_deleted (in ObjTag co);

A call to this method notifies the **COsubscriber** object of the deletion of the CO. The subscriber may assume that both the reference to this CO and the UID of this subscription is no longer valid.

In the case where a subscribed **COsubscriber** is (temporarily) not reachable by notification service (e.g., due to data flow control when the server object is overloaded), the notification service is waiting for the server object's (subscriber) readiness for 3 seconds maximum[1] and tries to send the request. With invocation of **obj_deleted()** a previously initiated but (for this subscriber) uncompleted notification is aborted - a deletion is notified instead. This means the **obj_deleted()** notification kills processing of a yet uncompleted notification and a subscriber may not assume that each notification is delivered.

After notification of the **COsubscriber** the corresponding data to a CO known by the CO Administrator is deleted. No subsequent query to **COadminPublisher** delivers this data of the CO deleted.

oneway round_trip (in ObjTag called_co);

---

1. This particular time value was chosen based on the requirements for the current implementation project. However, a standard implementation may parameterize this value and set it (e.g., based on an environment variable).

This method is the response to a round trip message requested by a call of the method **round_trip()** of a **COpublisher**. With initiating a round trip message the calling subscriber of a CO specifies its **uid** as an argument. This response call delivers the object tag of the called CO. This is because a **uid** only is unique within the scope of a **RealPublisher**. In the case where an object is subscribed to several COs, it would be possible that some **uids** contain the same value.

The parameter **called_co** (the object tag) denotes the object that was previously called. The rules for invocation of **round_trip()** are defined in the interface description of **RealPublisher**.

2.2.3.4 Related Interfaces

COpublisher: an object that wants to subscribe to an application object uses the **COpublisher** interface.

2.2.4 RealPublisher

2.2.4.1 Purpose

The **RealPublisher** interface can be used to delegate the actual publishing functionality out of a **COpublisher2** implementation. Meaning, this is the interface a CO uses to delegate publishing of attribute changes to the notification service. All the **COpublisher2** implementation needs to do is

- create/get a reference to a ('personal') **RealPublisher** object
- forward all calls to the methods defined by the **COpublisher** interface (**subscribe()**, **unsubscribe()** methods) to that **RealPublisher** object
- call the appropriate **set_<type of attribute>()** method of **RealPublisher** whenever one of its attribute values changes. The 'appropriate method' is the method of the **COsubscriber** interface with the same name as the called method of **RealPublisher**. This means that the CO fully decides how to notify the subscribers. In cases where several attributes have to change simultaneously a CO may use the **set_attributes()** method. But a CO should not use **set_attributes()** and **set_XXXX()** methods intermixed for the same attributes. A subsequent **set_XXXX()** may abort a previously initiated but uncompleted **set_attributes()** in a certain situations when a subscriber is overloaded (see definition of **COsubscriber** interface for more details).

2.2.4.2 Formal Description

```
#pragma prefix “org.omg”
module ODS
{
    interface RealPublisher : COpublisher
    {
        // This attribute is a reference to the CO which
```

```
        // uses a RealPublisher object to delegate
        // notification about attribute changes
        readonly attribute COpublisher masterCO;

        void set_long (in AttrName name,
            in long   value);
        void set_float (in AttrName name,
            in float value);
        void set_string (in AttrName name,
            in string value);
        void set_object (in AttrName name,
            in Object value);
        void set_any (in AttrName name,
            in any value);

        void set_long_seq (in AttrName name,
            in LongSeq value);
        void set_float_seq (in AttrName name,
            in FloatSeq value);
        void set_string_seq (in AttrName name,
            in StringSeq value);
        void set_object_seq (in AttrName name,
            in ObjectSeq value);

        void set_attributes (in AttrSeq attrs);

        void obj_deleted();
    };

    #pragma version RealPublisher 1.0
};
```

2.2.4.3 *Attributes And Methods*

The **set_<type of attribute> ()** and **set_<type of attribute>_seq ()** methods follow the same pattern as the **set_<type of attribute>()** and **set_<type of attribute>_seq()** methods of the **COsubscriber** interface.

Their purpose is different, however. It is a request for forwarding the value in parameter 'value' of the attribute denoted by parameter 'name' to the subscribed objects.

A call to such a method **set_<type of attribute>()** causes a **RealPublisher** object to call the corresponding method of all **COsubscriber** objects, which are subscribed to the **COpublisher** and therefore subscribed to the corresponding **RealPublisher** object. The corresponding methods are the methods of the **Cosubscriber** interface that have the same name.

The notification service will not wait for completion of publishing. The call to the **RealPublisher** is handled asynchronously and returns immediately to provide a de-coupling from the subscribers.

void obj_deleted ();

With invocation of this method a CO (**COpublisher**) notifies its **RealPublisher** that it will be deleted immediately after the call returns. The **RealPublisher** notifies all the **COsubscriber** objects of the deletion of the CO by calling the corresponding **obj_deleted()** method of each **COsubscriber**. The subscriber may assume that both the reference to this CO and the UID of this subscription is no longer valid.

In case a subscribed **COsubscriber** is (temporarily) not reachable by the notification service (e.g., due to data flow control when the server object is overloaded), the notification service is waiting for the server object's (subscriber) readiness for 3 seconds maximum and tries to send the request.

With invocation of **obj_deleted()** a previously initiated but (for this subscriber) uncompleted notification is aborted, a deletion is notified instead.

After notification of the **COsubscribers** and regardless whether the notification was successful or not the **RealPublisher** initiates deletion of itself. The **RealPublisher** and the corresponding data to a CO known by the CO Administrator shall be deleted. No subsequent query to **COadminPublisher** shall deliver this data of the CO deleted.

The notification service will not wait for completion of publishing. The call to the **RealPublisher** will be handled asynchronously and will return immediately to provide a de-coupling from the subscribers.

oneway round_trip (in BasicPublisher:UID initiator);

This method (inherited by the **COpublisher** interface) requests sending a round trip message.

With invocation of **round_trip()** the called **RealPublisher** object will forward this call 1-to-1 to the subscriber with the uid given in parameter **initiator**. The **RealPublisher** calls the corresponding method **round_trip()** of the initiator and passes the object tag as an argument. Only the initiator which has the given UID is called, no round trip operation is performed on all the other subscribers.

2.2.4.4 Related Interfaces

COsubscriber: a **RealPublisher** distributes attribute changes to subscribers that have implemented a **COsusbscriber** interface.

2.2.5 COadmin

2.2.5.1 Purpose

The **COadmin** interface provides a means for COs to notify **COadminSubscribers** about their creation, without the CO having to know the number or identity of the interested (subscribed) **COadminSubscriber**.

With notification about creation the calling CO gets a **RealPublisher** object. There is also a service to delete **RealPublishers**. The **COadmin** interface comprises the services needed for COs - these are the event suppliers. The services for subscribing and unsubscribing on these creation events is covered by the **COadminPublisher** interface.

The entity of the notification service that handles all these services (a CO-Administrator) can be seen as an object implementing both interfaces.

2.2.5.2 *Formal Description*

```
#pragma prefix “org.omg”
module ODS
{
    exception BadTag{};
    exception NoMatch{};
    exception NoResources {};

    interface COadmin
    {
        RealPublisher obj_created (in COpublisher2 obj,
                                   in ObjTag tag)
            raises (BadTag, NoResources);

        void delete_objs_by_name (in ObjTag tagpattern);
            raises (BadTag, NoMatch);
    };

    #pragma version BadTag1.0
    #pragma version NoMatch1.0
    #pragma version NoResources1.0
    #pragma version COadmin 1.0
};
```

2.2.5.3 *Attributes And Methods*

RealPublisher obj_created (in COpublisher2 co, in ObjTag tag);

A call to this method notifies the **COadmin** object of the creation of the CO '**co**,' which bears the object tag '**tag**.' The notification results in the following:

- **co** is added to the **COadmin** object’s internal memory, meaning that the results of subsequent calls to any of the methods **get_all_objects()**, **get_objs_by_name()** of the corresponding **COadminPublisher** will include **co** (and its object tag), if **co** matches the criteria passed to these methods.
- An instance of a **RealPublisher** is created. This instance´s attribute **masterCO** is initialized with **co**. The **co** uses this object to delegate notification of subscribers.

- If the **COadmin** object is for some reason unable to create a **RealPublisher** object, an exception NoResources{} is raised.
- If the '**tag**' argument passed to the method violates the syntax rules, an exception BadTag{} is raised.
- For every subscribed **COadminSubscriber** object that is subscribed to the corresponding **COadmin**, its method **COadminSubscriber::obj_created()** is called, with **co** as the first parameter, **tag** as the second parameter. This call has to be deferred until after the **RealPublisher** object is returned to the caller (i.e., after the call to **obj_created()** proper has terminated).

void delete_objs_by_name (in ObjTag tagpattern);

A call to this method notifies the **COadmin** object of the destruction of all objects, which matches the object **tag tagpattern**, resulting in the following:

- If the **tagpattern** argument passed to the method violates the syntax rules, an exception BadTag{} will be raised.
- If no object whose object tag match the argument **tagpattern** is present in **COadmin**'s internal memory, the following exception is raised: COadmin::NoMatch{};
- The objects whose object tag match the argument **tagpattern** are removed from the **COadmin** object's internal memory.
- For every subscribed **RealPublisher** object that is connected to a CO that matches the tag given in parameter **tagpattern**, the **obj_deleted()** method is invoked by the **COadmin** (i.e., as soon as this method is called), the **COadmin** object may not assume that the objects matching the given **tagpattern** still exist.

2.2.5.4 *Related Interfaces*

None.

Note – An administrator entity that implements **COadmin** also implements an interface **COadminPublisher**. Additionally it has to provide **COadminControl** for internal purposes meaning the capability to initiate creation of **RealPublishers** running on different machines.

2.2.6 *COadminPublisher*

2.2.6.1 *Purpose*

The **COadminPublisher** interface serves the following purposes:

- provides a means for interested objects (which must fulfill the **COadminSubscriber** interface) to register (subscribe) for notification about CO creation.

- provides a means for interested objects (probably, but not necessarily **COadminSubscriber**s) to get information about the number and identity of existing COs.

2.2.6.2 *Formal Description*

```
#pragma prefix “org.omg”
module ODS
{
    interface COadminPublisher : BasicPublisher::Publisher
    {
        BasicPublisher::UID subscribe_ad_subscriber(
            in COadminSubscriber sub)
        raises(BasicPublisher::SubscribeError);

        BasicPublisher::UID subscribe_ad_selective(
            in COadminSubscriber sub,
                in TagSeq tagpatterns)
        raises(BasicPublisher::SubscribeError, BadTag);

        void reset_selection(
            in BasicPublisher::UID sub,
                in TagSeq tagpatterns)
        raises(UnknownID, BadTag);

        COseq get_all_objects();

        COseq get_objs_by_name(in ObjTag tagpattern)
        raises(BadTag);
    };

    #pragma version COadminPublisher 1.0
};
```

2.2.6.3 *Attributes And Methods*

```
BasicPublisher::UID subscribe_ad_subscriber (
                        in COadminSubscriber sub);
```

This method subscribes the object **sub** to notification about creation of COs.

A call to the **subscribe_ad_subscriber()** method on a **COadminPublisher** object A with the arguments **sub** registers the object sub with the **COadminPublisher** object A (on which the method is invoked).

If with completion of **subscribe_ad_subscriber()** the number of objects registered with the **COadminPublisher** would exceed a maximum system parameter for subscribers, then the **subscribe_ad_subscriber()**method will throw an exception BasicPublisher::SubscribeError{SUB_TOO_MANY} and the object referenced by the argument **sub** will not be registered.

After **subscribe_ad_subscriber()** has returned, and until **unsubscribe()** is successfully invoked on object A with the subscribers **uid** as argument, the **COadminSubscriber::obj_created()** method will be invoked once on the object **sub** every time a creation event on A is notified via its corresponding **COadmin** interface. This invocation will take place immediately after the method on **COadmin** interface has been invoked.

If an invocation of **obj_created()** on **sub** results in the raising of an exception, A will not invoke the **obj_created()** method the object **sub**, until the **subscribe_ad_subscriber()** method is again invoked successfully with **sub** as its first parameter. If notification of **sub** raises an exception, **sub** is automatically unsubscribed from object A.

After invocation and successful completion the method returns an unique ID (UID) that can be used for unsubscription at a later time.

BasicPublisher::UID subscribe_ad_selective (in COadmin Subscriber sub,
inTagSeq tagpatterns);

This method is similar to **subscribe_ad_subscriber()**. The difference is that notification of a subscriber is only performed if the created object (that has notified CO Administrator) matches one of the **tagpattern**s the subscriber has specified with this call. The method subscribes the **COadminSubscriber** object **sub** to notification about creation of those of **CO**'s whose tags are matching the tag patterns given in the **tagpattern** argument.

A call to the **subscribe_ad_selective()** method on a CO object A with the arguments sub registers the object **sub** with the CO object A (on which the method is invoked), meaning that:

- If with completion of **subscribe_ad_selective()** the number of objects registered with the CO would exceed a maximum system parameter for subscribers, the method **subscribe_ad_selective()** will throw an exception BasicPublisher::SubscribeError{SUB_TOO_MANY} and the object referenced by the argument **sub** will not be registered.
- If one of the tag patterns contained in an element of parameter **tagpatterns** is not compliant with the syntax rules defined for object tags, an exception BadTag{<badTag>} is raised – where <badTag> means the first erroneous tag pattern in the sequence **tagpatterns**.
- After this method has been called, and until the **unsubscribe()** method of object A has been called with the **COadminSubscriber**'s **uid** (see **BasicPublisher::Publisher**) as its argument, every creation of a CO results in an

appropriate call to **sub**'s **COadminSubscriber** method **obj_created()**, if and only if the tag of the created CO matches one of the **tagpatterns** given in the **tagpatterns** argument.

If any such call to a **COadminSubscriber sub** results in the raising of an exception, A will not invoke these methods of the object **sub** until the **subscribe_ad_subscriber()** method is again invoked successfully with **sub** as its first parameter. If notification of **sub** raises an exception, **sub** is automatically unsubscribed from object A.

After invocation and successful completion the method returns a unique ID (UID) that can be used for unsubscription at a later time.

void reset_selection (in BasicPublisher::UID sub,
in TagSeq tagpatterns);

This method is used to change the set of **tagpatterns** associated with a subscriber identified by **sub**.

The result of this method is independent from the method the caller used previously to get subscribed to. For example, the caller may get subscribed either by calling **subscribe_ad_subscriber()** or by calling **subscribe_ad_selective()**.

If the calling **COadminSubscriber** object with the identification UID is not already subscribed to the called **COadminPublisher** object, an exception unknownID{} is raised.

If one of the tag patterns contained in an element of parameter **tagpatterns** is not compliant with the syntax rules defined for object tags, an exception BadTag{<badTag>} will be raised - where <badTag> means the first erroneous tag pattern in the sequence **tagpatterns**.

After invocation and successful completion of the method, notification of a subscriber is only performed if the created object matches one of the tag patterns the subscriber **sub** has specified with this call.

If the number of argument **tagpatterns** elements is 0 (zero), the **CoadminPublisher** object will call the appropriate method of **sub**'s **COadminSubscriber** interface for each creation of a CO regardless of the tag of the created object (this is like getting subscribed by calling **subscribe_ad_subscriber()**).

COseq get_objs_by_name (in ObjTag tagpattern);

This method returns a list of COs whose object tag matches the **tagpattern** passed as argument, respectively, if and only if:

- the **COadmin** object was notified of the creation of such a CO by a call to its **obj_created()** method (about behavior see description of **obj_created()**) AND
- before invocation of **get_objs_by_name()** no method **delete_objs_by_name()** with a matching tag as parameter (same tag or tag that denotes a superset of parameter **tagpattern**) was invoked on the corresponding **COadmin** (to notify the **COadmin** object of CO destruction).

If the **tagpattern** argument does not match the syntax rules, an exception BadTag{} is raised. Otherwise (i.e., tag with correct syntax but no matching object is found in the **COadmin**'s internal memory), a void object reference (**CORBA::_nil**) is returned. The list is returned as a sequence of **COwithTags** structures.

COseq get_all_objects ();

This method returns a list of COs. A CO will be included in that list if and only if:

- the **COadmin** object was notified of the creation of such a CO by a call to its **obj_created()** method (about behavior see description of **obj_created()**) AND
- before invocation of **get_objs_by_name()** no **COadmin** method **delete_objs_by_name()** with a matching tag as parameter (same tag or tag that denotes a superset of parameter tag) was called to notify the **COadmin** object of CO destruction.

The list is returned as a sequence of **COwithTags** structures. Otherwise (i.e., no object is found in the **COadmin**'s internal memory), a void object reference (**CORBA::_nil**) is returned.

2.2.6.4 Related Interfaces

COadminSubscriber: the publisher that implements a **COadminPublisher** interface provides subscribers that implement **COadminSubscriber** interfaces.

2.2.7 COadminSubscriber

2.2.7.1 Purpose

The **COadminSubscriber** interface enables objects that implement it to:

- register (subscribe) for notification about CO creation with an object that implements a **COadminPublisher** interface,
- receive notification about CO creation from a **COadminPublisher** object.

2.2.7.2 Formal Description

```
#pragma prefix "org.omg"
module ODS
{
    interface COadminSubscriber :
        BasicPublisher::Subscriber
    {
            void obj_created(in COpublisher obj, in ObjTag tag);
    };

    #pragma version COadminSubscriber 1.0
};
```

2.2.7.3 Attributes And Methods

void obj_created (in COpublisher obj, in ObjTag tag);

A call to this method notifies the **COadminSubscriber** object of the creation of a CO. The CO's object tag is passed in the **tag** argument.

In case a subscribed **COadminSubscriber** is (temporarily) not reachable by the notification service (e.g., due to data flow control when the server object is overloaded), the notification service is waiting for the server objects readiness and tries to send the request.

In this case a **COadminPublisher** buffers a maximum of 2 notifications[2]. Otherwise, if a third notification occurs the first notification is aborted and notification is resumed with a call of **obj_created()** where the parameter **tag** is an empty string and **obj** is a nil-reference. This means if a subscriber has received an 'empty' notification, it has to ask **COadminPublisher** for the object references that are not delivered (by using the **get_objs_by-_name()** method).

2.2.7.4 Related Interfaces

COadminPublisher: a **COadminSubscriber** subscribes to a CO administrator using its **COadminPublisher** interface.

2.2.8 RPfactory

2.2.8.1 Purpose

In the interest of flexibility, it must be possible to delegate the creation of **RealPublisher** objects out of the **COadmin** object. This is done using **RealPublisher** factories, the interface description of which is given below. A configuration file determines which **RealPublisher** factory has to be used for a specific **COpublisher** object. Upon start-up, the **COadmin** object reads a plain text file. Each line of this file has one of the following formats:

<empty>		(1)
#<comment>		(2)
<name of publisher factory>	default	(3)
<name of publisher factory>	<object tag>	(4)

The items in lines of the formats (3) and (4) are separated by one or more tabs, respectively. Lines of one of the above formats may occur in the file in any order.

---

2. This particular number of notifications was chosen based on the requirements for the current implementation project. However a standard implementation may parameterize this value and set it (e.g., based on an environment variable).

The configuration file is interpreted by the **COadmin** as follows:

- The **COadmin** uses whatever mechanism it chooses to create **RealPublisher** objects when the **createObj** method is called, unless specified otherwise in the configuration file.
- The first line in the file which contains only an object name (an object name that must be translated into an object reference by a naming service) and the keyword 'default' (format 3 above) will be interpreted as follows: the **COadmin** uses the factory identified by the name to create a **RealPublisher** for every **COpublisher** object passed as an argument to the **obj_created()** method, unless specified; otherwise, in another line of format (4) in the configuration file. All subsequent lines which contain a name of a factory will be ignored, (i.e., they have no effect on the behavior of the **COadmin**).
- The first line in the file that contains a name of a factory and an **object tag** (format 4) will be interpreted as follows: the **COadmin** uses the factory identified by the name to create a **RealPublisher** for every **COpublisher** passed as an argument to a call of the **obj_created()** method, if the **tag** argument to that method call matches the tag. All subsequent lines which contain a name of a factory and a tag will be ignored (i.e., the first line found that matches the tag of the created object will be applied).
- An empty line (format 1) and a line beginning with a hash sign (format 2) has no effect on the behavior of the **COadmin**.
- Any line that does not conform to one of the formats specified above is considered faulty. The behavior of the **COadmin** when encountering a faulty line is not defined here, but must be documented by the provider of a specific implementation of the proposed standard.

2.2.8.2 Formal Description

The **RPfactory** interface must be implemented as specified here by anyone who wants to customize the **COadmin** by supplying alternative mechanisms for creating **RealPublisher** objects, as described in the previous section. The supplier of a **COadmin** implementation may, but need not provide any implementation of the **RPfactory** interface.

```
#pragma prefix “org.omg”
module ODS
{
    // an ID by which a RealPublisher is identified
    // by its corresponding CO Administrator
    typedef long RPID;

    interface RPfactory
    {
        RealPublisher create_rp(
            in COpublisher2 co,
                in ObjTag tag,
                in COadminControl calling_ad,
```

```
                    in RPID rpid)
                raises(NoResources);

            void delete_objs_by_name(in ObjTag tagpattern);
                raises (BadTag, NoMatch);
        };

        #pragma version RPID1.0
        #pragma version RPfactory1.0
};
```

2.2.8.3 *Attributes And Methods*

RealPublisher create_rp (in COpublisher2 co, in ObjTag tag,
in COadminControl calling_ad,
in RPID rpid)

This method creates a **RealPublisher** object for the **COpublisher2** object **co** with object tag **tag**, meaning that:

- if the factory object is for some reason unable to create a **RealPublisher** object, an exception NoResources{} is raised.
- a **RealPublisher** object is created, whose **master_CO** attribute is equal to **co**.

The parameter **co** represents the reference to the application object (CO) for which the newly created **RealPublisher** will be a proxy in distribution of attribute changes.

The parameter **tag** represents the object tag of the assigned CO (represented by parameter **co**).

The parameter **calling_ad** holds a reference to the calling administrator object.

The parameter **rpid** is a unique identifier assigned by the calling administrator object. Since a **RealPublisher** has to be unique within the scope of an administrator object assignment of an RPID is the responsibility of the caller. The factory forwards the **rpid** and the reference in parameter **calling_ad** to the **RealPublisher** internally. This **rpid** is used for callbacks to the **COadminControl** interface (e.g., in case of recovery procedures).

After invocation and successful completion the method returns a reference to the created **RealPublisher** object.

void delete_objs_by_name (in ObjTag tagpattern);

A call to this method requests the **RPfactory** object for destruction of all objects that match the object tag **tagpattern**, resulting in the following:

- if the **tagpattern** argument passed to the method violates the syntax rules, an exception BadTag{} will be raised.
- if no object whose object tag matches the argument **tagpattern** is present in **RPfactory**'s internal memory, the following exception is raised: NoMatch{};

- the objects whose object tag matches the argument **tagpattern** are removed from the **RPfactory** object's internal memory.

2.2.8.4 Related Interfaces

None.

Note – An object that implements an **RPfactory** interface is only invoked by a CO administrator.

2.2.9 COadminControl

2.2.9.1 Purpose

In the case when factories of **RealPublishers** are distributed objects (see **Rpfactory**) a CO Administrator has to provide an interface to get informed from these remote entities. For the same reason a **RealPublisher** has to notify its corresponding CO when the object reference has changed, a CO Administrator needs it too (see **COpublisher2**). The interface **COadminControl** provides methods for **RealPublishers** to call back its administrator.

2.2.9.2 Formal Description

```
#pragma prefix "org.omg"
module ODS
{
    interface COadminControl
    {
        exception BadID{};

        void reset_rp(in long rpid, in RealPublisher rp)
            raises(BadID);

        void rp_deleted(in long rpid) raises(BadID);
    };
    #pragma version COadminControl 1.0
};
```

2.2.9.3 Attributes And Methods

void reset_rp (in long rpid, in RealPublisher rp);

A call to this method tells a **COadminControl** object that the reference to its **RealPublisher** object identified by the **rpid** is no longer valid (the **rpid** which is assigned by the called **RPfactory** during creation of a **RealPublisher** and returned to the calling CO Administrator). A reference to a new, replacement **RealPublisher** object is passed as a parameter.

If with invocation of **reset_rp()** the **COadminControl** object does not have registered a **RealPublisher** with an **rpid** given in parameter **rpid**, the exception BadID{} is raised.

If with invocation of **reset_rp()** the calling **RealPublisher** gets an exception, the **RealPublisher** remains and will not be deleted.

void rp_deleted (in long rpid, in RealPublisher rp);

A call to this method tells a **COadminControl** object that the reference to its **RealPublisher** object identified by the **rpid** is no longer valid (the **rpid** which is assigned by the called **RPfactory** during creation of a **RealPublisher** and returned to the calling CO Administrator).

If with invocation of **rp_deleted()** the **COadminControl** object does not have registered a **RealPublisher** with an **rpid** given in parameter **rpid**, an exception BadID{} is raised.

Regardless if with invocation of **rp_deleted()** the calling **RealPublisher** gets an exception or not the **RealPublisher** will be deleted.

2.2.9.4 Related Interfaces

RealPublisher: a **RealPublisher** uses the **COadminControl** interface to call back its administrator.

2.3 BasicPublisher Module Interface Definitions

2.3.1 Type Definition UID

2.3.1.1 Purpose

The type **UID** is used as a key for subscription and unsubscription of subscribers. Use of such a unique identifier is necessary in algorithms where comparison of object references is needed - this is not identical with CORBA references [Callb].

2.3.1.2 Formal Description

```
#pragma prefix “org.omg”
module BasicPublisher
{
    typedef long UID;
    typedef sequence<UID> UIDSeq;

    #pragma version UID1.0
    #pragma version UIDSeq 1.0
};
```

2.3.1.3 Description Of The Elements

A UID represents a unique key to identify a method’s calling entity. An instance of a UID contains an arbitrary unique number defined locally by the called object.

2.3.2 Publisher

2.3.2.1 Purpose

This is the abstract Publisher interface. Methods of this interface allow to subscribe and unsubscribe certain subscribers for notification (about events). The abstract Publisher interface is introduced to enable a generic subscribe and unsubscribe mechanism.

Each particular Publisher will inherit from this abstract interface. The two inherited methods **subscribe** and **unsubscribe** allow to register any abstract subscriber for an event. Subscribers registered through these methods have to pull the event. Abstract subscribers are only notified that an event has happened. The event itself is not propagated automatically. Instead interested subscribers have to pull the event actively (when they have been notified). This model enables a very generic pull model.

Each Publisher may specify additional **subscribe/unsubscribe** methods for particular subscribers. The two new methods allow to register concrete subscribers for certain known events. This allows to introduce Push Style notification. Subscribers registered through these methods don't have to pull the event. Instead when the notification happens, the event is pushed immediately to the registered Subscribers. This enables a more coupled push model.

2.3.2.2 Formal Description

```
#pragma prefix “org.omg”
module BasicPublisher
{
    interface Publisher
    {
        enum SubscribeErrorCode
```

```
    {
        SUB_TOO_MANY,
        SUB_NOT_REGISTERED
    };

    exception SubscribeError {
        SubscribeErrorCode error;
    };

    UID subscribe(in Subscriber sub, in boolean send_ref)
        raises(SubscribeError);

    boolean is_subscribed(in UID sub);

    void unsubscribe(in UID sub)
        raises(SubscribeError);
};

#pragma version Publisher 1.0
};
```

2.3.2.3 *Attributes And Methods*

Exception SubscribeError

SubscribeError is an exception that will be raised when errors occur during subscription and unsubscription.

It will contain SUB_TOO_MANY if too many objects are registered for subscription an the limit is exceeded.

It will contain SUB_NOT_REGISTERED if this Object/UID is not registered for subscription

Exception SubscribeError

SubscribeErrorCode is an enumeration that indicates the reason for the exception.

UID subscribe (in Subscriber sub, in boolean send_ref)

This method subscribes the subscriber object **sub** to notification about certain events, offered by the Publisher. The parameter **send_ref** decides whether to send the Object Reference during the notification or not. If **send_ref** equals FALSE, then **update_subscriber()** is called. If **send_ref** equals TRUE, then **update_subscriber_from_publisher()** is called.

A unique ID (UID) is returned that can be used for unsubscription at a later time. If the method fails, exceptions of type SubscribeError can be raised indicating the source of the problem.

void unsubscribe (in UID sub)

This method unsubscribes the subscriber object with its UID **uid** from notification about certain events. The unique ID (UID) **sub** must have been returned from a previous call to subscribe. If the method fails, exceptions of type SubscribeError can be raised indicating the source of the problem.

boolean is_subscribed (in UID sub);

This method is used to test whether the subscriber with the UID is still subscribed to the publisher. Since a subscriber can be unsubscribed automatically in case of an exception the publisher gets with notification a subscriber has no information about that.

This call returns a value TRUE if and only if the UID given in argument **sub** is known by the called publisher; otherwise, the method return a value of FALSE.

2.3.2.4 Related Interfaces

A **Publisher** expects a **Subscriber** interface implemented by a client object that is calling **subscribe()**.

2.3.3 Subscriber

2.3.3.1 Purpose

This is the abstract **Subscriber** interface. Methods of this interface allow to receive notification about events. The abstract **Subscriber** interface is introduced to enable a generic subscribe and unsubscribe mechanism. Each particular Subscriber will inherit from this abstract interface. The two inherited methods **update_subscriber()** and **update_subscribe_from_publisher()** allow to notify **Subscriber** when certain events occur. The **Subscriber** itself then has to pull the event from the **Publisher**. This model enables a very generic pull model.

Each particular **Subscriber** may specify additional notification/update methods. The new methods allow **Publishers** to send an event directly to the **Subscriber**, using Push Style and avoiding the extra **round_trip** for pulling the event. This requires a more narrow coupling because **Subscriber** and **Publisher** have detailed knowledge of each other.

2.3.3.2 Formal Description

```
#pragma prefix “org.omg”
module BasicPublisher
{
    interface Subscriber
    {
        void update_subscriber();
```

```
        void update_subscriber_from_publisher(
            in Publisher pub);
    };

    #pragma version Subscriber 1.0
};
```

2.3.3.3 *Attributes And Methods*

This method notifies the subscriber object about an event offered by the publisher. The parameter **send_ref** passed during subscription (see **subscribe()** function of the Publisher interface) decides, whether to send the object reference or not.

- If **send_ref** equals FALSE, then **update_subscriber()** is called**.**
- If **send_ref** equals TRUE, then **update_subscriber_from_publisher()** is called. The subscriber can pull the event from the publisher.

2.3.3.4 *Related Interfaces*

A Subscriber has to register by calling the object that implements a Publisher interface.

ODS IDL *A*

```
#ifndef _ODS_IDL_
#define _ODS_IDL_

#pragma prefix “org.omg”
module ODS
{
  // type definitions

        typedef string                  AttrName;
        typedef sequence<AttrName> NameSeq;
        struct Attr
        {
            AttrName name;
            any      value;
        };
        typedef sequence<Attr>          AttrSeq;
        typedef sequence<long>          LongSeq;
        typedef sequence<float>         FloatSeq;
        typedef sequence<string>        StringSeq;
        typedef sequence<Object>        ObjSeq;
        typedef string                  ObjTag;
        typedef sequence<ObjTag>        TagSeq;
        struct COwithTags
        {
            COpublisher    co;
            ObjTag         tag;
        };
        typedef sequence<COwithTags>   COseq;
        typedef long RPID;

        #pragma version AttrName     1.0
        #pragma version NameSeq      1.0
        #pragma version Attr         1.0
```

```
        #pragma version AttrSeq        1.0
        #pragma version LongSeq        1.0
        #pragma version FloatSeq       1.0
        #pragma version StringSeq      1.0
        #pragma version ObjSeq         1.0
        #pragma version ObjTag         1.0
        #pragma version TagSeq         1.0
        #pragma version COwithTags     1.0
        #pragma version COseq          1.0
        #pragma version RPID           1.0

// exception definitions

        exception BadAttributeName
        {
            AttrName name;
        };
        exception UnknownID {};
        exception BadTag{};
        exception NoMatch{};
        exception NoResources {};

        #pragma version BadAttributeName   1.0
        #pragma version UnknownID          1.0
        #pragma version BadTag             1.0
        #pragma version NoMatch            1.0
        #pragma version NoResources        1.0


// interface definitions

        interface COpublisher : BasicPublisher::Publisher
        {
            BasicPublisher::UID subscribe_co_subscriber(
                in COsubscriber sub)
            raises(BasicPublisher::SubscribeError);

            BasicPublisher::UID subscribe_co_selective(
                in COsubsciber sub,
                    in NameSeq attr_names)
            raises(BasicPublisher::SubscribeError,
                BadAttributeName);

            void reset_selection (in BasicPublisher::UID sub,
                            in NameSeq attr_names)
            raises(UnknownID, BadAttributeName);

            oneway round_trip(BasicPublisher::UID initiator);
        };

        interface COpublisher2 : COpublisher
```

```
{
    void reset_real_publisher(
        in RealPublisher real_publisher);
};

interface COsubscriber : BasicPublisher::Subscriber
{
    void set_long (in ObjTag co,
        in AttrName name,
        in long value);
    void set_float (in ObjTag co,
        in AttrName name,
        in float value);
    void set_string (in ObjTag co,
        in AttrName name,
        in string value);
    void set_object (in ObjTag co,
        in AttrName name,
        in Object value);
    void set_any (in ObjTag co,
        in AttrName name,
        in any value);

    void set_long_seq (in ObjTag co,
        in AttrName name,
        in LongSeq value);
    void set_float_seq (in ObjTag co,
        in AttrName name,
        in FloatSeq value);
    void set_string_seq (in ObjTag co,
        in AttrName name,
        in StringSeq Value);
    void set_object_seq (in ObjTag co,
        in AttrName name,
        in ObjectSeq value);

    void set_attributes (in ObjTag co,
        in AttrSeq attrs);

    void obj_deleted (in ObjTag co);

    oneway round_trip (in ObjTag called_co);
};

interface RealPublisher : COpublisher
{
    readonly attribute COpublisher masterCO;

    void set_long (in AttrName name,
        in long   value);
    void set_float (in AttrName name,
```

```
            in float value);
        void set_string (in AttrName name,
            in string value);
        void set_object (in AttrName name,
            in Object value);
        void set_any (in AttrName name,
            in any value);

        void set_long_seq (in AttrName name,
            in LongSeq value);
        void set_float_seq (in AttrName name,
            in FloatSeq value);
        void set_string_seq (in AttrName name,
            in StringSeq value);
        void set_object_seq (in AttrName name,
            in ObjectSeq value);

        void set_attributes (in AttrSeq attrs);

        void obj_deleted();
    };


    interface COadmin
    {
        RealPublisher obj_created (in COpublisher2 obj,
                                   in ObjTag tag)
            raises (BadTag, NoResources);

        void delete_objs_by_name (in ObjTag tagpattern);
            raises (BadTag, NoMatch);
    };

    interface COadminPublisher : BasicPublisher::Publisher
    {
        BasicPublisher::UID subscribe_ad_subscriber(
            in COadminSubscriber sub)
        raises(BasicPublisher::SubscribeError);

        BasicPublisher::UID subscribe_ad_selective(
            in COadminSubscriber sub,
                in TagSeq tagpatterns)
        raises(BasicPublisher::SubscribeError, BadTag);

        void reset_selection(
            in BasicPublisher::UID sub,
                in TagSeq tagpatterns)
        raises(UnknownID, BadTag);

        COseq get_all_objects();
```

```
        COseq get_objs_by_name(in ObjTag tagpattern)
        raises(BadTag);
    };

    interface COadminSubscriber :
        BasicPublisher::Subscriber
    {
            void obj_created(in COpublisher obj, in ObjTag tag);
    };

    interface RPfactory
    {
        RealPublisher create_rp(
            in COpublisher2 co,
                in ObjTag tag,
                in COadminControl calling_ad,
                in RPID rpid)
            raises(NoResources);

        void delete_objs_by_name(in ObjTag tagpattern);
            raises (BadTag, NoMatch);
    };



    interface COadminControl
    {
        exception BadID{};

        void reset_rp(in long rpid, in RealPublisher rp)
            raises(BadID);

        void rp_deleted(in long rpid) raises(BadID);
 };


    #pragma version COpublisher          1.0
    #pragma version COpublisher2         1.0
    #pragma version COsubscriber         1.0
    #pragma version RealPublisher        1.0
    #pragma version COadmin              1.0
    #pragma version COadminPublisher     1.0
    #pragma version COadminSubscriber 1.0
    #pragma version RPfactory            1.0
    #pragma version COadminControl       1.0

};

#endif
```

BasicPublisher.IDL *B*

```
#ifndef _BasicPublisher_IDL_
#define _BasicPublisher_IDL_

#pragma prefix “org.omg”

module BasicPublisher
{
        // type definitions

        typedef long UID;
        typedef sequence<UID> UIDSeq;

        #pragma version UID1.0
        #pragma version UIDSeq1.0

        // interface definitions

        interface Publisher
        {
            enum SubscribeErrorCode
            {
                SUB_TOO_MANY,
                SUB_NOT_REGISTERED
            };

            exception SubscribeError {
                SubscribeErrorCode error;
            };

            UID subscribe(in Subscriber sub, in boolean send_ref)
                raises(SubscribeError);

            boolean is_subscribed(in UID sub);
```

```
        void unsubscribe(in UID sub)
            raises(SubscribeError);
    };

    interface Subscriber
    {
        void update_subscriber();

        void update_subscriber_from_publisher(
            in Publisher pub);
    };

    #pragma version Publisher1.0
    #pragma version Subscriber1.0
};

#endif
```

References *C*

[Gam95] Gamma, Helm, Johnson, Vlissides. Design Patterns, 1995 Addison-Wesley

[Callb] Distributed Callbacks and Decoupled Communication in CORBA, Washington University of St. Louis, http://siesta.wustl.edu/~schmidt/report-doc.html

Requirements D

D.1 Statement Of Proof Of Concept

The general concept of the chosen approach has been proven by Orthogon during the Integrated Tower System (ITS) presentation in summer 1997 at the DFS site (German ATC authority).

Currently Orthogon implements a CORBA interface for the HMI of the ITS (which is developed with Orthogon's product ODS Toolbox) within a commercial project for the DFS. This interface fully corresponds to the interface proposed in this submission.

D.2 Resolution Of Mandatory And Optional requirements

All the specified modules and interfaces are mandatory.

Failure and recovery scenarios described in chapter 2.5 shall be supported, too. This support shall at least cover the functionality mentioned in the scenarios, whereas it is possible to enhance optionally the failure and recovery features beyond the given description.

There are few arbitrary defined absolute values like e.g. number of seconds for time out which – though mandatory – can be handled differently in different standard implementations. All these values are marked and described correspondingly in the document.

D.3 Responses To RFP Issues

The specification proposed in this document addresses explicitly only the interface to the ATC Display Manager. It is not a specification of the Display Manager – understood as a HMI component of the ATC system – itself. Therefore, there are no interfaces to particular ATC-relevant entities. Thus all the specific requirements listed in RFP's Section 6 are not addressed as assumed by OMG. The reason is given in Section 1.10 above.

D.4 Relationship To Pending OMG Specifications

The communication from the ATC Display Manager to other components of the entire ATC system is not discussed here. For this purpose a standardized interface to these components is required. Having e.g. a standard IDL interface to the Flight Data Processor system it will be possible to use functions provided there when an HMI needs e.g. to report about flight leg modifications done by the ATC controller.

These issues may be handled by other ATC-related RFPs which will target particular ATC sub-system(s). One of such RFPs is about to be published (“Surveillance Manager for Air Traffic Control and Management” RFP, OMG Document: transprt/98-06-01).

The specification proposed in this document is closely related to the existing OMG specifications of the Event Service and Notification Service.

In the ATC domain the use of the Event Channel for the HMI was not satisfactory because there is a need of well defined data structures. It is also required to have direct access to the application objects (COs) at the ATC server side (i.e. the ground systems like RDPs, FDPs, all components of tower systems etc.). In addition a publish-and-subscribe mechanism is needed so that many consoles which may be both the same[1] as well as different[2] type can use the same CORBA interface for their HMI layer and still get only the information they need. A publish-and-subscribe mechanism is also not specified in the Event Service.

One of the main goals in development of this interface was to achieve a good maintainability and changeability mainly for the HMI part in ATC systems (which also implies: provide a means that supports writing 'safe' code). The MVC paradigm is the right solution for this problem. Using the publish-and-subscribe mechanism is another step to achieve de-coupling between application and HMI component - an ATC application needs not to be tailored to the current needs of a HMI process.

Avoiding visibility is another very important requirement in HMI for ATC. An entity in the HMI component deals only with one and the same object reference per object. This simplifies coding and makes it more safe. There is no need to manage mapping of event channels or notification entities to application objects.

The semantics of supported events is clear, too. There are only 'attribute changes' and ’creation/deletion of objects'. There is no setting of filters and no selection of events via a third entity. Such things are error prone. In cases of wrong settings errors in HMI components may occur: for instance, events from the wrong objects. These are errors of a kind not known before in HMI components programmed in the old fashioned style (legacy systems not following the MVC paradigm). This would mean, the new technology is less safe than the old one because it introduces another possibility to code bugs in the HMI component. It is also not compliant with the goal to reduce effort

---

1. For example there are usually very many (40, 120 or more) CWPs installed in an Upper Air-space Management ATC center. These CWPs are usually able to perform exactly the same functions for any sector in the controlled air space; they are 'exchangeable'.
2. For example consoles in tower systems may differ significantly concerning the design and implementation depending on the specialized functions they support.

in maintenance and adaptations of user interfaces. The maintenance issue is perhaps one of the most important problems for the end users (final customers) - according to the submitters' experiences in ATC domain, the HMI layer is far more often changed then the underlying server layer which can often be seen as legacy systems.

The Notification Service is much more generic than the one proposed in this document. But this flexibility will cost performance (e.g. the rules for filtering have to be interpreted at runtime each time an event occurs). This means that the flexibility the Notification Service provides is not needed but must be handled by the programmer and its overhead may create performance bottlenecks in the entire ATC system.

Summarizing the aforementioned concerns the Notification Service is not used because:

1. A direct access to the application CO (not its proxy) is required.
2. The number of events is reduced to simplify the use of the interface and to avoid processing overhead.
3. There is no need for extensive filtering as it is provided in the Notification Service.
4. The proposed CORBA interface shall include only basic publish-and-subscribe functions which due to its intended simplicity shall be:
   - slim hence fast
   - robust - it shall not allow unpredictable errors in effect of maintenance operations performed by technical personal in an ATC center (non-programmers)
   - easy-to-learn hence easy-to-use by HMI programmers and ground system programmers
   - simple to implement in the existing ATC systems - as low as possible interactions with the legacy code.
5. A running implementation is required now. The CORBA interface to the ODS Toolbox based exactly on the proposed specification has been implemented in 1998 and is currently used by the DFS in the IDVS project. The implementation will be commercially available as a COTS product in 1999.

The publish-and-subscribe mechanism proposed in this document is not as flexible as OMG's Notification Service. There may be cases where an application object (CO) has to distribute also other event types than defined here. In this case a possible scenario could be that this CO has to deal with more than one notification service: one for high performance forwarding of attribute changes and one for distributing more complex events.

The interface definition provided in this proposal is well suited to the ATC domain. It fulfils many important requirements not covered by other standards of the OMG although the services are principally nearly the same.

Index

Index