Journal of Computer Science 2 (6): 535-541, 2006
ISSN 1549-3636
© 2006 Science Publications

MEDiator: A Tool for Automatic Management of Event Domains

Markus Aleksy, Lisa Köblitz and Martin Schader
Department of Information Systems, University of Mannheim
Schloss, D-68131 Mannheim, Germany

Abstract: In this study, we describe our software component MEDiator. Our development is based on OMG's Management of Event Domains specification. It allows the efficient management and simplified operation of different CORBA Notification Services running concurrently. After a brief introduction into the specifications of the Notification Service and the Management of Event Domains, we describe their architecture and discuss the most important interfaces. Following, we review the shortcomings of the current specification and delineate our approach to solving the problems that result from these deficiencies.

Key words: CORBAservices, CORBA, event domains

INTRODUCTION

The Common Object Request Broker Architecture (CORBA) Standard[1] defined by the Object Management Group (OMG) is widely popular in the area of distributed, object-oriented applications. In addition to the independence of employed hardware architecture, operating system and programming language, this is mainly a consequence of the specification of an interoperable system architecture that governs information exchange between implementations based upon products of different providers.

To describe the interfaces of classes offering services, the OMG introduced the *Interface Definition Language* (IDL). IDL is a purely declarative language; with its help, the necessary data types and interfaces, together with their attributes, operations and exceptions, are defined. No algorithmic details are implemented, however. CORBA's programming language independence is based on the IDL. Only an IDL compiler translates the interface definitions into a concrete programming language.

The *Object Request Broker* (ORB) is the fundamental component for communication in distributed CORBA applications. In order to aid application developers during their work, the OMG has, furthermore, standardized a number of system-related services, the *CORBAservices*. These services extend the basic functionality of the ORB.

For example, with the *Event Service*[2], which realizes the Publisher-Subscriber design pattern[3], CORBA was extended by an asynchronous, decoupled communication mode. In this context, different roles, namely the *Publisher* and the *Consumer*, as well as two different message models, the *Push* and the *Pull* model, were defined. Furthermore, the standard differentiates a typed from an untyped model. The core of the specification is the *Event Channel*, which acts as a mediator between the publishers and the consumers. By specifying the *CORBA Notification Service* (CNS)[4], which extends CORBA's Event Service, many necessary extensions, for example, the possibility to filter events, were later added. This specification, however, still contained several weaknesses. Problems such as coordinating the collaboration of several CNSs running in parallel were left to the developers although such questions are of essential importance for the scalability or fault tolerance of a distributed system. Moreover, some of the CNSs' processes, e.g., connecting several event channels, are rather complex.

In order to correct the deficiencies just mentioned, the OMG published the *Management of Event Domains* (MED) specification[5]. MED is a supplement to the proper service specification and makes design of the CNSs' processes much more user-friendly.

An implementation of the currently available version 1.0 of the MED specification raises a considerable number of problems since the standard contains various flaws. Nevertheless, we decided to realize the specification in our MEDiator project.

The architecture of the med specification: When joining several event channels and linking different clients with these event channels, a rather elaborate topology can evolve very rapidly, the management of which can become extremely complex. By creating an *Event Domain*, even complex topologies can be administered conveniently.

The purpose of an event domain is to manage one or more groups of interconnected event channels. These event channels can be created through already existing implementations of the CNSs that might be running in

Corresponding Author: Markus Aleksy, Department of Information Systems, University of Mannheim, Germany

parallel on different hosts. This would entail the considerable advantage that existing programs using the CNSs' event channels can be extended to employ event domains without any program modifications.

In analogy to the structure of the CNS, the MED specification defines IDL interfaces for untyped event domains that manage generic, untyped event channels. Furthermore, IDL interfaces for typed event domains that can manage untyped as well as typed event channels are provided. Moreover, the specification contains IDL interfaces for *Event Log Domains* managing untyped and typed event channels and *Logs* that are defined in the *Telecom Log Service* specification[6].

The IDL interfaces `EventDomainFactory` and `EventDomain`: The interface `EventDomain-Factory` specifies operations for creating and managing untyped event domains.

Event domains supporting the CNSs' untyped event channels are specified with the IDL interface `EventDomain`. Clients, i.e., *Suppliers* or *Consumers*, wanting to register with an event domain first have to connect to an event channel of that event domain.

In order to group together event channels or to connect a client to an event channel, proxy objects are used; these are based upon IDL interfaces specified in the CNSs. If an event channel's proxy supplier is connected to the proxy consumer of another event channel, then the first event channel is called *Supplier Channel* and the other is the *Consumer Channel*. A connection between two event channels can already be set up by solely utilizing the CNSs' IDL interfaces. This approach, however, is quite tedious and needs a number of operation invocations: with the help of a `ConsumerAdmin` object of the supplier channel, a proxy supplier is constructed and with the help of a `SupplierAdmin` object of the consumer channel, one obtains a proxy consumer. Subsequently, the proxy consumer and the proxy supplier have to be connected to the supplier channel and the consumer channel, respectively. In the Management of Event Domains approach, these steps are combined into one single operation.

Before two event channels can be connected, they both have to be registered with the event domain and need to have obtained a unique ID. The connection itself has a specific data structure that contains the IDs of the event channels, the "`ClientType`" and the "`NotificationStyle`". The `ClientType` determines whether the connection is to be used for events of type `Any`, `Structured`, or `Sequence`; whereas, the `NotificationStyle` indicates whether the two channels will communicate using a push or a pull method. An event channel that has been registered with an event domain can be associated with an arbitrary number of other event channels or clients of the event domain.

By generating connections, a topology of event channels is created. This corresponds to a directed graph where each event channel registered with the event domain is a vertex (or node) and each connection is an edge of the graph. The graph can be of arbitrary complexity; it can contain cycles or diamond shapes, meaning that the same event may reach a vertex in the graph by more than one path. Within that graph, suppliers can send events to consumers. When a supplier uses the event channel with which it is registered to send an event, the event is not only delivered to the consumers of that event channel but also to all the event channels that are connected to it. The reason is that the proxy supplier of the supplier channel plays the supplier role for the consumer channel and vice versa, the proxy consumer of the consumer channel plays the consumer role for the supplier channel.

Before creating an event domain, it should be considered whether cycles or diamonds in the directed graph are admissible or not. Note that cycles may result in the unpleasant consequence that events might loop endlessly through the graph and that, in topologies containing diamonds, a consumer may receive the same event more than once. It is possible to prevent such behavior by setting the Quality of Service (QoS) properties `CycleDetection` and `Diamond-Detection` appropriately. If, for example, the `CycleDetection` value is set to `ForbidCycles`, then an attempt to establish a connection between two event channels that would introduce a cycle will raise a `CycleCreationForbidden` exception.

In an event domain, information on the event types provided by suppliers, as well the event types, in which the consumers are interested, can be stored. Each event channel contains a local database that provides information on the event types that are being offered or subscribed. In the CNS, a mechanism is defined that enables a supplier to inform all consumers on the event types that it will be propagating in the future. Here, the supplier has to not only manage communication completely, but the event channel is subsequently responsible for communicating the information to each consumer. The supplier merely informs its proxy consumer whereupon the event channel informs all the consumers connected to it. If a connection of this event channel to other channels exists, the information will also be passed to all the consumers of these event channels; information concerning the event types can therefore be communicated to each of the consumers in the event domain. Analogously to this "*subscription_change*" mechanism, an "*offer_change*" mechanism is built into the CNS, which enables a consumer to inform suppliers that it is interested in certain kinds of event types.

The IDL interfaces `TypedEventDomainFactory` and `TypedEventDomain`: The `TypedEvent-DomainFactory` interface specifies operations for creating and managing typed event domains.

The IDL interface `TypedEventDomain` is a subinterface of the `EventDomain` interface and thus, inherits all functionality from an untyped event domain. In addition to untyped communication, a typed event domain also supports a typed communication mode.

All the operations of an untyped event domain, for example, registration of a client with an untyped event channel, are extended for typed event channels and for clients needing to use typed events.

If a typed connection between two typed event channels has to be formed, both event channels must have been previously registered with the typed event domain. The typed connection itself has its own, specific data structure that consists of the IDs of the event channels, the `NotificationStyle` (push or pull) and the name of the interface the channels will use to interact.

The IDL Interfaces `EventLogDomainFactory` and `EventLogDomain`: The IDL interface `Event-LogDomainFactory` specifies operations for creating and managing event log domains.

An event log domain maintains one or more topologies of interconnected event channels and logs, where each event channel and event log may be capable of supporting both typed and untyped communication. Logs are objects that implement the IDL interface `NotifyLog` or `TypedNotifyLog`, respectively.

The `EventLogDomain` is a subinterface of the IDL interface `TypedEventDomain`, which, in turn, inherits from the `EventDomain` interface. Therefore, an event log domain is a specific typed event domain and inherits all the functionality of a typed event domain, for example, adding or removing typed or untyped event channels from a domain. In addition, an event log domain defines operations for managing typed or untyped logs, which are described in the Telecom Log Service[7].

Critical assessment of the specification: It is the MED specification's goal to provide the definition of a simplified management structure for different CNS event channels. The MED architecture enables developers to reuse and enhance existing implementations based on the CNS. For example, when relying solely on the CNS, six operations have to be invoked to connect a client to an event channel; the MED specification defines IDL interfaces that establish such a connection with a single operation invocation. In the same way, connections between event channels can be easily installed.

According to the CNSs' rules, QoS properties that are set on the event domain level should be on a hierarchically higher level than QoS properties set on the level of event channels. If, for example, on event domain level the QoS property *Order Policy* was set to the value *FifoOrder*, then any event channel registered with the event domain must send events according to the FIFO mode. However, since this hierarchy can only be supported by new implementations of the CNS, observing the hierarchy rules is sacrificed in favor of compatibility. By doing this, the general concept of QoS properties is broken.

The default values set for the QoS properties `CycleDetection` and `DiamondDetection` allow cycles as well as diamonds. This does not seem to be particularly appropriate as it is better to avoid cycles and diamonds in order to prevent circling or multiple deliveries of events, a point that is repeatedly underlined in the specification. Potential problems are discussed in the specification; however, no example where the admittance of cycles within an event domain would have any advantage is mentioned. The only conceivable reason why, for example, diamonds should be allowed is that in distributed systems, one can always argue with the failure of one of the participating hosts. If various paths exist, transmission reliability can be increased. It has to be noted, however, that the number of deliveries per event and therefore network traffic, will grow proportionally with the number of diamonds. In order to prevent unintentional network load, the default values should disallow cycles and diamonds.

In general, the current specification does not yet appear to be a conscientious piece of work. Another example: an operation `get_typed_connection`, which would be the counterpart of the `EventDomain`'s operation `get_connection`, is missing in the IDL specification of interface `TypedEventDomain`. The obvious flaws of the specification, which we will discuss in more detail in the following section, are even more serious.

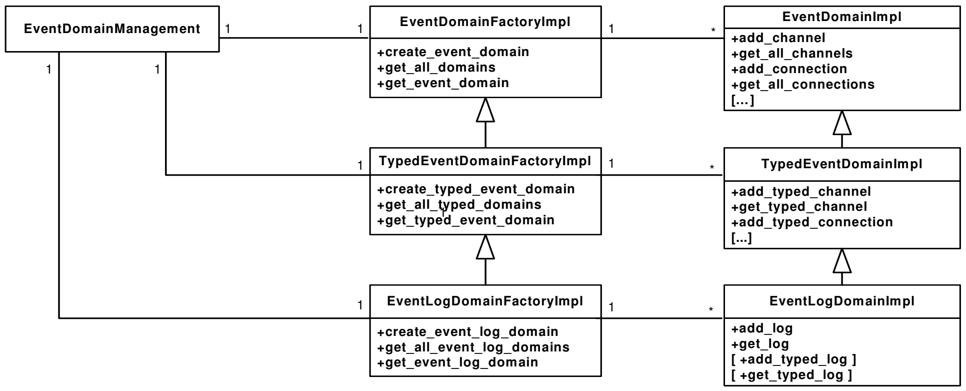

Fig. 1: UML diagram of MEDiator

Deficiencies of the specification: Currently, only version 1.0 of the MED specification is available. Several deficiencies are evident, however, in that version, e.g., it contains several invalid identifiers and some of the definitions are imprecise or inconsistent. We can distinguish the following four deficiency categories:

Errors concerning identifiers: In the MED's IDL specification, invalid identifiers are used several times. This holds, for example, for the connect operations defined in the IDL interface `TypedEventDomain`, which raises the "wrong" type of exception. Errors of this kind may be detected by thoroughly studying the different documents; but, the implementation of the specification is made more difficult.

Imprecise specification of default channels: Under the current specification, it is provided that one specific event channel in an event domain is designated to be the domain's *Default Consumer Channel*. In the case that a consumer is registered without specifying an event channel ID, it is connected to the default consumer channel. Likewise, a *Default Supplier Channel* has to be identified; this channel will be connected to a supplier that registers without an event channel ID. According to the specification, the first event channel that registers with the event domain is to be used as default consumer channel and also as default supplier channel. Furthermore, operations that can be invoked to later install some other event channel as default supplier channel or default consumer channel are specified. But, it is not clarified what should happen when the default supplier channel or the default consumer channel are removed from the event domain.

Inconsistencies concerning exception `Diamond-CreationForbidden`: In the module `CosEvent-DomainAdmin`, the exception `DiamondCreation-Forbidden` is defined such that it only contains a single diamond. In the description of operation `add_connection`, however, one finds the following sentence: "*The exception contains as data a sequence of conflicting paths, each path being a sequence of channel member identifier.*" To that purpose, the type `DiamondSeq`, defined in the same IDL interface, would have to be used and the exception `DiamondCreationForbidden` would have to be defined as: `exception DiamondCreation-Forbidden { DiamondSeq diam; };`

Fundamental error in the Event Log Domain architecture: Probably the most serious error is contained in the architecture of the event log domain, which should be able to manage typed as well as untyped logs. The necessary `#include` statements, which read in `DsTypedNotifyLogAdmin.idl` and `DsNotifyLogAdmin.idl`, have the consequence that a multiple inheritance structure that will be rejected as erroneous by the IDL compiler is created.

Selected implementation aspects: In our MEDiator implementation, each of the above-mentioned IDL interfaces is implemented through a corresponding class `<Interface> Impl.java`. Figure 1 shows the UML class diagram for all implemented classes. The class `EventDomainManagement` serves as the basis for MEDiator. Depending on the command line parameters, it generates an event domain factory (`EventDomainFactoryImpl`), a typed event domain factory (`TypedEventDomainFactory-Impl`), or an event log domain factory (`EventLogDomainFactoryImpl`), which is able to create any number of event domains (`EventDomainImpl`), typed event domains (`Ty-pedEventDomainImpl`), or event log domains (`EventLogDomainImpl`), respectively. In the following, we describe our implementation and explain how we solved the problems caused by the specification's deficiencies.

Properties of an event domain: At the event domain level, there are two QoS properties:

`CycleDetection` und `DiamondDetection`. If, during the creation of the event domain, the respective QoS property is not handed over, the event domain is assigned the default values, that is, both cycles as well as diamonds are allowed. In creating an event domain, not only QoS, but also admin properties can be defined. In the current specification, however, there are no admin properties defined on the event domain level. However, it is conceivable that in the future, for example, the maximum number of event channels in the event domain will be defined or that complexity will be reduced through a limitation of the number of cycles.

Default channels: As already described, it is not specified exactly what should happen if the default supplier channel or the default consumer channel respectively disconnects from the event domain. Since the first event channel that connects with the event domain shall be defined as the default supplier channel and default consumer channel, it is surely the most intuitive solution that, in the case of disconnecting one of the default channels, the event channels next in line, meaning those with the next lowest event channel ID, should take the place of the disconnected default channels. Therefore, this solution was implemented.

Using an adjacency matrix: An event domain contains a group of event channels that can be connected with each other and that lead to a directed graph whose vertexes represent the event channels while the edges represent their connections with each other. There are two ways of implementing a directed graph: the representation in an adjacency matrix and the representation in an adjacency list. The advantage of the list is its relatively small size with regard to $O(|V|+|E|)$, with $|V|$ being the number of vertexes and $|E|$ being the number of edges. An adjacency matrix requires $O(|V|^2)$, allowing, however, an easy calculation of the incidence that in an adjacency list depends on the arrangement of vertexes and edges. Although, with respect to the asymptote, using an adjacency list is equally efficient as using the matrix, Coreman[8] suggests the use of an adjacency matrix as long as the number of vertexes is relatively small and especially for non-weighted graphs, for overview and saving purposes in general. Since in the adjacency matrix the connections are saved as boolean values, each entry requires only one bit.

Event channel IDs and event domain IDs: When connecting, the event domain assigns an individual, unique ID to each event channel. Here it was an issue to decide whether an ID that became available again due to disconnecting event channels should be reused or whether a variable should be used that is incremented during any connecting process. An integer variable that is initialized with 0 can be incremented up to 2,147,483,647 times. This means that over 2 billion event channels can be connected to an event domain. However, since it seems incomprehensible that more than 1,000 event channels should be connected at the same time. Assuming that the server that runs the management of the event domains will not be rebooted for two years, about 3 million event channels can be newly connected each day. However, it seems realistic to assume that the number of event channels within an event domain will remain in the 2- or 3-digit range. Assuming that 1,000 new event channels are connected each day, the implementation could run for more than 5,800 years. In the case that IDs that become available again shall be reused, this would require saving all free IDs in a list. During each connection process, this list has to be searched for the smallest ID. In the worst case, the result of the search would be that the smallest free ID would equal a value assigned by the incremented variable.

Each event domain is also assigned a unique ID by the event domain factory that creates it. In this case, for the reasons discussed above, it is even more worthwhile to use a variable that is incremented when creating an event domain.

Cycles: Before connecting two event channels through the method `add_connection`, it has to be checked whether the addition of this connection would cause a cycle. If cycles are not allowed, the output is the exemption `CycleCreationForbidden`, which consists of a sequence of all event channels that would have formed the cycle. If cycles are allowed, the connection is performed.

A directed graph is called strongly connected if for all vertexes it holds true that two vertexes always have a mutual connection. Although an event domain is not necessarily a strongly connected, directed graph itself, it can be divided into strongly connected components. Each strongly connected component equals a group of event channels that form a cycle or an individual event channel. This fact can be used in order to identify the cycles that exist within an event domain. If a new connection between two event channels shall be established and cycles are prohibited, it is checked whether this would create strongly connected components that consist of more than one individual event channel. If this is the case, the new connection would create a cycle and will raise an exception.

The method `get_cycles` shall consist of a list of all of the graph's cycle sequences. A cycle sequence consists of the IDs of all event channels forming the cycle. In order to find all cycles of a directed graph, it is not enough to find the strongly connected components. Cycles that are contained within the cycles remain undetected. In order to find these cycles, each strongly connected component has to be analyzed with regard to further cycles. Each connected component that contains more than two event channels can theoretically contain additional cycles. One could check now for all combinations whether it remains a strongly connected component even after removing one event channel. If this is the case, another cycle exists. This process would have to be continued until only individual event channels remain.

The brute force method for finding cycles is to search all combinations of event channels for cycles. The number of search runs potentially increases with the number of event channels, leading to an extremely long runtime. By using strongly connected components, multiple combinations can be excluded from the search, leading to a shorter runtime. However, even this improved solution represents a significant effort. Assuming 100 event channels form a strongly connected component, in the worst case, 100!-times double in-depth searches had to be conducted until all cycles would be identified. It is very questionable as to whether the use of the method `get_cycles` justifies this effort since only relatively few situations can be imagined in which cycles could be desirable.

Diamonds: According to the specifications, a diamond exists within each event domain if there is more than one ways of getting from one event domain to another. If diamonds are not desired, it has to be checked before each connection whether a diamond would be created. If this is the case, the exception `DiamondCreationForbidden` will be raised, containing several sequences that each represents a possible path. If diamonds are allowed to be formed, the connection is established in any case.

The method `get_diamonds` shall deliver a list consisting of all diamond sequences within the event domain. A diamond sequence consists of the IDs of all event channels that form the path from a start vertex to a target vertex. If there is more than one possibility how to send an event from Event Channel 1 to Event Channel 2, all possibilities must be output in a sequence form.

In order to find all diamonds of an event domain, a tree for the path from each starting vertex to each target vertex is created displaying the alternate paths. If this tree has branches, a diamond exists.

Subscription and offer channels: The method `get_subscription_channels` expects an event channel ID as a parameter and should provide a list that includes the event channel IDs of all event channels that can be reached from the event channel that is handed over as a parameter. An event channel can be reached if the directed graph includes a path to it.

If a path exists, meaning that the event channel can be reached from the event channel that is being handed over, the event channel that can be reached is a subscription channel of the event channel that hands it over. In order to save all the subscription channels of the event channel handed over as a parameter, an in-depth search starting from the event channels that hand over is to be performed. As soon as an event channel is found, it is included in the list.

If Event Channel A is a subscription channel of Event Channel B, it holds true that Event Channel B is the offer channel of Event Channel A. The method `get_offer_channels` is supposed to feed back a list of all offer channels of the event channel handed over as a parameter. In order to create this list, an in-depth search analogously to the method `get_subscription_channels` is used. However, in order to find all offer channels, the in-depth search has to run through the connections in the opposite direction. Therefore, before starting the in-depth search, the adjacency matrix representing the graph is to be transposed.

Tests: To test the functionalities of the software, the following test scenario was used among others: with our MED implementation, an event domain factory is created. With the help of the class Notification-Server, which uses the CNS, six event channels are generated and registered at the event domain. An untyped event domain is then created by means of the class `Testscenario`. No QoS properties are specified during that process, i.e., the default values are set and cycles as well as diamonds are admissible. Following, connections between event channels are constructed with `ClientType "ANY_EVENT"` and `NotificationStyle "Pull"`. Event channel 2 is appointed as the default supplier channel. One pull consumer is then connected to the event domain, a second is registered with event channel 4. Subsequently, three pull suppliers are registered; one is connected to event channel 3, one to event channel 5 and the last supplier is connected to the default supplier channel. The resulting configuration is shown in Fig. 2.

Pull consumer 1 now receives events sent by pull suppliers 1 or 2, since a path exists from event channels 2 and 3, respectively, to event channel 0. Pull consumer 2 receives events triggered by pull supplier 3. Further, it should be noted that pull consumer 1 will receive events from pull supplier 1 twice, due to the diamond <20> and <2310>. Pull consumer 2 will repeatedly receive all events sent by pull supplier 3 due to the cycle, we created intentionally.

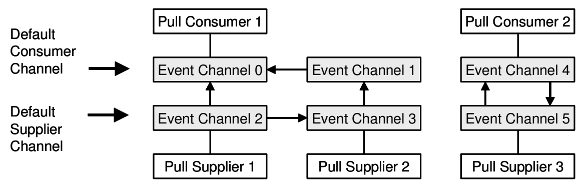

Fig. 2: A simple test scenario

Binding heterogeneous message services to mediator: In the realm of standardized Message-Oriented Middleware (MOM), Java developers have to decide in favor of one of two alternative specifications: CORBA Notification Service (CNS) or Java Message Service (JMS)[9]. If JMS is selected, development can be completely carried through in the "Java world." That decision may shorten the period of vocational adjustment and thus, reduce development time and cost. Should, at a later point in time, the necessity of integrating existing legacy systems into the current architecture become obvious, then this task can only be realized with increased efforts that will make the above mentioned advantages obsolete. On the other hand, developers can opt for the CORBA-based solution. Now, if they later find that integration of legacy systems is not necessary at all or only needed on a small scale, then the additional input would have been needless. Should it turn out that the initial decision has to be revised, then a bridge between the two messaging systems can facilitate protection of investment; those parts of the application that are already finalized could be utilized further on with the help of the bridge.

The MEDiator implementation is not limited to applications relying on different CNSs running concurrently. In the context of our implementation of a bridge between CORBA's Notification Service and the Java Message Service (JMS)[7] also JMS instances can make use of the MEDiator's functionality.

CONCLUSION

The aim to implement the MED specification in such a way that the third and highest level of standard conformity is realized was reached as planned: all modules of the specification were implemented.

While doing so, the main problem was the numerous flaws of the specifications that were listed in detail. It is surprising that still today only the first version 1.0 of the specification is available despite the fact that it was published approximately three years ago. It is difficult to comprehend why not even the most significant flaws were corrected and it can only be assumed that so far no greater need for an implementation existed. Since our solution shall be developed using open source technologies, further problems result from the lack of a completely implemented, freely available CNS and the lack of the equally unavailable CORBA Telecom Log Service. Most CNSs support exclusively an untyped communication and almost no ORB supplier offers an implementation of the CORBA Telecom Log Service. Also, the time-limited trial versions of most suppliers are limited and contain only selected CORBA Services.

However, event domains offer a comfortable extension of the CNS. Numerous methods make the administration of the often very complex topologies of event channels easier and allow getting an overview of the topology quickly. By using QoS properties, unwanted cycles and alternate paths can be avoided if necessary. Connecting event channels and connecting clients to event channels require some effort if CNS methods are being used. By adding an event domain, this can comfortably be conducted through only one method call.

Although the specification has some flaws at the moment, the usefulness of event domains is obvious and convincing. Therefore, it is actually surprising that within three years almost no supplier extended his CORBA Notification Service by this comfort.

REFERENCES

1. OMG, 2002. The Common Object Request Broker: Architecture and Specification Version 3.0. OMG Technical Document Number 02-12-06.
2. OMG, 2001. Event Service Specification. OMG Technical Document Number 01-03-01.
3. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, 1996. Pattern-Oriented Software Architectur-A System of Patterns. Chichester, John Wiley & Sons.
4. OMG, 2002. Notification Service Specification Version 1.0.1. OMG Tech. Doc. No. 02-08-04.
5. OMG, 2001. Management of Event Domains Specification 1.0.
6. OMG, 2003. Telecom Log Service Spec. 1.1.2.
7. Aleksy, M., M. Schader and A. Schnell, 2003. Design and implementation of a bridge between CORBA's notification service and the Java Message Service. Proc. of HICSS-36, IEEE
8. Cormen, T.H., C.E. Leiserson and D.L. Rivest, 2000. Introduction to Algorithms. Cambridge, MIT Press.
9. Sun Microsystems Inc., 2002. Java Message Service Version 1.1.