PAUL SCHERRER INSTITUT

SWISS LIGHT SOURCE

SLS-TME-TA-2004-0255
December 17, 2004

# An Event Service for the Propagation of Data

M. Böge, J. Chrin

Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland

Abstract

An event delivery mechanism based on the CORBA Event Service is described. Low-level hardware data are aggregated by event processing agents to produce complex events which supply summarized data to event channels for distribution to registered consumers, typically high-level software applications. The CORBA Notification Service, a recent extension to the Event Service, is also examined and potential enhancements to our event delivery system are identified.

# AN EVENT SERVICE FOR THE PROPAGATION OF DATA

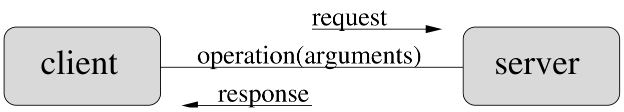
M. Böge, J. Chrin, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

*Abstract*

An event delivery mechanism based on the CORBA Event Service is described. Low-level hardware data are aggregated by event processing agents to produce complex events which supply summarized data to event channels for distribution to registered consumers, typically high-level software applications. The CORBA Notification Service, a recent extension to the Event Service, is also examined and potential enhancements to our event delivery system are identified.

## 1 INTRODUCTION

The accelerator device control system [1] at the Swiss Light Source (SLS) is based on the Experimental Physics and Industrial Controls System (EPICS) [2] whose communications protocol is Channel Access (CA) [3]. Application programming interfaces (APIs), such as the EZCA library [4] and the Common Device (CDEV) C++ class library [5], serve to hide the details of the CA protocol. A CORBA [6] interface to these APIs connects the controls system to the CORBA software bus used by developers of beam dynamics (BD) applications [7-10]. Access to the control system using the CORBA two-way communication model is the accustomed approach. This is illustrated in Fig. 1, wherein a CORBA Remote Method Invocation (RMI) is seen to take on the familiar appearance of a local function call.




```
C++:   object->operation(arg1, arg2, ...)
Java:  object.operation(arg1, arg2, ...)
Tcl:   $object operation arg1 arg2 ...
```

Figure 1: The CORBA request/response model

Situations exist, however, where the standard CORBA synchronous request/response exchange between client and server is not the optimal means of data transfer. One example is when a group of related devices, of interest to many clients, changes value. Each client would be required to either poll the server repeatedly for updated values or establish a (more involved) callback procedure. In such cases, data propagation is better served through an alternative, *reactive* form of programming, wherein clients are notified *en masse* of updated values. Such a delivery mechanism has been specified by the Object Management Group's (OMG) Common Object Services (COS) Specification Volume [11], in the form of the Event Service [12] and its extension, the Notification Service [13]. The latter, being a recent addition to the original OMG COS Specification, is yet to be implemented in our principal CORBA product, MICO [14]. Improvements to the event delivery mechanism introduced by the Notification Service specification are, nevertheless, addressed in this Note.

The OMG Event Service supports decoupled communication between multiple suppliers and consumers. This Note describes how low-level hardware events are aggregated to produce complex events that are propagated through the Events Service providing applications with a more personalized view of a given component of a control system. Indeed, much of the data on display in beam dynamics applications are received in this way. The Event Service has also been effectively employed for the monitoring of single hardware events.

## 2 EVENTS AND EVENT PATTERNS

Event driven systems are now commonplace in this era of information technology. A recent critique appears in [15]. They bring with them new terms and concepts, among which the notion of Complex Event Processing (CEP) is central. The principal components of CEP are briefly introduced here and their application in the BD environment is described in the subsequent section.

### *2.1 Aspects of an Event*

An event is an object that is a record of an activity in a system. The event signifies the activity and as such possesses a *form* and may optionally yield components such as *significance* and *relativity*.

The *form* of an event is an object. It may have particular attributes or data components, e.g. it may be as simple as a string or a tuple of data components, possibly including timestamps and other data pertaining to the event significance and relativity.

The *significance* of an event refers to the activity it signifies, while the *relativity* of an event describes its relationship with other events. The relationship between events is usually quantified in terms of *time, causality* and *aggregation*.

A *complex* event signifies an activity that takes place over a period of time and is an *aggregation* of other events which are referred to as its *members*.

Events in a distributed system occur in a relationship of dependencies or independencies. A set of events together with their causal relationship is called a *poset*, meaning a partially ordered set of events.

A *causal event execution* is a poset consisting of the events generated by a system and their relationship; it emphasizes the occurrence of events and their relativities online, in real time.

### 2.2 Complex Event Processing

Complex Event Processing (CEP) is central to any event-driven system. A complex event can signify an activity that consists of several activities in different parts of a distributed system. Conceptually, a complex event is regarded as an event at an higher level than the levels of its members. In our applications environment, a complex event often takes the form of an high-level event that is an aggregation of low-level hardware events, matching a certain event pattern. Event pattern rules and Event Processing Agents are the mechanisms for building CEP applications.

### 2.3 The Event Pattern

An event pattern is a template that matches certain sets of events. It describes precisely not only the events but also their causal dependencies, timing, data parameters and context. An event pattern is therefore a template for *posets*.

Associated with event patterns are *event pattern rules*. An event pattern rule is a reactive rule that specifies an action to be taken whenever an event pattern is matched. A reactive rule has two parts: a *trigger*, which is an event pattern, and a *body of actions*, which is an event that is created whenever the trigger is matched.

### 2.4 Event Processing Agents

An Event Processing Agent (EPA) is an object that monitors an event execution to detect certain event patterns. The EPA object is constructed from event pattern rules and local variables whose values determine its state. The EPA monitors its input to detect instances of the rule triggers and when a match is made executes the actions in the rule's body. As a result of executing a rule, the EPA changes both its local state variables and its output event. Events that are output depend on the class of the EPA [15]. Three proven types are:

- *Filter* - reduces event executions to relevent subsets,
- *Map* - aggregates and correlates events,
- *Constraint* - detects proper or improper behaviour.

The *event pattern map* is the class of most relevance to the work presented here. Maps use event pattern rules to aggregate a poset of events into high-level events and, as such, are the basis for defining relationships between sets of system-level events and higher level abstraction events. Fig. 2 shows a generic template for a map agent that aggregates causal sequences of events in its input, and creates an event consisting of a sequence of values that summarize the aggregated data. The next section describes an implemention of the EPA interface.

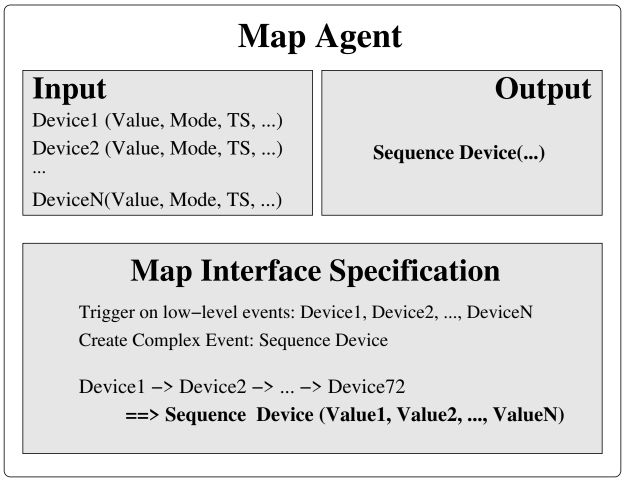



Figure 2: Event Processing Agent

## 3 CREATING COMPLEX EVENTS

The creation of complex events necessitates the aggregation of posets, an event pattern rule and an EPA. The newly formed complex event contains data that summarizes the aspects of the lower level events. Multiple views of a target system's activity can be constructed to run simultaneously, driven by the same lower-level events. Indeed, it is often the case that one cannot predict the most useful views in advance.

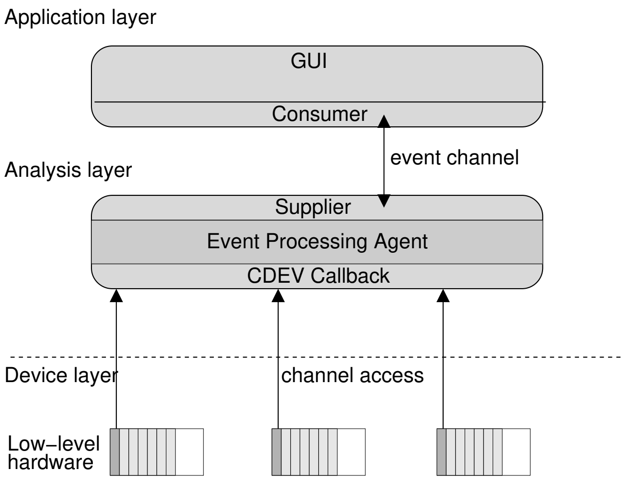



Figure 3: Data Aggregation and Propagation

Fig. 3 illustrates how control data are aggregated by an EPA to produce a complex event containing data that summarizes the aspects of lower level events. A typical EPA uses the CDEV API to establish a callback mechanism to the EPICS based control system. In this way, the relevant poset is aggregated. The EPA monitors its input to detect instances of the rule triggers. When a match is detected (i.e. the event pattern picks out the subset of events that signify that data transfer is complete), the agent executes the action of the rule's body, causing the EPA to change its local state variables and its output event. The rule creates a complex event with parameters that summarize the data transfer. The complex event is an higher-level event, giving only the data required by consumers (i.e. applications). Lower-level details, such as timestamps, are abstracted away.

EPAs exist for the aggregation of the various types of hardware devices, including Beam Position Monitors (BPMs) and various magnet groups, such that the corresponding event data provide a personalized view of a given component of the control system. Two examples are presented in the following.

### *3.1 EPAs for BPMs*

Separate EPAs exist for BPMs from the different accelerator facilities, namely the injectors, the booster and the storage ring. The principal task of these EPAs is to aggregate and analyze a specific set of BPM data, and to supply the summarized results to specific event channels serving various clients. These include the Tcl/Tk based orbit correction application [16] and the high-level Java application responsible for the analytical determination of the effects of the insertion devices on the closed orbit [17].

In addition, an EPA has been developed for the analysis of the entire contingent of BPM waveforms. For the storage ring, this constitutes 216 (sub-array) waveforms from 72 BPMs. Independent averages are calculated by the EPA, both over the complete waveform and over all waveforms for a given waveform element, to produce a complex event that is supplied to the specified event channel. This latter data is of relevance when the machine is operating in turn-by turn mode.

It is interesting to note that the viewing of these complex events at an high-level could be credited with the detection of anomalies that would otherwise not have been detected if only the low-level events were monitored. The onset of such an high-level anomaly would initiate a trace back procedure in order to locate the cause of the problem at the system-level.

### *3.2 The Tune EPA*

The computer-intensive calculation of the vertical and horizontal components of the machine tune parameter has also been incorporated into an EPA [18]. Posets from a dedicated Tune BPM are aggregated and when the event pattern rule is triggered (through a data transfer complete acknowledgement), the tune calculation is performed. A complex event is subsequently created, the form of which is a tuple of data containing a sequence of the measured tune values. The EPA additionally stores the full data complement, including that of the tune BPM waveforms, in virtual memory space, avoiding time consuming input/output operations. A dedicated Tune server provides methods that enable a client to both regulate input parameters to the tune calculation and to retrieve the full complex of results from the virtual data store.

## 4 THE EVENT SERVICE

In the OMG Event Service model, suppliers produce events and consumers receives them. Events are propagated through an event channel which acts as a mediator between the consumer and supplier. Communication is anonymous in that the supplier does not have knowledge of the receiving consumers. Event channels support different models of event delivery, the type of which depends on the collaboration between suppliers and consumers. This is illustrated in Fig. 4 which highlights the push and pull mechanisms established between the event channel and the supplier/consumer. The various push-pull permutations lead to the four event delivery models which our now outlined. References to Design Patterns [19] are made, where appropriate, following the convention of [20].

Fig. 4 further serves to emphasize that an event channel is able to fulfill all four roles simultaneously. Note that while the flow of events is always from supplier to consumer, the invocation of the method call, by which the event is transmitted, can be in either direction.

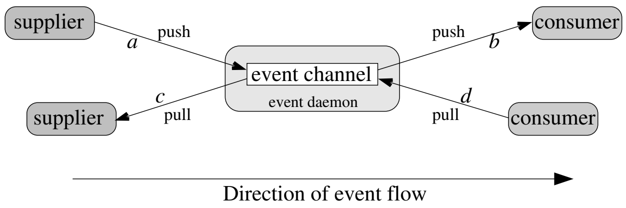



Figure 4: The Event Models

### *4.1 The Canonical Push Model*

The canonical push model is represented by path $a \rightarrow b$ in Fig. 4. Suppliers push events to the event channel, which subsequently forwards them to all registered consumers. The supplier is thus the ‘active’ initiator of events while the consumer is the ‘passive’ target. In object-oriented terminology, the event channel is said to play the role of *notifier* as defined by the Observer pattern [19].

### *4.2 The Canonical Pull Model*

The canonical pull model is represented by path $c \rightarrow d$ in Fig. 4. A consumer sends a pull request to the event channel which subsequently retrieves events from suppliers and delivers them to the requesting consumer. Here it is the consumer that is the ‘active’ initiator of events while the supplier is the ‘passive’ target of the pull request. The event channel is said to play the role of *procurer*.

### 4.3 The Hybrid Push-Pull Model

The hybrid push-pull model ('pushmi-pullyu'[1]) is represented by path $a \rightarrow d$ in Fig. 4. Suppliers push events to the event channel, while consumers pull events from the event channel at will. Here, where both the supplier and the consumer are active initiators of events, the event channel is said to play the role of *queue* as defined in the Active Object pattern.

### 4.4 The Hybrid Pull-Push Model

The hybrid pull-push model is represented by path $b \rightarrow c$ in Fig. 4. Here, the event channel pulls events from passive suppliers and delivers them to passive consumers. The event channel is said to play the role of *intelligent agent* as it is responsible for initiating the movement of events in the system.

### 4.5 Event Data Form

In practise, our environment is best suited to the Canonical Push model. An implementation of a push consumer is listed in Table 1. In this model, clients subscribe to the given event channel and passively receive data upon the occurrence of an event.

Event data is propagated through the Event Service system in the form of the *CORBA:Any*, a container for either built-in or user-defined data types that further holds a TypeCode which acts as a run-time identifier of the prevailing data type. The EPAs define a data structure into which they store an event message and then package their data structure into a *CORBA::Any*. For many of the event channels introduced in this work, an uniform data structure is used[2].

Table 1: Implementation of a push consumer in Java

```
class myPushConsumer extends
  org.omg.CosEventComm.PushConsumerPOA {
  public void push( org.omg.CORBA.Any any ) {
  \\ from IDL Module:  BDCDEV
  \\ typedef sequence<float> SeqFloat;
    if ( (any.type()).equivalent
                ( BDCDEV.SeqFloatHelper.type()) ) {
      float tuneSeq[];
      tuneSeq=BDCDEV.SeqFloatHelper.extract(any);
    }
  }
}
\\ Initialisation...
myPushConsumer impl = new myPushConsumer ();
byte[] oid = ("MyConsumer").getBytes();
\\ create POA from root POA
org.omg.PortableServer.POAPackage.POA poa=...;
poa.activate_object_with_id(oid, impl);
\\ Instantiate Consumer class
org.omg.CosEventComm.PushConsumer
    consumer = impl._this(orb);
\\ Resolve Event channel Name from Naming Service
org.omg.CORBA.Object objEventCh = ...
\\ Register As An Event Channel Consumer
org.omg.CosEventChannelAdmin.EventChannel channel
    = org.omg.CosEventChannelAdmin.EventChannelHelper
               .narrow(objEventCh);
org.omg.CosEventChannelAdmin.ConsumerAdmin
    consumerAdmin = channel.for_consumers();
org.omg.CosEventChannelAdmin.ProxyPushSupplier
    supplier = consumerAdmin.obtain_push_supplier();
supplier.connect_push_consumer( consumer );
```

### 4.6 Event Channels

The task of creating event channels has been separated from the EPAs which serve as the single data source to the event channels. Rather, a separate program that is initiated at server boot time takes on the responsibility for creating the event channels in the address space of the CORBA Event daemon. Object references to the event channels are exported to the CORBA Naming Service.

New event channels can be implemented on the fly by adding entries to a configuration file and restarting the event creation programme through a shell script that propagates the appropriate options.

Typically, event channel data is grouped according to accelerator components (e.g. storage ring, booster, injectors) and device type (e.g. magnets, BPMs). Table 2 lists a number of event channels to which various EPAs provide data. The channel names are self-describing, giving a good indication of their general content. An exact listing of the data being transmitted through each channel is given in [22]. Event channels (not shown in Table 2) also exist for propagating data from monitored hardware devices, specified directly by the user, and also from the TRACY accelerator model.

## 5 THE EVENT HORIZON

The CORBA Event Service implements a publish/subscribe application paradigm that provides for a natural programming style in which pertinent data can be passively received by any interested client application. Certain drawbacks nevertheless exist [21]. Publicized inadequencies include the necessity (at least for all ORBs known to the authors) to propagate event data under the auspices of type *CORBA::Any*, the absense of event filtering and the lack of explicit quality of service (QoS) control.

[1] "The pushmi-pullyu has no tail and a head at either end. In that way the pushmi-pullyu can talk while eating without being rude. The pushmi-pullyu people have always been very polite." From Hugh Lofting's, "The Story of Doctor Dolittle", 1920

[2] The event output class is actually wrapped in a struct, a legacy dating to the time of our first experience with Object Request Brokers (ORBs) [7]. Encapsulation of data into a struct was necessary in order to avoid a mis-interpretation of data when passing certain primitive types between various ORB implementations which, evidently, had not yet reached maturity

Table 2: Event Channels

| Prefix: `SLS:BdEvent:` | |
|---|---|
| **Booster/Injectors** | **Ring** |
| `LBinj:BPMwf` | `RI:BPMco` |
| `LBinj:MACHCV` | `RI:BPMcooco` |
| `LBinj:MAQ` | `RI:BPMwfHL` |
| `BO:MACHCV` | `RI:BPMwfprime` |
| `BO:MAQ` | `RI:MACHCV` |
| `BO:BPMco` | `RI:MAS` |
| `BRinj:BPMwf` | `RI:MAQ` |
| `BRinj:MACHCV` | `RI:TUNE` |
| `BRinj:MAQ` | |

These limitations however did not act as obstacles in this work. On the contrary, the restrictions imposed by the Event Service served to define the parameters of our design. Although the use of *CORBA::Any* is unavoidable, the choice of an uniform data structure at least results in a less complex and more robust transmission medium. To counteract the lack of event filtering, network traffic is minimized by connecting only a single supplier to a given channel. Clients interested in events from multiple sources thus register with multiple event channels. The absence of QoS is circumvented by imposing a queue length of a single event, an adequate figure given that only the most recent event is ever of interest.

Despite these simple design techniques, however, it is acknowldeged that the Event Service is eventually to be superseded by the Notification Service [13], which extends the Event Service in a manner that overcomes the drawbacks of the Event Service. In this respect, it is therefore worthwhile to look ahead to the enhancements that are to be offered and how they can serve to better our present event delivery system.

## 6 THE NOTIFICATION SERVICE

The limitations of the Event Service have been alleviated in the Notification Service[3] largely through the introduction of the *structured event type*, which provides a well defined data structure into which different event types may be mapped. Fig. 5 shows the format of a structured event. To illustrate how it serves in the capacity of event filtering and configurability, as defined by various QoS requirements, the data structure is examined in detail.

Each event is comprised of two main components: an header and a body. The event header consists of a fixed part and a variable part. The fixed header includes an event domain, event type and event name. The variable header holds a sequence of name/value pairs, where each name is a string and each value is of type *CORBA::Any*. While inclusion of these variable header fields is optional and their contents are unrestricted, the principal purpose of the header is to specify event-specific QoS properties. In this re-

[3]The Notification Service is a super-set of the Event Service

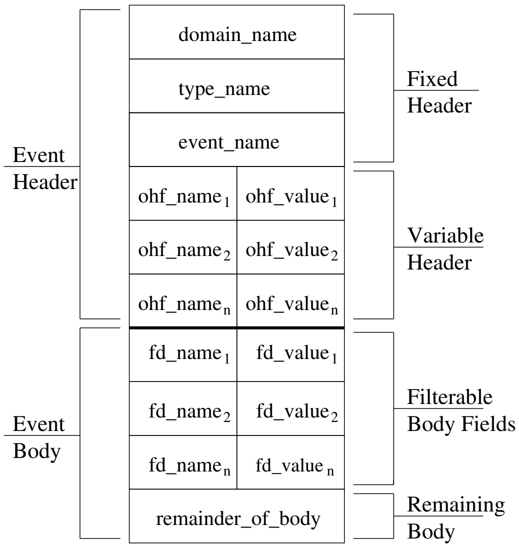



Figure 5: The structured event. The prefices ohf_ and fd_ respectively refer to ‘optional header field’ and ‘filterable data’.

spect, a set of standard optional header field (‘ohf’) names has already been defined by the OMG groupx along with the data types of their values. Table 3 summarizes a subset of ‘ohf’ names and the data types of their asscociated values.

Table 3: Standard optional header field

| Header Field | Type | Examplary Values |
|---|---|---|
| EventReliability | short | 0=BestEffort<br>1=Persistent |
| Priority | short | $-32767$=LowestPriority<br>32767=HighestPriority<br>0=DefaultPriority |
| Timeout | `TimeT` | 0=NoTimeout |
| OrderPolicy | short | 0=AnyOrder<br>1=FifoOrder<br>2=PriorityOrder |
| MaxQueueLength | long | 10 |

The second main part of the structured event is the event body which contains the contents of each event instance. The event body is itself comprised of two components, a filterable part and a remaining data part. As for the optional header fields (‘ohf’) of the event header, the filterable component of the event body (‘fd’) is similarly defined as a sequence of name/value pairs with each name being a string and each value of type *CORBA::Any*. The ‘fd’ fields are defined by the user and serve to act as a tag for filtering out data that are not required by the subscribing client application on a per-message basis. The remaining data part of the event header is of type *CORBA:Any* and is used to transmit the actual event data.

The filtering of unwanted events is accomplished through *filter* objects which encapsulate a set of constraints specified in the form of the extended Trader Constraint Language (TCL) [23]. A *filter* object may be attached to either the *proxy* object, to which clients ultimately connect, or the *administration* object responsible for creating the proxy interface. In the former case, only subscribers connected through that interface are subject to the filter constraint, while in the latter case, all proxies created through that administration object are affected.

Table 4: A structured event in Java

```
\\ Initialize orb
org.omg.CORBA.ORB orb
  = org.omg.CORBA.ORB.init(args, null);
\\ Event domain, type and name
CosNotification._EventType eventType
  = new CosNotification._EventType
                              ('BeamDynamics', 'Tune');
CosNotification.FixedEventHeader fixedEventHeader
  = new CosNotification.FixedEventHeader
                              (eventType, 'RI:TUNE');
\\ Variable Header, name/value pairs
CosNotification.Property variableHeaderSeq[]
  = new CosNotification.Property[1];
variableHeaderSeq[0]= new CosNotification.Property();
variableHeaderSeq[0].name
  = CosNotification.Priority.value;
variableHeaderSeq[0].value = orb.create_any();
variableHeaderSeq[0].value.insert_short( (short) 1);
\\ Filterable Data, name/value pairs
CosNotification.Property filterableDataSeq[]
  = new CosNotification.Property[2];
filterableDataSeq[0]= new CosNotification.Property();
filterableDataSeq[0].name
  = 'horizontalTuneAmplitude';
filterableDataSeq[0].value = orb.create_any();
filterableDataSeq[0].value.insert_float
                              ( (float) 0.00512 );
\\ Repeat for 'verticalTuneAmplitude'
\\ flterableDataSeq[1] = ...
\\ Event Data:  Horizontal and Vertical Tunes
org.omg.CORBA.Any data = orb.create_any();
\\ from IDL Module:  BDCDEV
\\ typedef sequence<float> SeqFloat;
float tuneSeq[] = new float[2];
tuneSeq[0] = (float) 20.421; \\ Horizontal
tuneSeq[1] = (float)  8.734; \\ Vertical
BDCDEV.SeqFloatHelper.insert(data, tuneSeq);
\\ Pack structured event
CosNotification.StructuredEvent event
  = new CosNotification.StructuredEvent();
event.header = new CosNotification.EventHeader
  (fixedEventHeader, variableHeaderSeq);
event.filterable_data = filterableDataSeq;
event.remainder_of_body = data;
```

Table 5: Specifying filter contraints using extended TCL

```
$domain_name == 'BeamDynamics' and
$type_name == 'Tune' and
$event_name == 'RI:TUNE' and
$priority > 0 and
($horizontalTuneAmplitude > 0.0015 ||
$verticalTuneAmplitude > 0.0034 )
```

Table 4 shows the incarnation of a structured event with domain name 'Beam Dynamics', type 'Tune' and event name 'RI:TUNE'. The filterable component consists of two $<$*name, value*$>$ pairs:
$<$'horizontalTuneAmplitude', horizontalTuneAmpVal$>$
$<$'verticalTuneAmplitude', verticalTuneAmpVal$>$

Table 5 examplifies use of the extended TCL to define the filter characteristics to be embedded into the filter object. Given this information, the event created in Table 4 would pass the filter imposed by the contraints given in Table 5 since the event belongs to the domain 'BeamDynamics', has both the correct type 'Tune' and name 'RI:TUNE', exhibits a positive non-zero priority and shows an amplitude for at least one component of the Tune value that passes its corresponding threshold value.

The structured event thus makes it possible to distinguish between the various event types. Suppliers or consumers can inform the event channel of event types either by directly registering the event type, or by attaching filter objects to the proxy and/or administration objects. In addition, since the Notification Service is always aware of the event types that are being either published or subscribed, it can act on this information to publish and transmit only those events for which there are registered consumers, thereby minimizing the network traffic. By comparison, suppliers in the Event Service implicitly continue to publish events even in the absense of interested clients. One purposeful and effective use of the Notification Service (as opposed to the Event Service) is for the dynamic logging of messages [24].

The efficiency and performance of the Notification Service (at least for a given implementation) would first need to be appraised. An important consideration is the speed with which event filtering and delivery is accomplished. How well large numbers of clients and servers are handled will also have to be evaluated.

## 7 CONCLUSION

The CORBA Event Service provides the event delivery mechanism for the propagation of aggregated low-level (and other) data to beam dynamics applications. Several Event Processings Agents (EPAs) that act as data suppliers to the CORBA event channels have been implemented. The EPAs are responsible for the capture of data from components of the low-level control system, their transformation according to predefined rules, and their subsequent delivery to event channels. These event channels are the primary source of information for users and are optimized to satisfy their reporting needs.

A first examination of the Notification Service reveals many notable features, including the ability to filter out unwanted data and to further publish and transmit only those precise events for which there are interested clients.

## 8 REFERENCES


[1] S. Hunt *et al.*, “Status of the SLS Control System”, PSI Scientific Report 1999, Volume VII, p. 32.

[2] EPICS, http://www.aps.anl.gov/epics/

[3] J. Hill, Nucl. Instr. Meth. A293 (1990) 352.

[4] N.T. Karonis, “EZCA Primer”, Argonne National Lab., http://www.aps.anl.gov/epics/extensions/ezca/index.php

[5] CDEV, http://www.jlab.org/cdev/

[6] OMG (CORBA), http://www.omg.org

[7] M. Böge, J. Chrin, “CORBA Objects for SLS Subjects”, SLS Note: SLS-TME-TA-2000-0162.

[8] M.Böge, J.Chrin, M. Munoz, A. Streun, “Commissioning of the SLS using CORBA Based Beam Dynamics Applications”, SLS Note: SLS-TME-TA-2001-0182.

[9] M. Böge, J. Chrin, “On The Use of CORBA in High Level Software Applications at the SLS”, SLS Note: SLS-TME-TA-2001-0183.

[10] M. Böge, J. Chrin, “Integrating Control Systems to Beam Dynamics Applications with CORBA”, SLS Note: SLS-TME-TA-2003-0225.

[11] OMG CORBA Services Specifications, http://www.omg.org /technology/documents/corbaservices_spec_catalog.htm

[12] Event Service v. 1.2, http://www.omg.org /technology/documents/formal/event_service.htm

[13] Notification Service v. 1.1, http://www.omg.org /technology/documents/formal/notification_service.htm

[14] MICO, http://www.mico.org

[15] D. Luckham, “The Power of Events”, Pub: Addison-Wesley, 2002.

[16] M. Böge *et al.*, “Orbit Stability at the SLS”, PSI Scientific and Technical Report 2004, Volume VI, p. 11.

[17] T. Schmidt, J. Chrin, A. Streun, D. Zimoch, “Feed-Forward Corrections for Insertion Devices at the SLS”, PSI Scientific and Technical Report 2004, Volume VI, p. 32.

[18] M. Muñoz, “Improvements to the Tune Measurement”, PSI Scientific and Technical Report 2004, Volume VI, p. 19.

[19] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.

[20] D.C. Schmidt, S. Vinoski, “Object Interconnections. The OMG Events Service”, C++ Report, Vol. 12, No. 2, 1997; http://www.cs.wustl.edu/˜schmidt/report-doc.html

[21] D.C. Schmidt, S. Vinoski, “Object Interconnections. Overcoming Drawbacks in the OMG Events Service”, C++ Report, Vol. 12, No. 6, 1997; http://www.cs.wustl.edu/˜schmidt/report-doc.html

[22] Event Channels, http://slsbd.psi.ch/˜chrin/

[23] Trading Object Service v. 1.1, http://www.omg.org /technology/documents/formal/trading_object_service.htm

[24] T. Modi, “Dynamic Logging and the CORBA Notification Service”, Dr. Dobb’s Journal, March 2001.