16

An Event Framework for CORBA-Based Monitoring and Management Systems

A. Schade
IBM Research Division, Zurich Research Laboratory
Säumerstrasse 4, 8803 Rüschlikon, Switzerland
Tel.: +41 1 724 84 05, Fax.: +41 1 710 36 08
email: san@zurich.ibm.com

Abstract

The focus of this paper is to describe an extensible framework for event class descriptions to be used in CORBA-based monitoring and management systems. The existence of a single root class from which all other event classes must be derived provides a common transfer format and thus promotes deploying typed event channels for the transfer of monitoring data. The introduction of this framework into a management architecture based on the CORBA interoperability platform by means of a meta object maintaining the event class information is illustrated. A specification language for event expressions derived from the General Event Monitoring language is presented and it is discussed as to how both the processing of these event expressions and the process of event generalization is simplified.

Keywords

Event Framework, Monitoring, Management, Distributed Applications, CORBA

1 INTRODUCTION

Management of distributed systems and applications is itself a distributed activity. Objects being managed reside on different nodes than the objects which observe their behavior and take appropriate actions to change it when necessary. This very high-level view already identifies two basic components of a management system: the managed system consisting of all objects being controlled and the managing system as a collection of facilities which carry out this control. Between these components information must be exchanged. Manager components within the managing system influence the behavior of *managed objects* by invoking management operations. The decision as to when which management operations are to be invoked is taken on the basis of *notification messages* emitted by managed objects. The behavior of managed objects is represented in terms of states and events indicating state transitions. Whenever an event of interest occurs within

J. Rolia et al. (eds.), *Open Distributed Processing and Distributed Platforms*

the managed system a corresponding notification message is generated which contains a characteristic set of attribute values according to the event type.

Due to the nature of distributed systems a number of problems arise as opposed to stand-alone applications. Components distributed over various nodes in a network may depend on different non-synchronized local clocks. Notification messages may not necessarily arrive at processing objects within the managing system in the same order as they have been sent by managed objects due to network delays, node failures or dynamic routing. In management systems these problems must be overcome to guarantee correct reporting of the system's behavior facilitating the appropriate control actions to be taken.

The frequency in which events occur and notification messages are generated upon detection of these events is very high for realistic applications. Along with this mass of notification messages a big amount of data is transferred. Manager components are only interested in a certain selection of notification messages according to their particular goal of management and the scope of the managed objects they are concerned with. Therefore, notification messages must be preprocessed before they can be evaluated by manager components to facilitate the triggering of management actions. This prep-rocessing not only reduces the number of notification messages but also combines and correlates them forming more abstract notifications. Thus the main objectives of event preprocessing are

- *event filtering* to avoid unnecessary network traffic and more important to present manager components only with those notifications they are really concerned with, and
- *event generalization* generating higher-level events by combination and correlation of low-level events.

Note that event preprocessing can be applied recursively on the way of notification messages to their respective destination. Objects which carry out event preprocessing are often separated from the manager components and reside at different locations.

The preprocessing of events is based on event expressions representing conditions on the order in which certain events are detected by the observing object. These conditions may be further constrained by so-called guards which are boolean expressions over attribute values contained in the notifications. In general, event expressions occur in two different places. The dissemination of notification messages by preprocessing components is based on event filtering conditions representing the *selection criteria* of subscribing objects. Subscription schemes of this kind are usually applied in management systems which contain various manager components with different management policies. The decision making is often based on *management rules* implementing a certain policy. The preconditions of the rules triggering the management actions again consist of event conditions as they also occur in the event preprocessing phase. The rule base, however, may be more sophisticated as conditions may require knowledge about the behavior of the managed system in the past.

From the process of event preprocessing as described above, requirements for an event subsystem supporting monitoring and management of distributed systems or applications can be derived. Rule bases and selection criteria are expected to reach a considerable level of complexity in realistic applications. Events should not be self-describing since in this case neither the syntax nor the plausibility of the selection criteria could be checked before run-time. Instead, the information necessary to check syntactical correctness and plausibility must be provided in a way independent of transmitted events in order to vali-

date management rules and event conditions before they are used at run-time. Therefore, this paper proposes an object-oriented framework of event classes based on which event conditions can be expressed and checked. A method is described by which event class descriptions are communicated in a distributed environment and a specification language for event expressions based on GEM (Mansouri-Samani et al., 1995) is derived.

The remainder of this paper is organized as follows. In section two a management architecture for CORBA-based application management is outlined in which the event class framework is to be implemented. The event model is presented in section three, followed by the description of the Event Specification Repository (ESR), a meta-object maintaining the event classes used in a management system. Based on this, in section five the GEM-based event expression language is illustrated. Section six then describes how notification messages can be evaluated applying the proposed event framework. As a result of this discussion, the interface definition of the ESR is finally derived in section seven.

2 THE MANAGEMENT ARCHITECTURE

The management architecture (Schade & Trommler, 1996) presented in this section is a synthesis of the CORBA approach for distributed object interaction and the functional capabilities of the OSI Model for Systems Management (ITU, 1992). Considering the communication structure in the OSI model it is evident that most of the enabling functionality is encapsulated in the management agent. Enabling functionality is mainly the set of services that have to be provided to support management communication using CMIP including protocol mapping, operation invocation, name resolution, and dissemination of events. We selected CORBA (OMG, 1996) as the basic communication platform for the management architecture since it is a widely spreading standard in which the event framework is to be implemented. It should be noted that the event framework as a model is independent of CORBA whereas the management architecture is not.

The majority of the services provided by an OSI management agent, like protocol mapping, data marshaling and target resolution, are either unnecessary in CORBA or already an integral part of the Object Request Broker itself. The deployment of CORBA for distributed application management will therefore result in a much simpler design of a management system consisting of smaller components. Like in the OSI architecture, *Managers* and *Managed Objects* remain parts of the architectural concept, though in a different manner than in OSI. The main difference is that management agents are no longer necessary. Only the processing and dissemination of event notifications must be supported externally to the Object Request Broker. For this purpose we introduce *Notification Handlers* (NH) as a new architectural unit providing the same function as the OSI Event Report Management Function (ITU, 1993). From and to NHs event channels are created through which notifications are forwarded using the push model as described in the COSS *event service* (OMG, 1995).

Managers, Managed Objects and Notification Handlers are equipped with standard CORBA interfaces. They communicate in terms of operations invoked at the interface of the respective target object using a corresponding object reference. Managers possess object references to the managed objects they are responsible for. The Dynamic Invocation Interface (DII) is used to keep manager components as generic as possible. Precise

knowledge about the attributes and management operations offered at the management interface is obtained using the CORBA interface repository. Through their management interfaces, MOs reveal their internal behavior by (change of) attribute values and emission of notifications representing the occurrence of events.

For each MO there is exactly one Notification Handler which receives these notifications and provides the first step of notification processing. The dissemination of notifications is implemented using a subscription model allowing managers to register themselves with Notification Handlers as target objects for certain types of processed notifications. A Notification Handler encapsulates an arbitrary number of Event Forwarding Discriminators (EFD). An EFD is created when an object subscribes for certain events by sending a corresponding event condition. The discrimination of notifications according to their type and contents is performed based on these filter conditions. The set of notifications which match a condition is forwarded to the respective subscribers. Aside from managers, these can also be Log Objects or other NHs, which makes it possible to cascade Notification Handlers providing different levels of notification processing.

A possible structure of a management system consisting of the three basic types of participating objects is shown in figure 1.

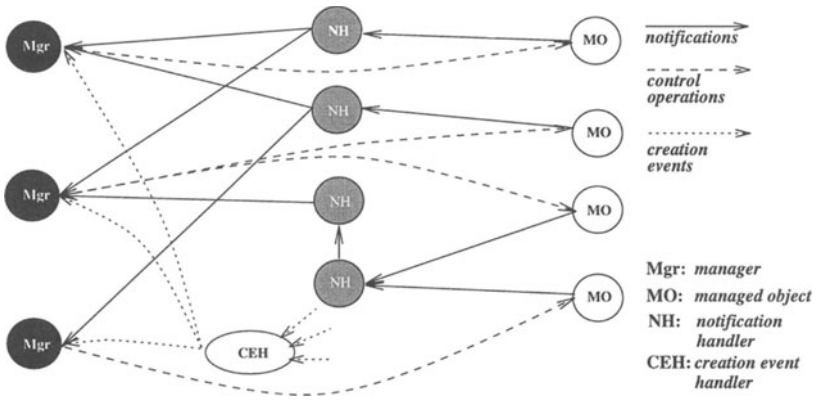

Figure 1 A CORBA-based Application Management Architecture.

In CORBA, clients can only submit requests to a server if they possess an object reference. Therefore, CORBA objects can only be created using the interface of an already existing factory object which then returns the new object reference. If the architecture is not to be limited to a *platform-centric* design (Yemini, 1993) with only one manager object, the object references of newly created instances must be provided to all interested managers (in terms of CORBA: to all instances which could potentially act as clients for the new objects). The problem of reference distribution also arises if auto-configuration (Hegering et al., 1994) is to be conducted, where managed objects can be incorporated originating from outside the management system. For this purpose a *Creation Event Handler* (CEH) has been introduced completing the minimal configuration depicted in figure 1.

New object instances register their name at the COSS *naming service* and cause a notification to be sent to the CEH informing all interested managers of their existence. This notification contains the name of the object, its interface type, the time of creation, and the object reference of the NH object which is associated with the created MO.* Additionally, the notification sent to the CEH can contain information about the creating factory object and the object triggering the creation which would provide the notion of ownership for security mechanisms. The CEH which belongs to the initial set of objects works similarly to a Notification Handler allowing managers to subscribe to creation events. With this approach there is no difference between objects created on behalf of the management system and *introduced objects* (ISO, 1995) created outside the management system.

3 THE EVENT MODEL

In this section an object-oriented event class framework is proposed for use within CORBA-based monitoring or management systems. In the literature, also other event frameworks have been elaborated (Hofmann et al., 1992; Marzullo et al., 1991; Spezialetti & Bernberg, 1995). They are also based on event specification languages, but lack an object-oriented design and hence do not provide the simplicity of specification and evaluation as the approach described in this paper. A comprehensive overview about traditional event frameworks is given in (Mansouri-Samani, 1995).

In an object-oriented framework, event classes specify the attributes contained in event messages of the corresponding event class type. They can be considered as classes without methods whose member variables are of CORBA-IDL types. The language used in this paper for specification of event classes adheres to the following syntax.

```
class          ::= ''class'' <name> [ base-type-spec ]
                   ''{'' { attribute }+ ''}'' '';''.
base-type-spec ::= '':'' <name> { '','' <name> }.
attribute      ::= <IDL-type> <name> '';''.
```

The framework is characterized by a single abstract base class, Event, from which all other events for notification messages are derived.

```
class Event {
        string  event_type;
        string  object_name;
        objref  origin;
        UTCtime time_of_occurence;
};
```

The definition of the class Event shows the generic attributes of notification messages common to all particular instances of event classes which can be derived from this abstract base class. Attributes assigned to each event may be of two kind: *Dependent* or

*New NH objects can be created in a similar manner by making their references available at the CEH. In this case there is no corresponding NH as for MOs.

Independent (Mohr, 1990). In contrast to dependent attributes which are determined be the particular type of the event, the common attributes contained in the base class are called independent. These are

- the event type, identifying the set of attribute sent with instances of this type,
- the registered name of the object originally sending the notification,
- its object reference, and
- the synchronized time at which the event responsible for the notification message occurred.

The class Event represents the root node of the inheritance tree established by subtyping to describe events emitted by "real" managed objects. Therefore event channels can be used as *typed event channels* based on the CORBA Event Service (OMG, 1995) using Event as the expected type for push supplier and push consumer interfaces.

An additional specification method for event classes is necessary since the concept of inheritance amongst structured types cannot be expressed in CORBA-IDL in which subtyping of interfaces is supported. This *Event Specification Language* (ESL) must be part of the management interface of each MO and is therefore integrated in the *Management Interface Definition Language* as introduced in (Schade et al., 1996) The set of all event classes available in the management system provides an object-oriented event framework.

The specification below shows an example of refining the abstract event class describing a type for events that could be emitted in the case of security violations, i.e., if access is denied for a particular operation at a certain interface.

```
class SecurityAlarmEvent : Event {
        objref  intruder;
        string  interface_name;
        string  denied_operation;
};
```

Besides the base class attributes three additional attributes values are conveyed with events of this type

- the object reference of the client attempting to invoke a protected operation
- the interface at which this happened, and
- the name of the operation for which access was denied.

The advantage of the described event framework is that events do not have to be self-describing. Self-describing events would be represented by name-value-pair lists specifying the name and the value of an event attribute. However, it would be impossible with such a method to apply any syntax and/or consistency checks for event forwarding conditions in Notification Handler components or managing rules in manager components. Applying the described event framework allows to check syntactical correctness and plausibility of these constructs. Whenever notification messages representing an instance of a particular event class are sent, the sending component creates them instantiating the corresponding event class and transferring it to the destined consumer.

4 THE EVENT SPECIFICATION REPOSITORY

The Event Specification Repository (ESR) (see figure 2) is a CORBA object which belongs to the initial configuration of the management system. It serves as a meta-object which contains the definitions of all event classes defined in the management system. The ESR (or more precisely its object reference) is known in advance to all other components of the management system that have to deal with either sending or receiving of notifications. Events available at the CEH indicating the creation of new objects are of type `Event`, the base class of all events. This is sufficient, since the only information which is relevant in the case of object creation is the name of the new object, its object reference and its time of creation.†

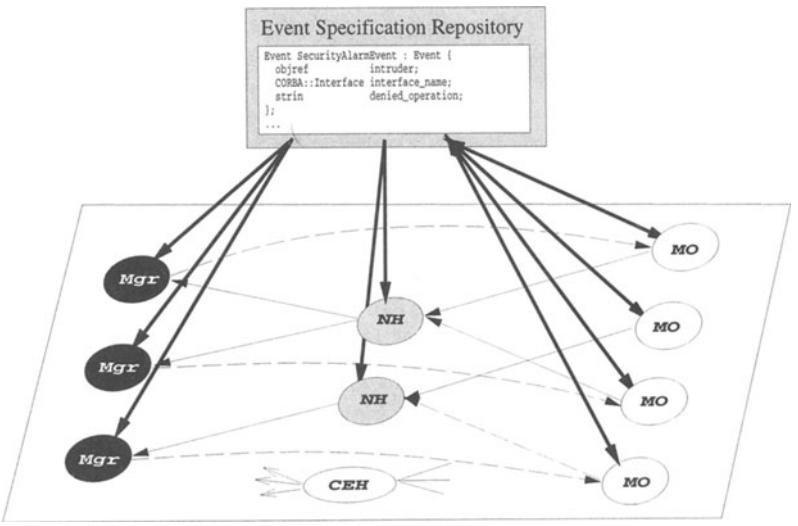

Figure 2 The Event Specification Repository.

Whenever an object of a so far unknown type is created and incorporated in the management system, the descriptions of the events it is able to send contained in the object's management interface specification are stored in the Event Specification Repository. This is done by a generic function `register`. Once the object has been registered in the management system, its event class definitions become available to every other object by querying the ESR.

With this approach event forwarding rules as well as management rules can be checked for syntactic correctness and plausibility at compile time of the rule. Note that this is not the setup time of the whole system since new objects can be included in the management environment at run-time, and rules coping with their events can be added to Notification Handlers and manager components. However, these checks can be applied before the first

†It could be argued that even a special event class should be introduced here. If so, it could contain attributes like the name of the factory object which has been used to create the object instance.

event is sent and processed. Thus, inappropriate rules can be rejected before actually using them.

Though the event class framework is object-oriented, the mapping process of the event transfer representation (using the abstract Event type) to the real type of the event is a *resolution* rather than a *downcasting* process. The target type is by definition a subtype of Event but it is not necessarily known at compile time of the NH. It may become known later when a new MO is registered. As a consequence, rule and filter conditions can be loaded dynamically whose event conditions refer to event classes of already known MOs.

5 EVENT FORWARDING RULES

In this section event forwarding rules are examined as an example of using event expressions which similarly occur in management rules. For each Event Forwarding Discriminator within a Notification Handler there is exactly one event forwarding rule which implements the decision to be taken by the EFD. When an object subscribes to certain notification messages by sending the respective rule to the NH, this rule is parsed and an EFD implementing this rule is created.

An event forwarding rule consists of a left-hand side representing an event expression, and a right-hand side specifying the target which the event should be forwarded to if the event condition is true. A target can either be a manager component which subscribed to notifications establishing the EFD, or a Log Object, or another Notification Handler. Additionally the right-hand side of an event forwarding rule can contain statements indicating what kind of event should be forwarded and which values should be assigned to its attributes.

The left-hand side of an event forwarding rule is the actual event forwarding condition. An event forwarding condition is specified as an event expression whose syntax has been derived from GEM, the *General Event Monitoring Language* specified in (Mansouri-Samani et al., 1995). Assuming the (rather unlikely) existence of a global synchronized clock, GEM allows to formulate event expressions over the order of incoming events further constrained by guards which are boolean expressions over event attribute values. For the specification of event ordering conditions GEM provides five event composition operators

- **e1 & e2** – *conjunction*: occurrence of both e1 and e2 irrespective of their order;
- **e1 ; e2** – *sequence*: e1 followed by e2;
- **e1 | e2** – *alternative*: occurrence of e1 or e2;
- **! e1** – *exclusion*: no event e1, i.e. e1 ; e2 ! e3 occurs when e1 occurs followed by e2 but not e3 in between;
- **e1 + <time-period>** – *delay*: defines a certain point in time, <time-period> units after the occurrence of e1.

When monitoring a distributed system it is not always possible to assume that the order in which monitoring messages are collected reflects the order in which they have been sent. Therefore, it is necessary to re-order the messages to present observing objects with a consistent picture of the system behavior. Given the existence of a global clock or a set of synchronized clocks which is assumed in GEM this task is easy to solve, since

the time of occurrence attribute available for each event can be used for comparison. This reconstruction has to be carried out at run time in management systems.

The period of waiting for the arrival of a certain message must be limited. As opposed to the usual introduction of a maximum tolerated delay, GEM uses the different approach of an event specific delaying technique. In GEM delays are dealt with at two complementary levels. A detection window is assigned to each rule which maintains an ordered hierarchical event history over the period of time defined by the window. The detection window, thus specifies the maximum tolerated delay the rule can cope with. Additionally, scheduled time events can be used as explicit terminator events which allows to define a maximum delay relative to the specific event.

Event expressions can also be evaluated by GEM if a set of loosely synchronized clocks is used. Given a clock synchronization as it typically exists in work station clusters, the event re-ordering capability depends on the preciseness of the clock service available. For systems which generate a big amount of monitoring events in parallel this preciseness might be to small.

In absence of a global clock, re-ordering of messages is still possible to the extent of preserving casual relationships between events. In this case, event re-ordering techniques such as the Vector time method (Mattern, 1989) can be applied which is an extension of Lamport's well-known Logical clock method (Lamport, 1978). Each process is assigned its own clock which is updated as internal events occur. The set of all clock values forms a vector which is known to each process. The other clocks are updated when a received message has a specific clock value which is higher than the corresponding local value. Two events a and b timestamped with the vectors v_a and v_b respectively are causally related ($a \rightarrow b$) if for each vector position i holds $v_a[i] \leq v_b[i]$ (or $v_a[i] \geq v_b[i]$ for $b \rightarrow a$). Otherwise the two events are independent of each other.

The problem with this approach is that all messages, i.e. also management operation requests and normal operational messages, passed between components of the distributed system must be time-stamped. Is is very questionable if the quality of monitoring systems is suitable for realistic management purposes in which some casual dependencies between events might not be discovered. The usual approach taken to perserve casuality is to introduce an additional layer in the communication protocol or to augment the stub functions adding the time vector to each message header. In any case, the extension of the normal communication mechanism introduces additional overhead which must be weighed up against the cost of providing a sufficiently precise time service.

To implement this scheme, the framework root event class has to be modified. The attribute representing the time of occurrence must be replaced by a sorted list of pairs of process identification (original object) and corresponding clock value. A list instead of a vector is necessary to cater for a variable number of objects.

Considering the problem of event abstraction, new instances of event classes must be created upon reception of low-level events by EFDs according to the event forwarding rule. Event abstraction can be explained using a layer model in which the most specific events are emitted by objects on the lowest layer. On their way up to the upper-most layer representing the finally receiving objects which do not forward events anywhere else, events from lower layers can be collated producing more abstract events received by the next layer. In this sense, generalization/specialization implied by inheritance relationships between event classes is orthogonal to event abstraction according to the layer model.

The concept of multiple inheritance provides a simple way of constructing intermittently generated, more abstract events. Inheritance relationships between classes imply containment relationships between instances of these classes. The following example illustrates how this helps to simplify event construction.

According to the following event forwarding rule

```
e1 ; e2 when (e1.origin == e2.origin and e1.object_name == "MO1")
  ==> MgrX as Sample e (e1,e2)
      {
         e.text = "Sample event combined from e1 and e2";
      }
```

an event `e` of type `Sample` is sent to the subscribing manager component `MgrX` with its `text` attribute set accordingly if the two events `e1` and `e2` have been received from `MO1`. If the generated event class inherits from one or more condition event classes, the attributes of the generated event can be assigned in the same way as constructor invocations for superclasses in C++. In the example, `e` is assumed to be an instance of an event class `Sample` derived from two event classes `e1` and `e2`. Though there are no explicit event class constructors in the event class framework, attributes of event instances can be assigned by specifying instances of supertypes whenever instances of subclasses are built. Additionally `e`'s own attributes can be set using an attribute assignment clause (see figure 3).

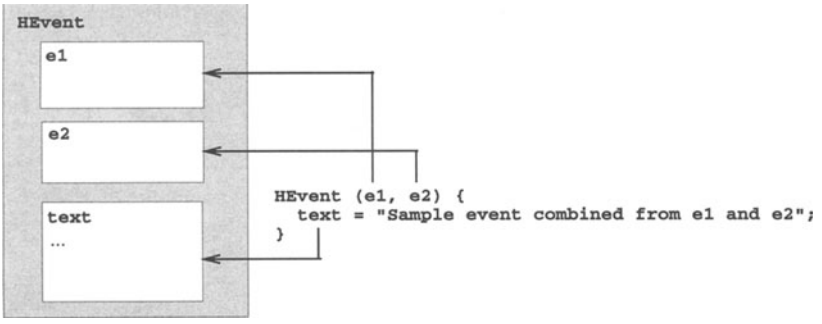

Figure 3 Construction of higher-level Events.

6 EVENT CLASS RESOLUTION

This section describes the event class resolution applied when a notification in the transfer format is received by a processing object. The process of sending and resolving notification messages using the proposed event class framework via the Event Specification Repository is illustrated in figure 4.

Part of the registration of a new component being introduced in the management system is the declaration of the events it is able to emit in terms of notification messages. This declaration is done by sending the event classes defined in the management interface to

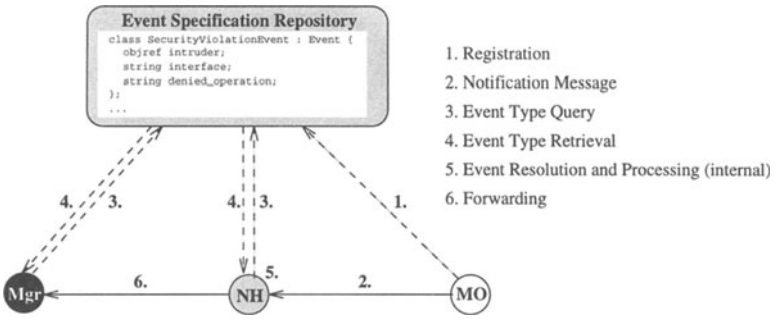

Figure 4 Processing of Notification Messages.

the ESR (1). Whenever a processing object (Notification Handler, manager component, etc.) receives an event of a certain type for the first time (2), it retrieves the event class description from the ESR (3,4). With this information it is capable to determine the attributes of the actual notification message after resolving the transfer representation `Event` to the real event class according to the event type attribute contained in the notification (5). The interface which must be provided by the ESR meta object to support the described method is presented in the next section.

The Event Specification Repository contains meta information, i.e., information about classes describing the way notification messages will be structured. If a hierarchical management architecture is applied, it is possible to keep this information inside a domain which is a building block of the management hierarchy. Such a building block will consist of one ore more manager components being responsible for a set of managed objects within the domain. The domain itself will provide a management interface to its environment to which other higher-level managers can be connected. These domains encapsulate their contents in the sense that neither the managed objects nor the exchanged notification message types (event classes) will be known to the outside world. Thus, there must be one Event Specification Repository per domain.

7 EVENT SPECIFICATION REPOSITORY INTERFACE

Having described both the properties of the Event Specification Repository and the evaluation of event expressions in Notification Handlers and manager components we can finally derive the interface the ESR offers to the management environment.

The Event Specification Repository object implements two operations. `RegisterEvent Classes` is used by event sender objects to make their event classes available to receiving objects in the system. The second operation `RetrieveEventClass` will be used by receiving objects to resolve abstract event representations in the transfer format to their actual event classes upon event reception.

```
// IDL
interface EventSpecificationRepository {

  typedef struct attribute {
        string        attribute_type;
        string        attribute_name;
  } Attribute;
  typedef sequence<Attribute> AttributeList;

  typedef struct event_class {
        string        event_class_name;
        NameList      parent_classes;
        AttributeList attributes;
  };
  typedef sequence<EventClass> EventClassList;

  void       RegisterEventClass (in EventClassList event_class_list);
  EventClass RetrieveEventClass (in string event_class_name);
};
```

Event classes are stored in terms of their attributes and parent classes. The event class name (type identification) is used as a search key in the Event Specification Repository. The inheritance relations must be preserved in order to allow checking of higher-level event generation in event forwarding rules as described in section 5.

8 CONCLUSION

In this paper an object-oriented event class framework for monitoring and management systems has been presented. It has been shown how this concept can be applied in the context of CORBA-based event communication using event channels. The framework is characterized by the existence of a single root class Event from which all event classes for real-life events must be derived. This abstract base class only specifies the properties which are common to all events used in a management system or for monitoring purposes, that is, name and object reference of the event origin and the time of the event occurrence. The root class Event is also used as the base type for typed event channels based on the CORBA event service.

In the CORBA-based management architecture (Schade & Trommler, 1996) the Event Specification Repository has been introduced. This meta object maintains all event classes which can be used by event sender objects and thus provides the information necessary to evaluate event forwarding rules in Notification Handlers and trigger conditions for management actions in rule-based manager components. Event forwarding rules are based on event conditions representing expressions over the order of detected events and the values of their attributes. A specification language for event expressions has been derived from GEM (Monsouri-Samani et al., 1995) to express these event conditions. Its use has also been discussed in the paper. Finally, a method of registering event classes used by additionally introduced managed objects has been presented. One of the key requirements

for application management is to keep the management system open and extensible. This is achieved by the method of registering event classes used by additionally introduced managed objects has been presented.

The main advantage of the described event class framework is that the efficiency of the monitoring process in management systems can be improved because notification messages do not need to be self-describing. Such notification messages are based on events which would merely consist of a list of attribute names and their corresponding values. With the event framework presented in this paper event attributes can be standardized leading to an extensible set of pre-defined event classes. The knowledge of the event classes thus allows to check the syntax and the plausibility of event forwarding conditions and management rules at the time of their compilation before any notification messages are actually processed.

In the future, it must be investigated to what extent pre-defined event classes for standard notification messages can be provided for application management. The general idea is to initialize the Event Specification Repository with a number of standard event classes describing the contents of basic notifications which occur in all management systems independently of their particular purpose. These event classes can then be refined and customized according to the special needs of the management system which accelerates and simplifies the adaptation process.

9 REFERENCES

Hegering H.-G. and Abeck S. (1994) Integrated Network and System Management, Addison-Wesley Publishing Company, Workingham (England).

Hofmann, R., Klar, R., Mohr, B., Quick, A. and Siegle, M. (1992) Distributed Performance Monitoring: Methods, Tools, and Applications. University of Erlangen-Nuremberg, Germany.

ISO (1995) ISO/IEC IS 10746-2 Reference Model for Open Distributed Processing - Part 2: Foundations. ISO/IEC JTC1/SC21/WG7.

International Telecommunication Union (1992) Data Communication Networks – Information Technology – Open Systems Interconnection – Systems Management Overview. ITU Recommendation X.701, Geneva.

International Telecommunication Union (1993) Data Communication Networks – Information Technology – Open Systems Interconnection – Systems Management: Event Report Management Function. ITU Recommendation X.734, Geneva.

Lamport, J. L. (1978) Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, **21**(7), 558–65.

Marzullo, K., Cooper, R., Wood, M. D. and Birman, K. P. (1991) Tools for Distributed Application Management. *IEEE Computer*, **24**(8), 42-51.

Mattern, F. (1989) Virtual time and global states of distributed systems. in *Parallel and Distributed Algorithms*, 215–26. Elsevier Science Publishers B.V. North-Holland.

Mohr B. (1990) Performance Evaluation of Parallel and Distributed Systems, in *Proceedings of the CONPAR 1990 - VAPP IV*, Springer Verlag Berlin, 176–87.

Object Management Group (1995) The Common Object Request Broker: Architecture and Specification. OMG Document PTC/96-08-04.

Object Management Group (1996) CORBAServices: Common Object Services Specification. OMG Document 95-03-31, upd. 96-07-15.

Mansouri-Samani M. (1995) Monitoring of Distributed Systems. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing.

Mansouri-Samani M. and Sloman M. (1995) GEM – A Generalized Event Monitoring Language. Imperial College Research Report No. DoC 95/8, Imperial College, London.

Schade A. and Trommler P. (1996) A CORBA-Based Model for Distributed Application Management. *Proceedings of the 7th IFIP/IEEE Workshop on Distributed Systems: Operations & Management*, L'Aquila.

Schade A., Trommler P. and Kaiserswerth M. (1996) Object Instrumentation for Distributed Applications Management. in: *Distributed Platforms* (eds. Schill A., Mittasch Ch., Spaniol O. and Popien C.), Chapmann & Hall, London.

Spezialetti, M. and Bernberg, S. (1995): EVEREST: An Event Recognition Testbed. in *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, IEEE Computing Society, pp. 377–85.

Yemini Y. (1993) The OSI Network Management Model. *IEEE Communications Magazine*, **31**(5), 20–9.

10 BIOGRAPHY

Andreas Schade received his diploma (Dipl.-Inf.) in computer science in 1994 from Humboldt-University Berlin, Germany. He is a Research Staff Member at the IBM Zurich Research Laboratory, where he has been since completing his studies. His research interests are distributed object-oriented systems, distributed applications and their management.