Object Instrumentation for Distributed Applications Management

A. Schade, P. Trommler and M. Kaiserswerth
IBM Research Division, Zurich Research Laboratory
Säumerstrasse 4
8803 Rüschlikon
Switzerland

Published in:
Schill A., Mittasch Ch., Spaniol O., Popien C. (eds):
Distributed Platforms, Chapmann & Hall, London, 1996.
pp. 173-185

Abstract

The development of complex distributed applications running on heterogeneous computer networks raises the strong demand for effective observation and control, i.e., for the management of the entire system behavior. Distributed systems create a new class of requirements especially in the areas of installation, consistent configuration, fault detection, and management exceeding what is necessary for stand-alone applications.

In this context more flexibility and generality is required for both the management system and the instrumentation of the application components than that provided by existing network management systems. This paper proposes a method to support the development of manageable distributed applications based on a formal management interface definition and an instrumentation library. This method is also applicable to already existing components of distributed applications provided their source code is available.

Keywords

Distributed Applications; Systems Management; Interface Definition; Instrumentation

1 INTRODUCTION

With the growing number of distributed and networked applications running on heterogeneous computer networks with a large number of components, the aspect of observing and controlling the complexity of these systems is increasingly important. Such distributed

applications create a new class of requirements especially in the areas of installation, consistent configuration, fault detection, and management exceeding what is necessary for stand-alone applications. Addressing these requirements in a general way, one can make no *a priori* assumptions about the internal structure and behavior of the managed system components.

In this paper we will concentrate on the aspect of making components of distributed applications manageable by enabling observation and control. It should be emphasized that there is in principle no difference from the management point of view whether an application component was planned to be part of a distributed system from the beginning, possibly using some interoperability environment such as OMG's OMA or OSF's DCE, or whether it was developed as a stand-alone application that became part of a networked system due to later integration. As in existing network management approaches, the management system is presented with an abstract view of the application components to be managed.

We will focus on how the distributed pieces of these applications may be integrated into a general management system. The paper presents an approach for the common and flexible instrumentation of managed components that make up a distributed application. This approach requires that the source code of the respective components is available.

Many projects in the area of distributed applications, which consist of multiple communicating components, show a strong need to examine and affect the behavior of the system. Most management systems follow a *platform-centered* (Yemini, 1993) model having a central manager as the focal point for control and observation of the entire system. As this is not the most flexible approach, recent research promotes architectures in which management functionality can be distributed. The solution presented in this paper is independent of the employed management architecture, be it platform-centered or more advanced and distributed.

The management of arbitrarily distributed applications requires flexible methods of adapting the managed components to the management system. In this context, *adaptation* means how an existing component is instrumented to reveal its internal behavior and how this information is provided to the object's environment. The managed system itself and the process of instrumentation, i.e., how application components are made manageable, should address the following requirements:

- *Generality* – No restrictions are imposed on the type of applications that can be bound to the management system. There must be a common way of binding arbitrarily behaving application components.
- *Extensibility* – It must be possible to introduce new objects to an existing system, to register and manage them homogeneously and dynamically without having to

  – reconfigure the management system or
  – extensively modify the newly managed object.

- *Flexibility* – There must not be any limitations concerning the topology of the distributed application. The initial distribution of its components can be modified by migration of such components. Such behavior can be controlled by the management system.

The basic issues concerning the structure of and the communication flow in the management system will be outlined in section 2. We will then discuss in section 3 how the behavior of a managed object can be represented in terms of its interface description. In section 4, the management interface components and possible implementation approaches in the form of source code instrumentation are presented. Finally, in section 5 we give a summary and present an outlook on further work.

2 MANAGED OBJECTS

The management process is comprised of two activities: observation and control. Observation, often also denoted as monitoring, means retrieval and preprocessing of information about the current behavior of the managed system components. These data form the basis upon which the controlling system enriches its knowledge (Park et al., 1994) about the managed system and takes decisions as how to mesh with its activity. The intervention is exercised in terms of management operations that modify the behavior of the affected managed components. From this very high-level point of view, management can be illustrated as a *continuous*, *closed-loop activity* (Hoffner, 1993) (see figure 1) based on a bidirectional communication between managing and managed system.

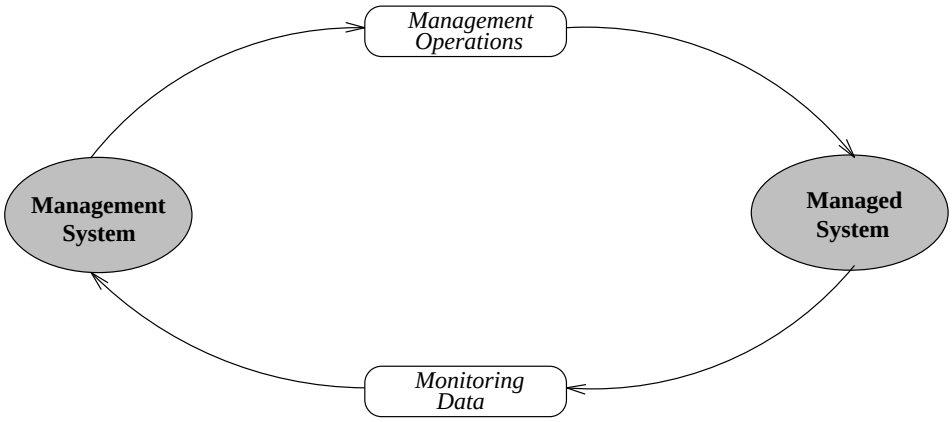

Figure 1 The Process of Management

This communication is realized in different ways in existing network management approaches like OSI and SNMP, and in interoperability environments such as CORBA and DCE which do not aim at management specifically. Regardless of which approach is ultimately chosen, the managed components must be equipped with:

- a management interface at which the controlling operations will be invoked and
- special program code, so-called *probes*, that generates notification messages and that implements the actual management operations.

This extension of the managed components is called *instrumentation* (Mansouri-Samani et al., 1993). The management interface provides for an abstract representation of the encapsulated managed entity which facilitates the transparent handling of heterogeneous hardware and software components. This notion leads us to a more direct definition of

managed objects as it is given in the OSI model (ITU, 1992) or in SNMP (Case et al., 1990).

A managed object is a controlled entity which is (a) equipped with a management interface and (b) instrumented with auxiliary management software probes.

Aside from its management interface a managed object must still provide the ordinary operational interfaces (also often referred to as *primary interfaces*) of the encapsulated component. A schematic illustration of a managed object is shown in figure 2.

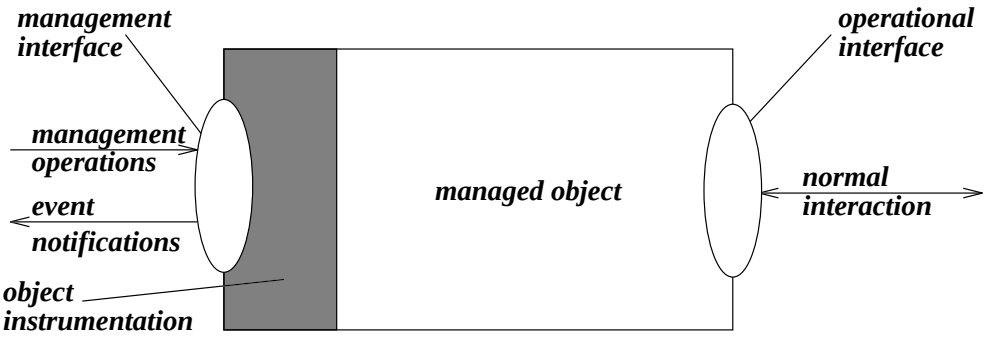

Figure 2 Structure of a Managed Object

There is an ongoing debate in the literature whether a managed object should offer more than one management interface (Rubin et al., 1994). The only justification we could find for multiple management interfaces is that they could provide different views, i.e., different levels of abstraction of the same managed object. Multiple management interfaces can thus be used to ensure compatibility in an evolving environment.

As mentioned above, communication via the management interface is bidirectional. Management operations are submitted to the object and monitor information becomes visible at the management interface.

3 MANAGEMENT INTERFACE DESCRIPTION

Because the behavior of a managed object is revealed in terms of states and events and is influenced by means of management operations, these items must be defined by an interface description. In contrast with other approaches to distributed object interaction, object management imposes somewhat different requirements. For example, in CORBA (OMG, 1993b), objects (i.e. service providers) are referred to by an object reference. This reference has a number of properties, so-called *transparencies*, which means that all intrinsics, such as the location of the object and the possible contributors in the provision of the service, are encapsulated and invisible to the service requester. In a management environment, the hiding of such information is not a suitable approach for interface handling. The management interface of an object is a description of how this object can be managed; it reveals the object's behavior explicitly and provides a certain view of the managed object.

Our definition of a management interface is therefore an extension to interface description languages known from CORBA-compliant systems and DME (cf. OSF, 1994 and Hegering et al., 1994). It is a first step towards the notion of an *interface* as defined in

the Open Distributed Processing Reference Model. Such an *interface* is an abstraction of the behavior of an object (ISO/IEC DIS 101746-2, 1995). The purpose of this section is to describe to what extent a management interface represents an abstraction from the managed application component. Our proposed *Management Interface Description Language* (MIDL) consists of three elements: *attributes*, *state definitions*, and *management operation* specifications.

3.1 Attributes

All attributes specified in the management interface description denote state variables of the managed object. Together they represent the attribute vector of the managed object. A single attribute is specified by:

- name,
- type,
- an informal description, and
- a tag indicating whether the value of the attribute can be modified, or only queried, by the manager.

3.2 Object States

States used in MIDL represent an abstraction of the ordinary state concept of managed objects. States subdivide the life cycle of the managed object into disjoint periods of activity. The current state is therefore the period of activity the object has recently entered. These periods are denoted by state names specified in the interface definition. Their specification is merely a set of identifiers that depend on suitable instrumentation of the managed object. This instrumentation is done by annotating the object code with calls to the *Management Adaptation Library* described below. The union of all periods of activity identified by the states defined in the management interface description is equal to the union of all possible instances within the definition range of the attribute vector.

3.3 Management Operations

Management operations, the third element of MIDL, are defined in terms of their signature, i.e., their name, the attributes whose values they affect, and the types of their parameters. As management operations can be considered remote procedures invoked by the manager, their execution semantics are similar to those of common remote procedure calls (RPC). Two methods are identified in CORBA (OMG, 1993b):

1. *at-most-once*. This specifies a synchronous call paradigm where the caller synchronizes with the callee upon receipt of the call's return.
2. *best-effort*. This method represents an asynchronous paradigm. It is not specified when or whether the call returns.

Aside from synchronous and asynchronous service requests, CORBA also introduces the so-called *deferred synchronous* semantics. Upon submission of the call, the caller obtains

a handle to be used later to inquire about the completion state of the operation (OMG, 1993a). For further distinction, we will refer to this as the *actively deferred synchronous call* method.

How do these RPC semantics relate to management operations and their semantics? In our model, management operations can be classified into *non-critical* and *critical* operations. *Non-critical* operations are executable at any time, regardless of the state the managed object is in. They either do not change the object's attributes, or they never cause inconsistency or loss of data.

The execution of a *critical* operation, however, is only allowed when the managed object is in a certain state. For example, instructing an object to reinitialize while it is carrying out a unit of work may cause that work to be lost, which often will not be the desired effect of the management operation.

In our approach, non-critical operations are represented as synchronous calls. Critical management operations, however, must follow the deferred synchronous paradigm, which we call *passively deferred synchronous operations*. As such, operations may be executed only when the managed object is in a certain state, the operation remains pending after it is submitted if the object is not in a permissible state for that operation. We decided to queue exactly one operation per managed object because the order of execution will be important and can be preserved in this way. If an operation is pending, other critical (but not immediately executable) operation requests will be ignored. However, all non-critical operations remain available since the possibility of retrieving information about the object must not be restricted.

Implementing *passively deferred synchronous operations* implies that there must also be a mechanism to abort a pending management operation that cannot be performed due to the current object state. Otherwise the object would be unmanageable if it never reached a permissible state for the pending operation as the result of a configuration error, for instance.

Critical operations are defined by specifying constraints as a set of states associated with the definition of the actual operation. These states then represent the periods of activity during which the management operation can be performed. A management operation is specified by:

- name,
- a list of affected attributes and return/exception parameters,
- an informal description of the purpose of the operation, and
- execution constraints in the form of a list of allowed object states.

A common problem in toolkits for building distributed applications is the lack of means of specifying object interface semantics. CORBA, for instance, states explicitly that no operational semantics are specified in an interface description (OMG, 1993b). For the purpose of management, we decided to include at least an informal descriptive part in the specification of the attributes and the management operations.

3.4 Inheritance

Inheritance plays an important role in the OSI model for systems management. Classes for management objects (*Guidelines for the Definition of Managed Objects* GDMO) (ITU,

1992c) can be defined to represent a prototype for a concrete instance of this object type. These templates must be registered in the ISO naming tree for management. Name bindings for each instance of such a template form the *Management Information Tree* (MIT), that provides the concept of relative distinguished names for managed objects as the instances of their *actual managed object class*. The combination of templates is supported in the form of strict inheritance allowing an allomorphic (ITU, 1992b) relationship between types of managed objects. This relationship can, for instance, be used to support backward compatibility or management of an object as if it were of another type.

MIDL supports the concept of multiple inheritance as it is known from the interface definition language used in CORBA. Since our language represents an extension of the CORBA concept of interface definition, only the additionally introduced components should be handled in a special manner when inheritance relationships are resolved. Thus, for attributes and management operations the resulting interface specification is constructed by unifying the parent and the child interface specifications adhering to the rules of strict inheritance. This means that there must not be any two attributes or operations with the same name but different types or signatures respectively in different parent interfaces.

States, however, are excluded from the inheritance scheme. This decision was taken because there is obviously no meaningful semantics for state inheritance. States represent symbolic names for certain periods of object activity and are strongly implementation dependent. There is no formal mechanism, however, to enforce naming consistency in operation constraints when interface types are refined by multiple inheritance. State combination by building the Cartesian product of all states in the parent interface types may look attractive from a formal point of view. Semantically, however, this is an improper approach since it leads to an explosion of the state space in which the majority of combinations will never occur and it makes it impossible to define constraints for critical operations that are only available at the refined interface type.

In MIDL the complete state set must be redefined when inheritance is applied to management interface definitions. For inherited critical operations that remain otherwise unchanged only the execution constraints must be re-specified because an operation which has been unavailable under certain conditions in the parent interface must not become non-critical in the interface subtype.

3.5 Example

Consider an EDI (Electronic Data Interchange) interface agent that acts as the EDI-enabling front-end, e.g. (Hutchison et al., 1995), of a business application. It should be noted that in this example the interface agent, not the application, is the object of management. We therefore describe a possible management interface for this interface agent.

This EDI agent (see figure 3) cyclicly performs the following activities on behalf of the business application:

1. wait for a new message in the input queue,
2. read the incoming message,
3. translate message from the EDI syntax to the application syntax,
4. deliver the translated message to the application,
5. wait for the result message (or acknowledgment) from the application,

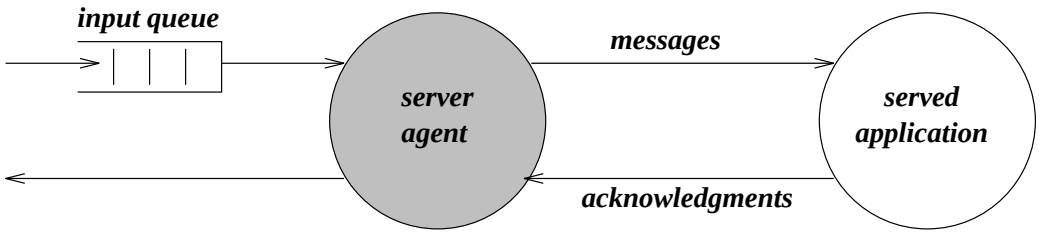

Figure 3 EDI Agent Connected with an Application

6. receive the result
7. translate the result back to the EDI syntax,
8. route the result to interested parties, and
9. remove the original message from the input queue (commit).

The EDI agent's actions lead to the following management interface specification:

```
interface EDIAgent {
  states = {
            Listening  (* Waiting for a new message *),
            TranslatingMessage,
            SendingToApplication,
            WaitingForResponse
                       (* Wait for application to respond *),
            TranslatingResponse,
            RoutingResponse
                       (* Send response to all interested
                          parties *),
            Committing (* Updating input queue *)
  }

  readonly attribute string  myName
                     (* Name of the object *);
  readonly attribute string  queueName
                     (* Name of the input queue *);
  readonly attribute long    queueLength
                     (* Number of messages currently in the
                        input queue *);
  readonly attribute string  originalMessage
                     (* Contains the last message received *);
  readonly attribute long    sequenceNumber
                     (* Current sequence number of message *);
           attribute boolean handleResponses
                     (* Flag to en-/disable handling of
                        responses *);

  ChangeQueueName(string queueName) while {Listening}
```

```
                (* Change the name of the input queue *);
  Kill()        (* Shut down the agent ungracefully *);
  Halt() while {Listening, TranslatingMessage}
                (* Suspend agents's operation, but not
                   while it is processing a response *);
  Continue()    (* Resume operation after a 'Halt',
                   otherwise no effect*);
}
```

The `EDIAgent` has six attributes that are visible to the management system. Only the queue name and the mode in which acknowledgments are handled can be modified. This is accomplished by the operation `ChangeQueueName` and the (read-write) attribute `handleResponses` which can be modified using a `set`-operation. The reason why an operation `ChangeQueueName` is desirable is that in case of a modified communication structure (re-configuration) the agent may receive its messages from another source using a different queue. The agent can only be re-configured safely if the current message has been processed. This is only guaranteed in the `Listening` state. All other attributes may only be read by the manager. Some operations cannot always be performed immediately. `Halt`, for example, may only be executed until the application has received a new message, hence ensuring that a consistent state is maintained between EDI agent and the application it serves.

The states defined in the above interface description are derived from the activities of the object. The association between characteristic object activities and the states is arbitrary. Therefore activities may be grouped together in states that have no atomic correspondence. For example, the state `Listening` is left when a message is found in the queue.

Aside from object state descriptions included in MIDL there are further differences between MIDL and CORBA IDL.

- We decided to incorporate optional descriptive units similar to PASCAL-comments. If specified they can be used to retrieve information about the attributes, states and operations which are offered at the object's management interface.
- All formal parameters of management operations other than *Get* or *Set* are considered to be `in`-parameters. This has two reasons. The first one is that no defined values could be assigned to `out`-parameters in case of asynchronous, deferred operation execution. Secondly, under the assumption that the attribute set is complete, i.e., that all variables whose values can be modified are included in the attribute set, `out`-parameters as well as return types are equivalent to *Get*-operations on the respective attributes. This implies that all actual parameters are passed *by-value*.
- In MIDL the return type of management operations is not specified. We decided for a generic return type of all management operations whose value is an enumeration representing the cases of success, pending execution, and invalid instrumentation of the requested operation. This approach makes exceptions other than standard CORBA-exceptions unnecessary.

4 MANAGEMENT ADAPTATION LIBRARY

The definition of an object's management interface alone is insufficient to make the object manageable. The object must also be instrumented to support the management functions mentioned above. A managed object will provide the management functions in two ways; first through annotations of the original code and second through the addition of a new thread for communication with the management system.

4.1 Instrumentation Requirements

The annotations of an object's original program code must meet the following requirements:

- When instantiated, the managed object must register itself with the management system and make its management interface accessible to the manager.
- The life cycle of the object must be subdivided into states as defined in the management interface specification.
- The management information base that stores the object's attributes must be represented in terms of data structures in the object code. An implicit retrieval function used by the manager to access an attribute value must be generated for each attribute.
- The actual management functions described in the management interface definition must be implemented.
- Program code must be provided to queue a pending critical operation and to remove it if it is aborted.

For these annotations we provide a *Management Adaptation Library* (MAL) that contains the necessary functions to meet the above requirements. The goal is to allow object instrumentation by annotating the object code with calls to the common MAL and to special functions that implement the management operations defined in the management interface description.

4.2 Implementation Aspects

From the management interface description, code can be generated that supports the object instrumentation. State definitions given in the management interface description are transformed into an enumeration type. The attribute vector is transformed into a corresponding data structure, which represents the MIB for the managed object. The association between object variables and attribute values stored in the MIB is established by name, i.e., each attribute is associated with an object variable of the same name. The same applies for the management operations for which prototypes are generated. For each management operation, a function must be provided to implement the particular operation. Since this requires knowledge of the object's implementation it cannot be generated automatically but must be provided by the implementor. Existing approaches like IBM's TMN WorkBench (IBM, 1995) introduce a *workspace* which contains all the user-defined code that cannot be generated automatically.

In order to associate the object with the management system an additional management thread is introduced. It preserves manageability in deadlock situations at the primary

object interface and provides for suspend/resume operations. In particular, the tasks of this thread are to:

- serve manager requests to retrieve and modify attribute values,
- invoke management operations if the current object state satisfies the specified constraints, and
- queue or abort an operation on behalf of the manager if it cannot be executed in the current state.

Aside from the implementation of management operations we envisage two special library functions, `initialize` and `state`.

The `initialize` function provided by the MAL must be the first operation to be executed when the managed object is instantiated. Its purpose is to start the management thread and to register the object in the management system.

State transitions in the object can be signaled using the `state` function, which updates all MIB entries, sets the MIB state indicator to the new state and triggers the execution of a possibly pending management operation that now meets its execution constraints.

In contrast to the `initialize` function, no generic function to shut down a managed object needs to be provided. Terminating an application component gracefully depends highly on its implementation. For this reason, a shutdown operation should be part of the component-specific management interface, if desired. On the other hand, since the management system must be able to detect when objects terminate, it is not necessary to unregister an object explicitly. This detection can be based either on operating system features (such as the `wait` system call in UNIX) or on recognition of an exception returned upon operation invocation of non-existing objects (invalid reference).

If we consider, for example, a C-language binding for the management interface description, an include file is produced when the description is compiled. It will contain the compilation results for attributes (MIB), state definitions (enumeration), and management operations (function prototypes).

5 CONCLUSION

In this paper we presented the design of a general method to integrate a distributed application into a management environment. The design consists of a language to define the management interface and an instrumentation method for the original code of the application components. It allows the construction of managed objects from arbitrary application components assuming one has access to the application code. Though it may seem to be a restriction, this is in fact not the case because the owner or vendor is the instance that should decide to what extent intrinsics or even implementation details should be revealed to a management system. These managed objects can be bound to the management system in a generic manner through the Management Adaptation Library. The objects can be equipped with a common management interface defined by the Management Interface Definition Language, an extension of CORBA IDL. The extended application components can be easily embedded into a management system for distributed applications and do not impose any restrictions on architecture and topology of this environment be it OSI,

SNMP, or CORBA-based. Its usage for agent-object communication in the OSI management model can be a step to overcome the lack of standards in this area.

The next step of our work will be to refine the presented design. We will focus on validating our approach against more complex distributed applications. We will also address problems of authentication, authorization, concurrency, synchronization, recovery, fault management and accounting. In addition to continuing work on constructing and integrating managed objects, we plan to examine management agents that can be provided with some form of intelligence to optimize event-forwarding tasks and the projection of their functionality onto different management architectures. Based on this, we would like to complete the scenario with a flexible manager which, in the final version, will also be able to coordinate its work with that of other managers.

6 REFERENCES

Case J., Fedor M., Schoffstall M. and Davin J. (1990) A Simple Network Management Protocol (SNMP). Request for Comments 1157, Network Working Group.

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

Hoffner Y. (1993) The Management of Monitoring in Object-based Distributed Systems. in *Integrated Network Management III*, (eds. Hegering H.-G. and Yemini Y.) North Holland.

Hutchison A., Kaiserswerth M., Moser M. and Schade A. (1995) A Standards-Based Communication Subsystem for Medical Applications, IMIA Yearbook of Medical Informatics '95.

IBM (1995) IBM TMN WorkBench for AIX: Managed Object/Agent Composer's User's and Programmer's Guide (SC31-8006).

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

ISO (1995) ISO/IEC DIS 10746-3 Reference Model for Open Distributed Processing – Part 3: Architecture. ISO/IEC JTC1/SC21/WG7.

International Telecommunication Union (1992) Data Communication Networks – Management Framework for Open Systems Interconnection (OSI) for CCITT Applications. CCITT Recommendation X.700, Geneva.

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

International Telecommunication Union (1992) Information Technology – Open Systems Interconnection – Structure of Management Information: Management Information Model. CCITT Recommendation X.720, Geneva.

International Telecommunication Union (1992) Information Technology – Open Systems Interconnection – Structure of Management Information: Guidelines for the Definition of Management Objects. CCITT Recommendation X.722, Geneva.

Mansouri-Samani M. and Sloman M. (1993) Monitoring Distributed Systems. *IEEE Network Magazine*, **7**(6), 20–9.

Object Management Group (1993) Object Request Broker Architecture. OMG TC Document 93.7.2, Draft 0.0.

Object Management Group (1993) The Common Object Request Broker: Architecture and Specification. OMG Document 93.12.43, Revision 1.2.

Open Software Foundation (1994) Using DCE and DME To Manage Software in DCE-Based Environments.

Park S.-S. and Shiratori N. (1994) Distributed Systems Management Based on OSI Environment: Problems, Solutions, and their Evaluation. in *Proceedings of the 13th IEEE Annual International Phoenix Conference*.

Rubin H. and Natarajan N. (1994) A Distributed Software Architecture for Telecommunication Networks. *IEEE Network Magazine*, **8**(1), 8–17.

Rose M. T. (1993) Challenges in Network Management. *IEEE Network Magazine*, **7**(6), 16–9.

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

7 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 applications and their management. In his spare time he is currently working towards a PhD degree. His e-mail address is `san@zurich.ibm.com`.

Peter Trommler is currently a diploma student at the University of Erlangen-Nürnberg, Germany. He is currently working on his diploma thesis on distributed applications management at the IBM Zurich Research Laboratory. His e-mail address is `prtromml@immd4.informatik.uni-erlangen. de`.

Matthias Kaiserswerth received his doctorate in engineering from the University of Erlangen-Nürnberg, Germany. At the IBM Zurich Research Laboratory he manages a group currently working on networked application integration and management. He is also a part-time instructor at the University of Erlangen-Nürnberg. His e-mail address is `kai@zurich.ibm.com`.