Middleware and Application Management Architecture

vorgelegt vom
Diplom-Informatiker
Sven van der Meer
aus Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Gorlatch
Berichter: Prof. Dr. Dr. h.c. Radu Popescu-Zeletin
Berichter: Prof. Dr. Kurt Geihs

Tag der wissenschaftlichen Aussprache: 25.09.2002

Berlin 2002
D 83

Middleware and Application Management Architecture

Sven van der Meer

Berlin 2002

Abstract

This thesis describes a new approach for the integrated management of distributed networks, services, and applications. The main objective of this approach is the realization of software, systems, and services that address composability, scalability, reliability, and robustness as well as autonomous self-adaptation. It focuses on middleware for management, control, and use of fully distributed resources. The term integration refers to the task of unifying existing instrumentations for middleware and management, not their replacement.

The rationale of this work stems from the fact, that the current situation of middleware and management systems can be described with the term *interworking*. The actually needed *integration* of management and middleware concepts is still an open issue. However, identified trends in communications and computing demand for integrated concepts rather than the definition of new interworking scenarios.

Distributed applications need to be prepared to be *used* and *operated* in a stable, secure, and efficient way. Middleware and service platforms are employed to solve this task. To guarantee this objective for a long-time operation, the systems needs to be *controlled*, *administered*, and *maintained* in its entirety, supporting the general aim of the system, and for each individual component, to ensure that each part of the system functions perfectly. Management systems are responsible for this objective. Usage and control result in mechanisms for mapping information across application, service, and network level. Control, administration, and maintenance reflect management tasks on each of those levels. Integration of middleware and management results in a system that provides distributed applications with all of the introduced functionality.

Following the hypothesis of this work, the target environments are evaluated to extract requirements for the integrated approach. Based on this evaluation, a general framework is developed that clearly identifies the certain levels of integration, their boundaries, and their individual objectives. Goal of the framework is to establish a software layer between the applications and distributed technologies that provides integrated management services without loosing the advantages of middleware.

The approach is discussed in three steps. First, a general framework is defined based on the hypothesis and an evaluation of the target environments. The second step is the derivation of a Middleware and Application Management Architecture (MAMA) from this general framework. The last step is the realization of this architecture and its exploitation.

The result is a software system that decouples distributed applications from concrete middleware and management technologies. The system's functionality is offered to applications via an Application Programming Interface (API), which provides access to basic middleware and management facilities. The API is supported by an Application Definition Language that combines interface definitions with semantic information in order to enable automated processes for control and maintenance. Furthermore, application services are included to realize standard features such as naming, service lookup and discovery, event notification, and visual administration.

The approach described in this thesis recognizes international standards and developments. In fact, the approach depends on commonly used and well-agreed technologies from the areas of telecommunications, computing, syntax notations, distributed systems, systems management, and user interfaces.

Zusammenfassung

Die Dissertation zum Thema: Middleware and Application Management Architecture beschreibt einen neuen Zugang für ein integriertes Managementsystem für verteilte Netzwerke, Dienst und Anwendungen. Das Hauptziel dieses Ansatzes ist ein Softwaresystem mit Diensten, welche Komponierbarkeit, Skalierbarkeit, Zuverlässigkeit und Robustheit genau so wie autonome Selbstanpassung ermöglichen. Es konzentriert sich auf Middleware für Management und Kontrolle sowie auf die Nutzung für verteilte Komponenten. Der Begriff Integration beinhaltet in dieser Arbeit die Vereinigung existierender Instrumente von Middleware und Management, aber keineswegs ihre Ersetzung.

Die Ausgangslage dieser Arbeit ist die Tatsache, dass die aktuelle Situation des Verhältnisses von Middleware und Management lediglich als *Zusammenarbeit* bezeichnet werden kann. Bekannte qualitative und quantitative Tendenzen in der Kommunikation und Datenverarbeitung verlangen nach integrierten Konzepten anstelle immer neuer Szenarien der Zusammenarbeit in spezifischen Konstellationen, die wie Inseln vielfältige spezifische Brücken zueinander benötigen. Die sich daraus ergebende Notwendigkeit einer *Integration* von Middleware- und Management-Konzeptionen ist nach wie vor ein ungelöstes Problem.

Auf diesem Hintergrund wurde die vorliegende Arbeit konzipiert. Es wird ein Modell für die Integration von Middleware und Management begründet, entwickelt und seine Funktionsfähigkeit nachgewiesen. Es stellt ein System dar, welches verteilte Anwendungen mit allen wesentlichen Funktionen ausstattet: Verteilte Anwendungen müssen so beschaffen sein, dass sie stabil, sicher und effizient genutzt und betrieben werden können. Middleware und Dienstleistungsplattformen werden eingesetzt, um diese Aufgaben zu erfüllen. Um diesen Zweck über große Zeiträume hinweg zu garantieren, muss das integrierte System in seiner Gesamtheit kontrolliert, verwaltet und gewartet werden. Damit werden das grundsätzliche Ziel des Systems und die Aufgabe jeder einzelnen Komponente unterstützt und gesichert, dass jeder Bestandteil des Systems selbständig und im Zusammenwirken ordnungsgemäß funktioniert. Managementsysteme sind für diese Zielstellung verantwortlich. Nutzung und Kontrolle verlangen einen Mechanismus zur Informationsumwandlung zwischen den Ebenen Anwendung, Dienste und Netzwerke. Zugleich sind Kontrolle, Verwaltung und Wartung auf jeder dieser Ebenen erforderlich.

Der Zielstellung dieser Arbeit folgend, ein funktionsfähiges Systems der Integration zu erarbeiten welches den gestellten Anforderungen gerecht wird, werden die Zielumgebungen analysiert, um deren Anforderungen an einen integrierten Ansatz zu bestimmen. Basierend auf dieser Analyse wird dann ein generelles Rahmenwerk entworfen, das exakt die einzelnen Ebenen für eine Integration definiert, ebenso die Teilziele der einzelnen Ebenen wie auch die Grenzen zwischen ihnen. Hauptziel des Rahmenwerkes ist die Definition einer Schnittstelle zwischen Anwendungen und verteilten Technologien, die integrierte Verwaltungsdienste realisiert ohne die Vorteile von Middleware zu verlieren.

Dieser Ansatz wird in drei Schritten dargelegt. Als erstes wird das generelle Rahmenwerk definiert, welches auf der Grundthese und der Analyse der Einsatzgebiete beruht. Von diesem Rahmenwerk wird eine Architektur abgeleitet. Der letzte Schritt besteht in der Realisierung, Implementierung und Verwertung dieser Architektur in geeigneten Szenarien.

Als Ergebnis der Arbeit entsteht ein Softwaresystem, das verteilte Anwendungen von konkreten Middleware- und Managementtechnologien abkoppelt. Dieses Systems und seine Funktionen wird den Anwendungen über eine entsprechende Programmierschnittstelle zur Verfügung gestellt. Diese bietet Zugang zu fundamentalen Middleware und Managementdiensten. Das wird von einer formalen Sprache unterstützt, die neben der herkömmlichen Definition von Objektschnittstellen auch semantische Bezüge der Anwendungen spezifizieren kann, um automatische Kontroll- und Wartungsprozesse zu ermöglichen. Weiterhin beinhaltet das System Basisdienste für die Anwendungen, wie zum Beispiel die Verwaltung von Namen, das automatische Suchen von Diensten, Nachrichtenverteilung, und grafische Verwaltung.

Der Ansatz, der in der vorliegenden Arbeit beschrieben wird, verwendet internationale Standards. Er basiert auf weit verbreiteten und allgemein anerkannten Technologien aus den Bereichen Telekommunikation, Computer, formale Sprachen, verteilte Systeme, Systemverwaltung und Benutzerschnittstellen.

Acknowledgements

I would rather be attacked than unnoticed. For the worst thing you can do to an author is to be silent as to his works.

Samuel Johnson

There have been three men that, to follow the citation, 'attacked' constantly. The first of them has been my mentor, Prof. Radu Popescu-Zeletin. I extend my gratitude to him for giving me the opportunity to work in the unequaled combination of the department for Open Communication System (at Technical University Berlin) and the Competence Center for Open Communication Systems (at Fraunhofer FOKUS). He inspired this work with his vision of I-centric Communication and he gave me both, time & freedom to work on my own behalf and stimulus & pressure to go where I have never been before. Prof. Geihs was a perfect adviser. He encouraged me with critical comments and the opportunity of a discussion with his research associates to improve the quality of this thesis. Last not least, Prof. Horst van der Meer, my father, who was there as a counselor every time I needed him. Thank you for the long discussions at our summerhouse, the endless hours on the telephone, and your long comments by email. There where several times when you showed me the right way!

Beside those three great men there is a huge number of friends, fellows, colleagues, students, and teachers that supported me since my first day at the Technical University Berlin up to my work as research assistant at OKS and FOKUS: the people who have joined my during my study, the people of ICE, the PCSS and the iPCSS and the UMS teams, and all the colleagues at Fraunhofer FOKUS. Thank you all for your help and patience.

This work has been nearly equally done at my offices at TUB & FOKUS and at several places where good friends and my lovely family reside. In words I cannot tell how I appreciated your help, encouragement, and support: Mimi & Horst, Kerstin & Thomas & Nico & Jakob, Ronny & Suse, Marianne, Trine, Kim, Sara, ...

Success is the sum of small efforts, repeated day in and day out.

Robert Collier

Finally, I'd like to thank everyone who spent any kind of effort, small or huge, once or often, knowingly or instinctively, for your time, guidance, and patience!

Berlin, May 2002

Sven van der Meer

Table of Contents

Chapter 1
Introduction

1.1. Motivation

The rational of this thesis stems from the need of **effective and secure usage, operation, control, administration, and maintenance** of billions of devices, millions of services, and tens of thousands of applications. This document describes a new approach for the integration of management and middleware. This necessity is underlined in a dedicated issue from the specific program of the European Framework Program 2002-2006 of the European Community that focuses on "new approaches for software, systems, and services that address composability, scalability, reliability, and robustness as well as autonomous self-adaptation. It addresses middleware for management, control, and use of fully distributed resources" [EU-FP6Draft]. The approach of this thesis is presented in form of a new general framework and the derived Middleware and Application Management Architecture (MAMA).

1.1.1. Trends in Communication and Computing

The technical basis of communication is shifting from typical insular solutions towards interworking environments. New services are going to influence many parts of our daily life (travel, education, entertainment, shopping, recreation, and medical services) and all places people live and work at (home, office, school and university, car, airplane, elevator, marketplaces, restaurants, clinics, etc.). The communication systems at such places can exchange many types of information, such as user profiles, accounting information, and personal data of users.

Applications and systems available today focus on issues like mobility in fixed and wireless networks, unification of services access, personalization of service delivery (at least for a certain set of services), and interoperability of services. With interworking environments, the degree of distribution of applications is growing. Four trends can be identified that influence distributed applications:

1. **Telecommunications** – The era of monolithic telecommunication networks with centralized intelligence is ending. Computing and telecommunication technologies converge. The integration of services based on the Internet Protocol (IP) and the advantages of packet-switched networks are going to change the characteristics of telecommunication networks. Looking at 3rd generation mobile networks (UMTS), new approaches are already in the phase of standardization. The interworking of formerly separated signaling protocols and the adaptation of information streams by Media Gateways indicates this development. [Magedanz01] [vdMeer00b]
2. **Network Computing** – The Internet has altered from a scientific network for technical specialists to a widely used market and information place. IP is going to be extended to overcome its original restrictions regarding, e.g., real-time transmission and quality of service demands. The Internet is now a business driver affecting all parts of global economy. The interconnection of many IP-based networks changes the characteristic of workflow process from isolated single host computing towards real network computing. IP is going to become the protocol that unifies network access. [Cerf00]
3. **Devices and Wearables** – The miniaturization of microelectronics has enabled small and powerful handheld devices and wearables, small devices with minimum power consumption, which people can easily carry. The idea behind wearables is to design gadgets out of simple electronic components in place of 'computers'. They need not to be connected to a network, at least not in a permanent way [Pentland99]. In addition to 'classical' Customer Premises Equipment (CPE, such as telephone, fax, Personal Computer – PC), microelectronic controlled equipment (home theatre, white goods, light control, burglar alarm, location systems, etc.) are applied in home and office environments. Most of these systems are currently stand-alone [Pfeifer99]. Integrating these devices in a communication environment allows new services and demands effective strategies for management.

4. **Context-aware Services** – The evaluation of service personalization approaches has shown that they are not sufficient to match user requirements. Service personalization considers factors like time (periods of time), costs, media conversion, and intelligibility to deliver information. Context-aware applications combine this information with measured data from the current environment of the user (like temperature, noise level) and evaluate them in order to adapt the environmental conditions with regard to the preferences of this very user. [Abowd99]

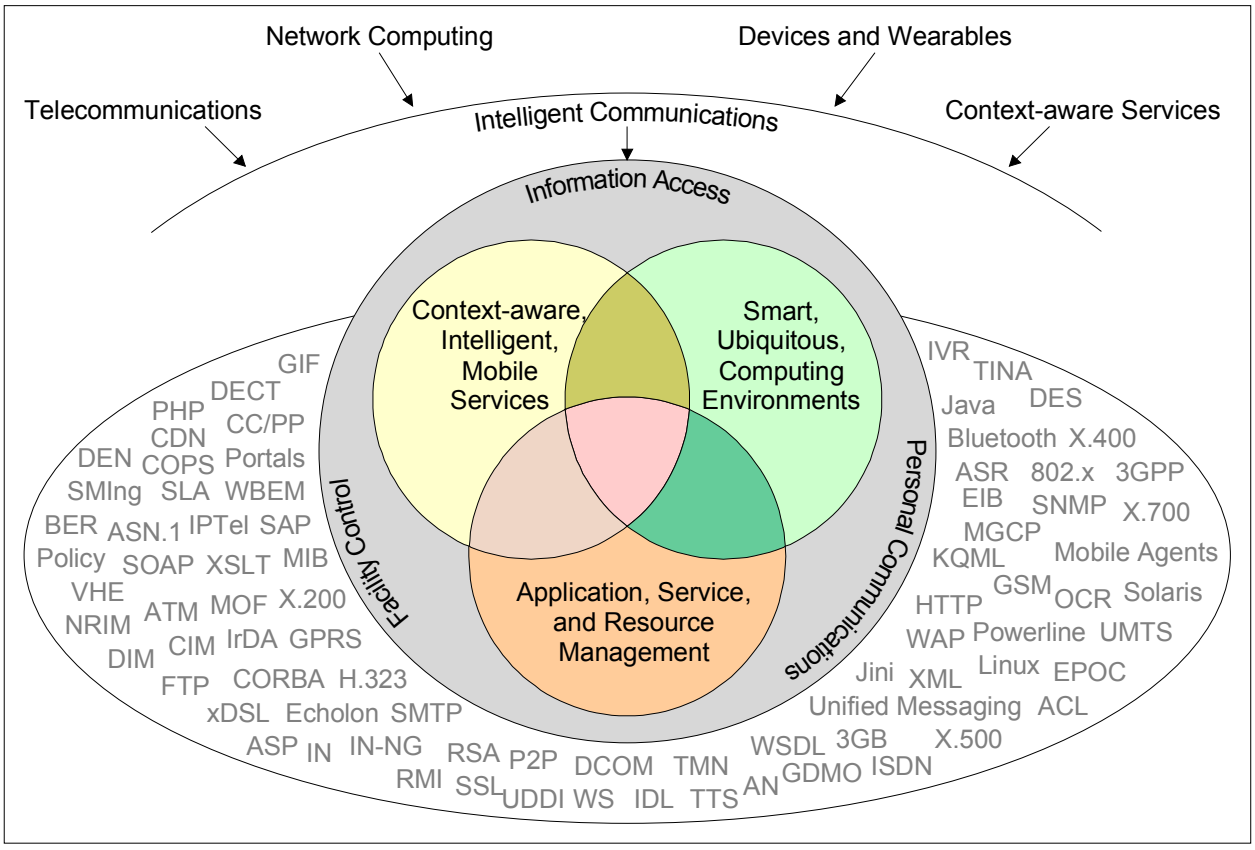

Figure 1-1: Trends in Communications [vdMeer01a]

These four trends influence the nature of (intelligent) communications. Figure 1-1 shows the resulting framework. Intelligent communications can be divided into three areas of concern: *information access*, *personal communications*, and *facility control*. These areas employ a huge number of technologies. Based on these areas, three important issues can be defined. The first issue is the realization of services, which deliver a nomadic user with context-aware information. The second issue is the definition of smart, ubiquitous computing environments, which are built out of heterogeneous resources. The third issue is to provide answers to new requirements for the management of applications, services, and resources.

The questions come from customers and users (regarding applications), service providers (concerning services), and network operators (dealing with resources). The offered applications and services become increasingly complex. The network is constructed out of ever more heterogeneous technologies. Operators and providers compete in a customer-driven market. This leads to a high degree of autonomy and, at the same time, to an intrinsic need for cooperation between operators and providers.

1.1.2. Integration of Management and Middleware

Do current management frameworks offer answers to the new questions? Are they applicable for future applications, services, and resources? Answers to those questions can be given after analyzing current trends in communication and computing and after evaluating the actual level of interworking and integration of management and middleware. Doing this, the answer is *no*, under most favorable conditions *maybe*.

The current situation can be described by the interworking (*not* the integration) of middleware and management systems. Furthermore, the available information sources within companies and organizations rely on many different solutions for storing data. They are still waiting to be harmonized. In short, this situation can be described as follows.

1. **Distributed applications** can **access classic management systems** via 'gateways' that translate specifications and interactions. Figure 1-2 shows an example where clients from the WWW[1] manage resources via CORBA[2]. The figure shows the CORBA gateway to the management systems. [CORBA-MAN]
2. **Legacy management systems** are used to **manage distributed applications**. Providers and operators use their running management systems to administrate distributed applications. Thus, the investment for the education of this staff is not wasted but reused for distributed applications. [CORBA-TMN]
3. **A few middleware concepts and many different products are used in parallel**. Most of them provide mechanisms for interworking (that means for the invocation of operations and the exchange of information). The information exchange often needs an informal agreement of developers on semantic. Each middleware concept includes basic management functionality on object level (e.g. based on the clustering concepts of ODP[3]; [Raymond95]). Furthermore, "middleware developers strive to support applications that meet the technical challenges of ubiquitous computing" [Geihs01].
4. **Service platforms form a layer of abstraction** enabling service creation and deployment, monitoring of distributed applications, and integration of legacy systems. Within those platforms, the management of services is included by means of the support of tools for the definition of services and business roles, an execution environment for services, and the management of the platform itself. [Funabashi00] [Magedanz01]

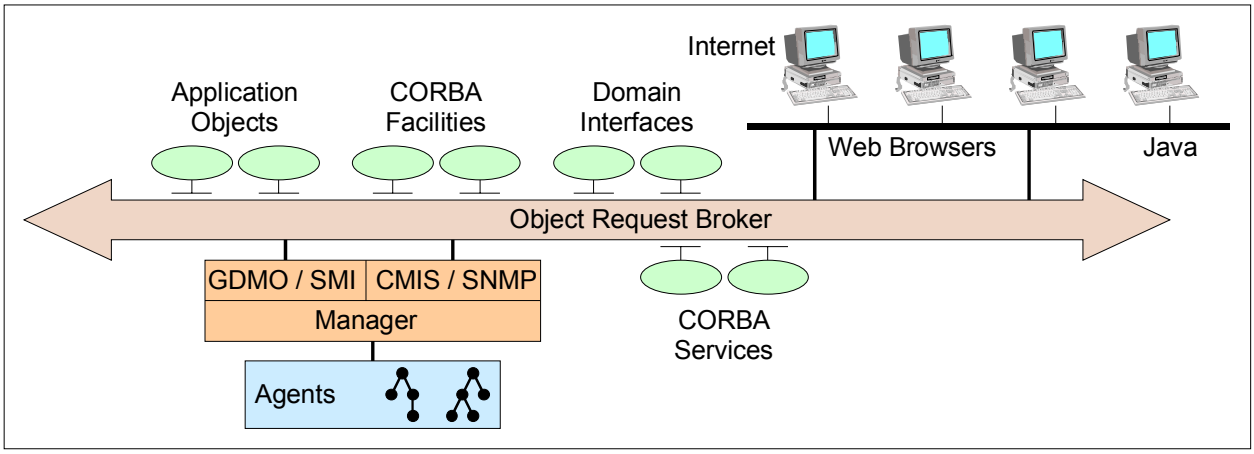

Figure 1-2: Distributed Applications accessing classic Management Systems [NMF-GB909]

Management is defined as all activities needed to operate communication networks and services in a secure and effective way. It identifies methods and provides tools to support configuration, monitoring, maintenance, and administration of these networks and their services. The target vision is to support user and provider in planning and operating networks and distributed systems. Nevertheless, middleware and service platforms do not reflect management standards sufficiently. Each of them comes with a new concept for managing objects or services. Furthermore, development of middleware does not follow management principles. The components object specifications, protocols, and data formats are designed specifically to support distribution. Here, an integration of management concepts into these components can be an important step. [Hegering99]

[1] World Wide Web

[2] Common Object Request Broker Architecture

[3] Open Distributed Processing

Distributed applications define new requirements, which middleware and management need to address. Applications and devices are not realized with a single technology. Devices and network appliances need to be accessed in a technology independent manner. Specifications must be independent of dedicated vendors. At the same time, the support of autonomy of operators and vendors is needed to enable them to offer clearly distinguishable products. These are basics for cooperative environments in a competitive world.

1.2. Objectives and Scope

The aim of this thesis is to describe a unified framework integrating concepts from middleware and management. This framework decouples the applications from the middleware/management architectures. The applications should be intrinsically manageable by means of configuration and fault management.

The three stages to reach this goal are shown in the middle of Figure 1-3. They depend on middleware technology (right side of the figure) and management technology (left side of the figure). The first stage is to identify the basic characteristics of middleware and management regarding their support of distributed applications.

The second stage is to define a general framework. This framework should describe individual layers that need to be distinguished in order to decouple applications from technology. These layers can be used to describe areas of concern and the functionality that should be integrated. The framework itself should focus on applications, services, and resources. It needs to be described in a way that is not prophetic (that is to say predictive by nature) and not too dogmatic (that is to say too authoritative and rigid in style). Furthermore, the framework can be used to identify components of a specific architecture. These components have to reflect actual developments and technologies for the execution of services as well as for the management of related resources.

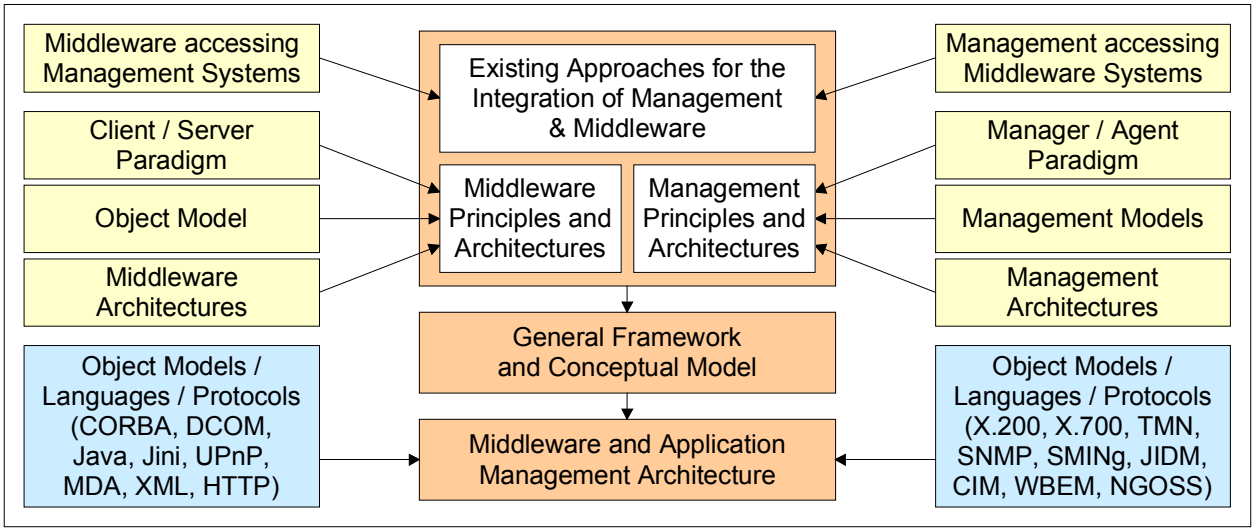

Figure 1-3: Development Process

The third stage represents the final result in form of the Middleware and Application Management Architecture that offers uniform management functionality for applications, services, resources, and components of the framework itself.

The concepts developed within this work combine the four characteristics of the current situation of middleware and management as described in section 1.1.2. This combination results not in a service platform in the traditional understanding (like the layered architecture of RM-OSI[4], the definition of planes of Intelligent Networks – IN, the separation of management and middleware of CORBA-TMN interworking

[4] Reference Model for Open Systems Interconnection

([CORBA-TMN]), or the programming language dependant integration of Jini and the JAVA Management Extension – JMX). Instead, the given approach focuses on the integration characteristics but the independence from technologies from the areas of middleware and management. This is achieved by a systematic analysis of the state-of-the-art middleware architectures and the long-time used management frameworks. Result of this analysis is a technological independent specification of components, protocols, and data formats incorporating features from both worlds.

1.3. Organization of this Thesis

The organization of this document follows the development process of the actual work. After this introduction, **Chapter 2** describes the hypothesis of this thesis, which is based on the identification of the major tasks of a distributed system. The requirements of distributed systems are analyzed according to environments this work is targeting. The hypothesis and the requirement analysis enable the definition of the general framework.

Chapter 3 is dedicated to the Middleware and Application Management Architecture. This architecture is derived from the general framework of chapter two. MAMA is defined by five individual solutions. This starts with the *Application Definition Language (ADL)* that represents a combination of management languages and middleware interface definition languages. Based on ADL, a *Core Model* is specified. The third solution is the MAMA *Application Protocol* that defines communication principles. The MAMA *Application Programming Interface (API)* decouples applications from middleware technologies. The last solution is a set of *Application Services*. The chapter is concluded by a discussion about the development of MAMA applications.

Chapter 4 features the realization of the architecture. It focuses on the implementations for the solutions introduced in chapter three. The realizations comprise a compiler for ADL, the c++ implementation of the MAMA API, and the application services for naming, directory, and visualization.

Chapter 5 summarizes this thesis. The first part of this chapter is dedicated to concluding remarks on the scientific results of this work. These conclusions indicate the basic assumption of this thesis along with the important results. The second part of this chapter deals with related applications and with other projects that investigate in similar issues.

After the **List of Figures**, the **List of Tables**, the **References**, and the **Acronyms**, three Appendixes provide additional information on several topics. **Appendix A** summarizes typographical conventions and graphical notations. Furthermore, it explains the symbols of languages that are used to specify grammars. **Appendix B** is dedicated to ADL. It presents the lexical conventions and the complete grammar of ADL. **Appendix C** provides the complete specifications of MAMA that are described in chapter three and four.

Chapter 2
Hypothesis and General Framework

This chapter is dedicated to the general framework for the integration of management and middleware. Section 2.1 starts with the definition of five terms: use, operation, control, administration, and maintenance. The combination of those terms provides the basis for the framework itself. They specify areas of concern. This first section further deals with the two major activities of distributed systems – information mapping and system management – that are used to set the focal point of the framework. Result of this excurse is the definition of the hypothesis. Section 2.2 analyses target environments. Section 2.3 describes the general framework. The objectives and the principles of the framework from chapter one are refined with the results of section one and two of this chapter.

2.1. Defining Use, Operation, Control, Administration, and Maintenance

Each distributed system is designed to accomplish a purpose. The system and its purpose can be viewed from different perspectives, whereas each perspective creates its own requirements. However, all perspectives belong to the same basic understanding: A distributed system needs to be prepared to be *used* and *operated* in a stable, secure, and efficient way. To guarantee this objective for a long time operation, the system needs to be *controlled*, *administered*, and *maintained* in its entirety, supporting the general aim of the system, and for each single component, to ensure that each ring of the chain functions perfect.

The terms *use* and *operation* describe the part of a system that is seen by users and customers. They are concerned about the system's ability to serve them. A company running a system relies on its efficient *operation* in order to generate revenue. This operation is supported by *controlling* the system. The term *control* describes the brief but permanently reoccurring task of keeping the system stable to serve its customers and to generate revenue. This includes e.g. the configuration of system components and the record of data for accounting.

Administration and *maintenance* reflect long term operation and control of a system. This general task is divided into several individual procedures. Administration starts with the permanent monitoring of the system and the logging of all occurring events to analyze the behavior of its components. The second aim of monitoring is the detection of system failures.

Two different types of failures can be identified. Technological failures are a result of hardware problems in computers and components and communication errors (e.g. broken link or overload). Management systems are already able to detect those failures and to follow (predefined) procedures to reestablish the communication infrastructure. The other type of failures relates to content and semantic issues. The forecast of those failures is still a difficult task. These failures occur when objects are initialized with wrong data, when data is changed during transmission, as a result of mistakes from programmers, or problems in the design of an application. They are especially difficult to handle when the distributed system is of intrinsically complex nature.

In Figure 2-1 the three areas of concern are separated. Applications provide the interface to users and customers enabling them to utilize services. Services are software assemblies of components that offer functionality for applications and provide the access to resources. Resources are software and hardware components needed for the provision of services and for their execution. In other words, a distributed system employs resources to complete services and applications.

The figure furthermore shows the two main activities inside such a system. Information is mapped across levels, upwards and/or downwards, for usage, operation, and control. The system is managed at each level through control, administration, and maintenance. The assignment of individual tasks to one activity de-

pends on the system's purpose. This is also true for the separation of the activities. The mapping of information is supported by management activities and the system's management relies on information mapping.

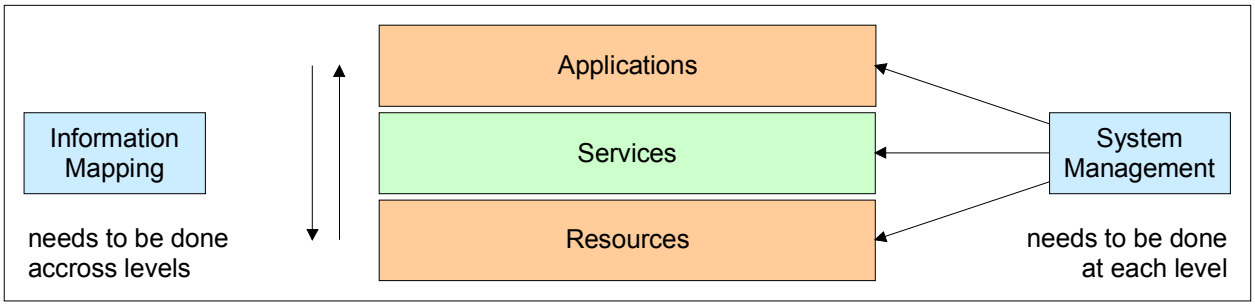

Figure 2-1: Areas of Concern and Activities of Distributed Systems

Both activities have commonalities. The mapping of information as well as the management of the system can be described in terms of presentation, specification, submission, re-specification, triggering, queuing, access, and execution. This layered approach is used to model modular client/server applications. It is built upon small and functionally specialized components that can be reused across multiple systems. Each layer of this model provides a specific function in the overall scope of the system.

The first layer focuses on the *presentation* of information along with the verification of results to support the specification and the submission of individual tasks. The *specification* answers six questions about a task: Who? (identifier) wants What? (request) Where? (destination) When? (schedule) Why? (purpose) and How? (execution plan). The answers to Who?, What?, When?, and Why? are the basis for a *submission* that is a complete job specification. The *re-specification* is responsible for the mapping of What? towards a set of commands that is needed to be executed to fulfill the purpose. *Triggering* activates and deactivates jobs based on date and time information, completion of other jobs, or other available data. *Queuing* provides load balancing and the prioritizing of jobs. *Access* functions as a mediator between the above layers and the execution layer. It provides interfaces to resources. *Execution* executes any job that is submitted from submission via access. The type of service executed here depends on the overall purpose of the system and might range from database access, media conversion, user interaction, up to device and network usage. All described layers are supported by navigation, security, metering, and logging.

These commonalities between information mapping and system management should lead to a similar design of software that handles them. However, the reality is different. Management activities are outsourced to specialized systems that act independent of the systems they manage. The reason for that is the evolution of network systems. Starting with basic services (telephony and data transmission), the first two types of networks that existed had had no need for automated management. The networks had been operated manually. Changes in society and on the market have led to the extinction of this business model. Networks have become connected, the market has demanded more than the basic services, and the number of devices has increased. This resulted in the actual need for an automated management of services and resources to continue to run the network in an efficient (and profitable) way. The operation of services and their management became separated areas with different approaches. Today, distributed systems and services run on top of middleware and are managed by dedicated management systems.

New developments can be identified. Formerly separated networks converge. Services for telecommunication, entertainment, information, and education move to the same infrastructure. The market changes from a provider dominated market towards a customer driven market. These developments demand for a rapid service creation, testing, deployment, operation, and withdrawal. The lifecycle of applications, services, and resources gets short, so that a dedicated management system cannot be provided just in time. Generically, two ways of evolution for management are reasonable. Management systems can be engineered basically to be applicable to any type of distributed system or management is incorporated into the very systems themselves. [Booz96]

The first approach is followed by several international standardization bodies and industry forums. With the Telecom Operations Map (TOM) and the Next Generation Operations System Support (NGOSS), the

TeleManagement Forum provides models for the realization of management systems based on the formal description of business models and the identification of generic management tasks, which can be applied to concrete distributed systems. With the Model-Driven Architecture (MDA), the Object Management Group (OMG) has recently published a method for a tool-supported design of distributed applications. The MDA is based on the Uniform Modeling Language (UML) and the eXtensible Markup Language (XML). The Distributed Management Task Force (DMTF) comes with an integrated approach of formerly separated data sources. Directory Enabled Networks (DEN) offer the unification of information stored in databases with a Common Information Model (CIM). This model can be mapped to widely used technologies like the Lightweight Directory Access Protocol (LDAP) and XML. The DMTF addresses also the Internet and the business needs with its Web-based Enterprise Management (WBEM) solution.

In the second approach, the basic characteristics of middleware and management are harmonized, in other words, integrated. This integration does not focus on the definition of management functionalities for middleware platforms, but on the integration of basic characteristics of definition languages, meta information, repositories, protocols and protocol elements, programming interfaces, and application services. It can be described in the form of a general framework.

This work follows the second approach. The integration is done in multiple steps. First, the target environments are analyzed. The environments fulfill business needs that are specified with business models and supported by service platforms. The environments reflect certain applications, services, and resources that need to be controlled, administrated, and maintained. Furthermore, the environments are affected by emerging technologies that promise to solve business needs and customer requirements more efficient or that enable services not possible to be realized yet.

The framework includes the identification of targeted solutions, basic principles, and general components. Furthermore it describes a mechanism that enables the integration of the two introduced activities of distributed systems: the mapping of information across layers and the management of layer entities.

2.2. Target Environments

The general framework focuses on specific target environments. In general, these environments are telecommunication systems. In more detail, these environments are based on object-oriented technologies and distributed systems. Business models and service platforms formulate the requirements of service providers and network operators. The service platforms are changing their nature from centralized towards decentralized and flexible structures. Some emerging approaches are important for the future applicability of the general framework.

2.2.1. Business Models

One of the strong driving forces in telecommunications is the need to create a collaboration consensus between the different players in shared business opportunities. The formal expression of that conception can be found in a business model, identifying the main players (stakeholders), their business roles, and interactions between them. It is based on the projection that the telecommunication systems evolve into an open, deregulated, multi-provider market and information place.

The Intelligent Network (IN) has defined a standard business model for telecommunications [ITU-Q1201]. The Telecommunication Information Networking Architecture (TINA) has combined new issues of the telecommunication market to develop a flexible model focusing on the telecommunication market [TINA-BM]. This model has recently been adopted by the Study Group 11 of the International Telecommunication Unit (ITU) as a reference model for the open telecommunication market.

The modeling of business processes is changing in many areas. Concepts like Collaborative Business (cBusiness) are going to overcome the legacy business process modeling that is focused on information of a company. The approach of cBusiness demands a negotiation resulting in the definition of a shared goal of collaboration between two or more companies. The business processes are no longer modeled using a linear timeline but based on clearly identified business roles. [Röhricht01] [Scheer02]

Beside these generic business models, other organizations have realized environment specific models for the description of roles and interactions. One example is the Printer Management Information Base (MIB) [IETF-RFC1759]. Here, the identified roles are closely related to a printing device. The interactions describe use cases for a printing device.

2.2.2. Service Platforms

Service platforms are an instrument to design, test, deploy, operate, and terminate services regarding business needs and technical constraints. The term "service" is used with a very diverse meaning that depends on the segmentation of areas of concern. A general model for this segmentation is to distinguish the problem areas of *transmission* (transmitting bits), *connectivity* (end-to-end stream binding), *communication* (sessions for users and applications), and *information* (access to distributed entities). Each area comes with a dedicated definition for the term service.

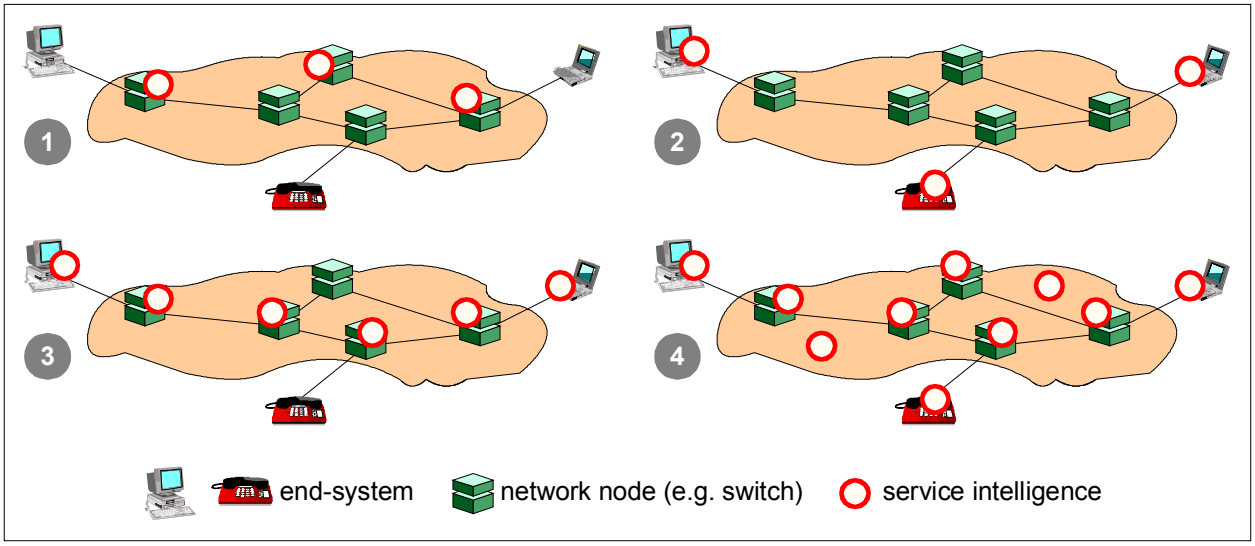

Figure 2-2: Service Platforms – Distribution of Intelligence [Campolargo99]

The major paradigm of a service platform depends on the actual place where the intelligence for information services resides. Two different approaches are used today (cf. Figure 2-2 case 1 and 2): The intelligence is located centralized within the network (telecommunication, [Magedanz96]) or the intelligence lies at the end systems (the Internet, [IETF-RFC1958]). Both approaches offer a limited flexibility since they cannot support all kind of services with a similar quality of service. For example, an emergency call demands a connection to a responsible operator within several seconds. This cannot be provided when the intelligence is located at the end-systems, since the emergency call must be transmitted in a very limited time window and highly prioritized inside the network [Draft-Hohno]. Solutions for the special requirements of telephony services within the Internet are currently under investigation [Draft-IEPREP]. Furthermore, the fast creation of new services from independent service providers requires an enormous degree of flexibility. This is usually not offered when the intelligence is located inside a huge telecommunication network only [DRAFT-OPES].

The convergence of telecommunications and the Internet has resulted in developments that focus on those issues. Within the last years, Distributed Processing Environments (DPE, [TINA-EMC]) and Open Service Access (OSA, [3GPP-OSAReq]) appeared aiming to distribute intelligence within the network and at the end systems at the same time (case 3 of Figure 2-2).

The trends described in Chapter 1 enable services that are created dynamically and can be activated anywhere [Magedanz01]. This requires service platforms that are capable of placing intelligence wherever it is needed, at the time it is needed there, and in a form that serves services best: centralized, distributed, at end systems, at switches, or at special nodes (case 4 of Figure 2-2, [Campolargo99]).

[Tönnby00] explains the shift from traditional solutions towards new approaches. Services for *information* are no longer built on vertically integrated services and networks but as horizontally structured applications built on top of services from any kind of network. Portals for information services get important for unifying information access [Scheer02]. *Connectivity* services integrate circuit switching and packet switching, with a significant trend towards packet switching. *Connectivity* is supposed to be ubiquitous and, when realized with computers, based on the Internet Protocol (IP). *Transmission* moves from narrow-band networks to broadband networks. *Communication* services recognize that communication is not only done between users, but also between users and 'things', and between 'things'.

2.2.3. Applications, Services, and Resources

The appearance of the Internet has been followed by two other categories of IP-based networks, Intranets and Infranets. Intranets are similar to the Internet, but they cover distinct organizational domains and restricted access policies. Infranets are sub-computer networks usually employed for infrastructure and equipment control. Currently, Infranets are realized with a variety of technologies. However, huge effort is spent to adapt or to migrate existing technology towards IP. There exist already more than 12 billion sub-computer devices, equipped with micro controllers [Luckenbach99]. Those devices produce an enormous amount of information that is processed within specialized application environments only, e.g. to control the elevators of an office building [Pfeifer99].

Linking the microelectronic controlled devices to the Internet imposes completely new services and scenarios that affect all areas of living and working. A (limited) collection of services and resources is presented in Table 2-1. Services and resources are categorized in four key areas: *communication*, *localization*, *information*, and *appliances*.

	Communication	Localization	Information	Appliances
Services	email, fax, voice, IP telephony, short message services, paging, instant messaging	hardware tracking and profiling, person tracking and profiling, theft protection	e-learning, kiosk information, media conversion, content networking, e-commerce	surveillance, localization of information, access control, automatic maintenance
Resources	telephone, mobile, fax, PDA[1], computer, pager, web phone, notebook	active badges, global positioning, cell location, wireless networks	object-oriented & relational databases, WWW[2], file systems, net news	smart IP devices, Powerline, Jini, embedded systems, network appliances

Table 2-1: Applications, Services, and Resources [vdMeer01a]

On the one hand, the four different areas include a huge number of services. On the other hand, the service vendors are no longer in the position to dictate services for the mass market. Instead, customers and end-users demand personalized services.

The services and resources need neither to be present permanently nor pre-configured for single use-cases. They can be available in a spontaneous way, technically backed by ad-hoc networks where devices are registered and accessible for the duration they appear within a dedicated environment [IBM99]. Services need to be designed and operated on heterogeneous networks with several different types and classes of devices.

[1] Personal Digital Assistant

[2] World Wide Web

2.2.4. Emerging Approaches for Distributed Systems

The 'traditional' approaches for distributed systems are based on middleware like the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM), and Enterprise Java Beans (EJB). Service platforms and architectures as TINA and OSA profit from the distribution transparency and a unified view to distributed objects. However, the underlying idea is quite similar – if not identical – to Remote Procedure Calls (RPC). This results in client/server applications with more or less intelligent clients.

New approaches for distributed systems follow other paradigms. Since those new developments have a good chance to influence future service environments, they are an essential part of this thesis. It must be possible to incorporate them into the framework developed within this thesis in order to provide a solution that is prepared for the future.

2.2.4.1. Peer-to-Peer Networks

The basic definition of Peer-to-Peer (P2P) is presented by [Shirky00] as "a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the Domain Name System (DNS) and have significant or total autonomy from central servers." This means, a system is a P2P system when it supports the temporary establishment of structures to fulfill a special task not demanding fixed configurations of hosts and devices.

A P2P system gives its nodes significant autonomy. Following this, [Schmid01] gives a more simple definition that a system is peer-to-peer when all components of the system are equal.

[Shirky00] identifies the distinction of ownership as a characteristic for P2P systems. The question here is who owns the hardware that the service runs on? The given example compares Yahoo[3] with Napster[4], where Yahoo runs on Yahoo's own hardware in Santa Clara while Napster runs on hardware owned by individual Napster users. P2P is able to decentralize not only features but also costs and administration. The concept of P2P has been successfully employed in other commercial products [Müller02].

2.2.4.2. Agents, Mobile Agents, and Mobile Code

Agent technology is not totally new. However, this level of usage of agent-based software has dramatically increased in the last year. Agents are software components that fulfill two orthogonal concepts. The first concept is the agent's ability for *autonomous* execution. The second concept is the agent's ability to perform *domain oriented reasoning*. In general, agents are software-based computer systems that comprise the following properties:

- Autonomy – agents operate without the direct external interventions and have control over their actions and their internal state.
- Sociability – agents interact with other agents via agent-to-agent communication languages.
- Reactivity – agents perceive their environment and response in a timely fashion to changes that occur within it.
- Pro-activeness – agents do not simply act in response to their environment but they are able to exhibit goal-directed behavior by taking initiative.

Complementary, but not mandatory, attributes are intelligence (goals, reasoning, planning, learning), mobility (remote execution, migration), and social ability (communication, co-operation). These attributes enable the distinction between Intelligent Agents (IA), that can act pro-actively, and Mobile Agents (MA), that can move between different nodes multiple times. Mobile agents are also called Mobile Code. Agents that combine intelligence with mobility are called Intelligent Mobile Agents (IMA).

[3] Yahoo – one of the first and a still very popular search engine within the WWW

[4] Napster – a system for the exchange of audio files

Special languages have been developed that enable agents to communicate not only information but also emotions as they appear in human-to-human communication. One example of those languages is the Knowledge Query and Manipulation Language (KQML). With such characteristics, agents can provide a do-what-I-mean paradigm (DWIM) where a user e.g. tells an agent that it should buy a new video recorder. The agent needs to fulfill several individual tasks to reach this goal: find a video recorder that is technically compatible with the existing television set, bargaining with vendors of video recorders, selecting, asking the users bank for a credit, etc. More complex scenarios can be set up when many agents collaborate to reach a joint goal. [Arbanowski98] [Breugst98]

2.2.4.3. Flexible Infrastructure

Software architectures like Jini and Universal Plug and Play (UPnP) already recognize the demand for scenarios of flexible infrastructure. Looking at mobile networks, the number of not permanently connected devices will increase within the next years. Additionally, the application areas introduced in section 2.2.3 describe environments where services, resources, and user-demands meet unplanned. Such environments need to offer service discovery, registration, and reservation without risking a single-point-of-failure.

Flexible infrastructure also includes an emerging area: smart devices. This class of resources is characterized by intelligence limited to the purpose of the actual device, but flexible enough to allow devices to exchange information and to 'combine forces'. To give a simple example, two lights in a room can be notified by a light sensor that the room is to dark. Both lights can decide to solve the problem in turning themselves on. Another example is a light switch, which is not preconfigured, that interacts with a light to control it. The two devices (switch and light) can detect each other, recognize their function, and configure themselves to provide the simple service of switching the light on and off (or to dim the light). [IBM99] [Rekesh99] [Inet01]

2.2.5. Web Services

Web Services are a development of the World Wide Web Consortium (W3C) [W3C-WS]. It focuses on the integration of the Hypertext Transfer Protocol (HTTP) and RPC as an important step for linking the WWW with RPC-based distributed object oriented systems. The first ideas spread out in 1997, but in 2001 the use of XML for remote operations became prevalent [Lee98][5]. The W3C coordinates a number of activities related to Web Services, including working group (WG) for an architecture and a WG for service description. Furthermore, the standardization of a Simple Object Access Protocol (SOAP) is aligned with Web Services.

SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. At its core, it is XML based. SOAP defines a framework for the definition of a message and its processing. A complete library of documents and specifications for Web Services, including SOAP, can be found in [W3C-WS].

The Web Services Interoperability Organization (WS-I) focuses on the development of standards for web services. [WSI-Intro] indicates that "the fundamental characteristic of web services is interoperability". The organization is established by companies who direct the development of web services. The target environment includes monitoring and analysis tools for control and administration of web services. The WS-I identifies XML Schema, SOAP, UDDI[6], WSDL[7] as basic technologies for web services [WSI-Profiles]. Information about the WS-I can be found at [WSI-WWW].

[5] This reference is an HTML document that was created 1998. The last update to this document was done on February 25th 2002.

[6] Universal Description, Discovery, and Integration of businesses for the web

[7] Web Services Description Language

2.3. General Framework

The General Framework offers concepts and rules for the definition of an architecture. The concepts are expressed in form of objectives and requirements. The main objective is the integration of middleware and management with unified management. This objective leads to a number of requirements, such as the support of services for naming of objects and building of repositories. The rules reflect the concept in a multi-layer model. This conceptual model defines rules that need to be followed in order to realize the concepts. Finally, the concepts and the rules can be viewed together to define an architecture and its components. The General Framework gives recommendations for the realization of identified components. Additionally, this section provides a review of available technologies.

2.3.1. Objectives

The main objective of this framework is the **integration** of management and middleware concepts while maintaining the independence from concrete technologies. The integration serves as a basis to develop service platforms with integrated management facilities that enable applications, services, and resources to be used, controlled, operated, administered, and maintained in a unified way. The independence provides the realization of the framework employing the technology of choice, whereas the decision for the technology can be neither foreseen nor predicted.

Components that belong to the framework should benefit from the integrated and **unified management** in the way that their management can be done by the framework itself (instead of a certain complementary management system).

The second objective is that the framework needs to be **designed for multiple purposes**. The target environment spans from small systems built for a simple purpose (as the usage and control of a home network) up to huge distributed systems covering a multitude of service within a multi-domain environment and many different roles and businesses.

The third objective is to **support the two major activities** (information mapping and system management) as introduced in section 2.1. The framework should offer mechanisms to map information across identified levels and, at the same time, to manage entities of those levels.

The fourth objective refers to the **acceptance of the framework** itself. The history of the Internet has taught that adoption is a better predictor than perfection. The acceptance of the framework is not given for the sake that it integrates middleware and management. Instead, the framework will be used only if it integrates management and middleware up to a reasonable level, employs widely known and accepted methods and technologies, and creates novel functions or improves existing ones.

2.3.2. Requirements

The objectives for the definition of the framework as well as the target environments lead to the definition of the requirements. In general, the framework should enable **interworking**, **portability**, and **scalability**. Interworking reflects the need of business roles and framework components to interact, since the target environment of the framework is a global market of converged services and networks. Portability is a requirement that has to be introduced because the further usage of existing technologies is not certain, emerging technologies have not shown their applicability, and future technologies are yet unknown. Scalability should prepare the framework for distributed systems for the range of small up to huge application areas.

The second block of general requirements relates to the functionality the framework has to support. Since applications, services, and resources are realized by distributed components, the framework itself must support **distribution**. The distributed components can be implemented with various technologies. This leads to the necessity of a framework that is **technology independent**. The integration of distributed systems in the Internet and the WWW imply requirements that are described as **web-enabled** and **directory-enabled**. Last but not least, the mapping of information across levels and the management of each level

follows pre-defined or negotiated methods of actions. The framework should offer mechanism to enable a **policy-based** realization of them.

The handling of distributed systems makes it necessary for the framework to offer the basic functionality that the distribution of components demands. Mechanisms must be identified for the **naming** and **addressing** of components as well as for their **registration**. The approach of a centralized naming service (as a single point of failure) seems not sufficient, especially looking at P2P networks and mobile agent technology. Registered components will seek for services with a certain Quality of Service (QoS). They should be aided by **discovery** and **lookup** services. The communication among components can be arranged in a peer-to-peer manner or by **message** services.

The support of automated control, administration, and maintenance must be based on formal **descriptions** of components and **policies** for their interaction. Policy and **profile services** (or data and type **repositories**) can be mechanisms to realize this. The **visualization** of components, interactions, and data types is a requirement for the manual control of a distributed system.

The technical challenges can be seen in the **interoperability** of components, the **control** of resources that are employed by the framework, and the definition of **formats** for data exchange and evaluation. **Connectivity** and **reliability** of framework services might be based on middleware technology. However, the framework has to ensure them explicitly.

2.3.3. Conceptual Model

The conceptual model identifies four planes. Each plane is dedicated to a specific problem context. Each problem context describes a dedicated viewpoint to the two areas of concern (information mapping and system management) and the five terms (use, operation, control, administration, and maintenance) introduced in 2.1. The planes are used to specify the different types of information that need to be mapped and the different levels of management that is needed. The approach of a conceptual model is taken from the IN standard series [ITU-Q1201].

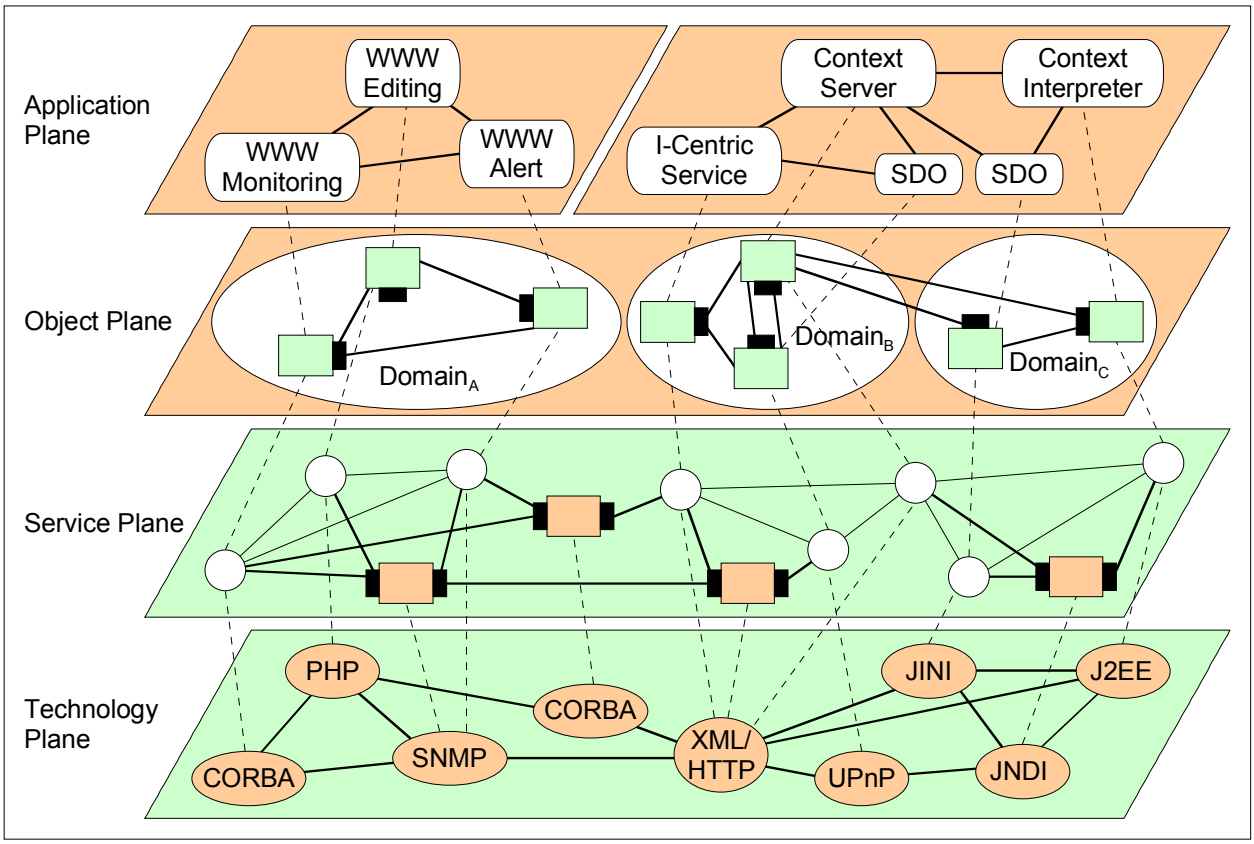

Figure 2-3: General Framework – Conceptual Model

The conceptual model provides the basis for a specific architecture. It presents rules for components of a specific architecture and describes relationships between those components. The four planes of the conceptual model are presented in Figure 2-3. The definitions of each plane can be employed to describe particular aspects of the architecture. The four planes are

- Application Plane, covering functionality within functional blocks;
- Object Plane, modeling functional blocks as computational objects;
- Service Plane, modeling core and application services as computational objects; and
- Technology Plane, identifying the employed technologies and their relationships.

On the first view, the conceptual model separates the applications from technologies. Applications become technology independent. This means, an application is not using a particular middleware or management technology. Those technologies become transparent for the application. The conceptual model allows substituting middleware and management technologies without changing the applications.

However, the conceptual model must not be seen as a dogma. Application might need direct access to technologies. This is especially important for the management of resources. This direct access belongs to the business model and the design of the application. An architecture derived from the conceptual model should not deny such a direct access.

In the vertical axis, Figure 2-3 shows two example distributed applications. The left side of the Application Plane depicts a WWW editing and monitoring system. The right side shows components of an I-Centric communication system [vdMeer01a]. Components of the two systems are shown for each plane. The significant interactions between components within one plane are presented with bold lines. These interactions follow the functionality that the particular plane describes. For the Service Plane, Figure 2-3 shows also the interactions that occur in the Object Plane (normal lines). The dashed lines are used to present the mapping of individual components between planes.

For example, the application component *WWW Monitoring* interacts with *WWW Editing* and *WWW Alert*. In the Object Plane, this component belongs to $Domain_A$. The Service Plane shows that this *WWW Monitoring* has relationships to two services. This can be a naming service to register the component and an event service where notifications are sent to. The dashed line from the component to the Technology Plane shows that *WWW Monitoring* is realized as a CORBA object.

2.3.3.1. Application Plane

The Application Plane focuses on the design and the implementation of applications. An application is an implementation of a set of functionalities that might be distributed over multiple hosts. This set of functionality does not include the support for distribution [TMF-ACT01-99]. Following [TINA-ODL], a group of object consists of object specifications that are called components. An application can be seen as a group of objects. An application component is then a single object specification within a group.

The design of an application can be derived from a business model. Appropriate models and tools can be used for design and implementation. The final implementation can profit from a concrete architecture. Some areas of applications have been introduced in section 2.2.3. The actual purpose of future applications can neither be predicted nor foreseen. The conceptual model gives no further recommendations for this plane.

2.3.3.2. Object Plane

The Object Plane concentrates on the modeling of (distributed) objects. An object is a part of an application that models a real world entity. It is implemented in a computational, identifiable entity that encapsulates state and operations. Attributes are sets of data with a fixed semantic [CORBA] [TMF-ACT01-99]. Figure 2-4 shows an object that offers interfaces with operations. These operations are internally implemented as data and methods.

The interfaces of objects are specified in a certain language. Many middleware and management architectures employ a specific interface language. The languages are combined with tools for automated processing such as compilers and interpreters. The selection of an appropriate interface language depends on the objectives of the specific architecture. Many languages only include the signature of an interface. This

signature might not be enough. Languages from the management area often add semantic information to individual parts of a signature. This information can be used to qualify an interface or parts of it. Furthermore, this information can provide the basis for repositories that enable lookup, monitoring, and configuration of objects.

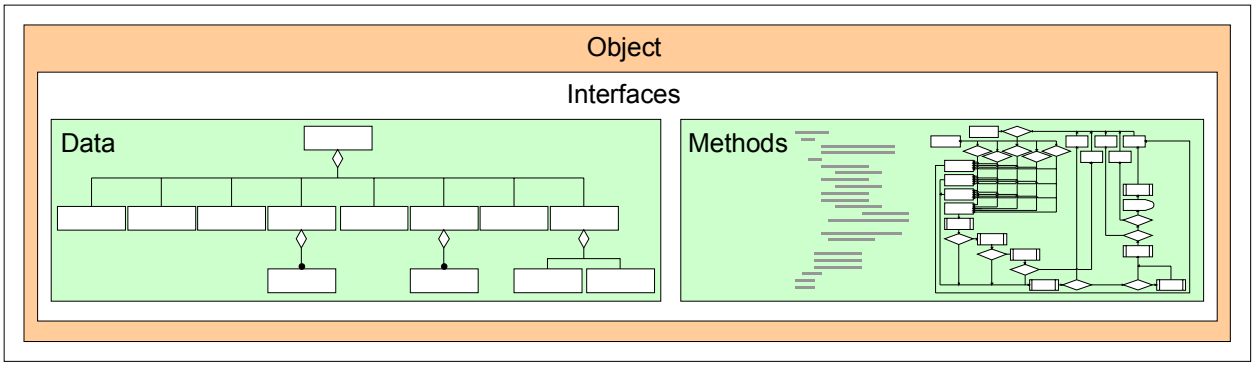

Figure 2-4: General Framework – Object

A specific architecture can support the application design with a set of basic specifications. This can be done e.g. in form of a core model (like the DMTF core model [DMTF-CIM]) or a structure of information (like the Structure of Management Information – SMI; [IETF-RFC2578]).

Beside objects, Figure 2-3 shows the concept of *domains*. A domain can be used to separate (or to delegate) responsibilities. Domains can be implemented e.g. for the collection of objects that deal with different connectivity services (signaling, packet switching), objects that belong to functional areas (like the management functions of [ITU-X700]), or objects that relate to layers of the Reference Model for Open Systems Interconnection (RM-OSI; [ITU-X200]). A domain can also be used to reflect geographical distribution or hierarchies of an organization. The reflection of business models is an important factor for modeling domains.

A protocol is needed to define how interactions between objects can take place. The characteristics of the protocol depend on the interface language. The specification of the protocol should enable an easy mapping to technologies of the Technology Plane. The protocol must enable the identification of called objects. Furthermore, it should provide a reliable and encrypted transmission. An Application Programming Interface (API) can be used to realize protocol mechanisms and to provide access to protocol features.

2.3.3.3. Service Plane

The Service Plane models a collection of interfaces and objects (services) that provide basic functions for (application) objects [CORBA-NS]. A service is independent from the application domain and from objects of the Object Plane. One of the most important services is a naming service, which enables the identification of (available) objects. The target environments introduced in section 2.2 demand for other services. The exchange of asynchronous events can be realized with an event service. Monitoring of objects and longtime configuration management relies on logged events. Ad-hoc networking and P2P communication need lookup services to search for objects that offer a specific functionality.

The basic services that must be provided by a specific architecture are a naming service and an event service. Furthermore, the architecture should offer services that combine information about object instances (naming) with information about object classes (specification). This combination allows the assembly of repositories that ease the administration and maintenance of a system. Additionally, the architecture should offer functionality for the visualization of instances and classes in order to support the manual management of a system.

2.3.3.4. Technology Plane

The technology plane is introduced to model middleware and management technologies. This plane functions as a mediator to the actually employed middleware with specific interface languages, communication protocols, and services. Management systems can be integrated into the architecture and existing

specifications can be reused if available. The technology comprises middleware and management architectures (such as CORBA, Java, Simple Network Management Protocol – SNMP, Telecommunication Management Network – TMN) as well as concrete products.

Figure 2-3 gives an example that includes eight different technologies. Beside two major middleware platforms (CORBA and Java), the figure shows a Java based directory platform (JNDI) and a Java based ad-hoc networking platform (Jini). PHP[8], XML/HTTP, UPnP, and SNMP are included as examples for middleware that is based on standards from the W3C and the Internet Engineering Task Force (IETF).

The connections between the technologies in Figure 2-3 should indicate that there must exist an interworking between these very technologies. This interworking is needed to enable a communication between the objects and services. To give an example: The application *WWW Monitoring* is realized in CORBA and the application *WWW Alert* is based on an SNMP Manager (dashed lines). Both objects should communicate with each other. This communication can be realized with a CORBA/SNMP bridge (horizontal integration within the Technology Plane). Another possible realization is that the application *WWW Alert* would also be a CORBA object that communicates with the SNMP Manager and functions as a gateway by itself (vertical integration realized in the Object Plane and the Technology Plane).

2.3.4. Components of an Architecture

The general framework defines the basic concepts and the conceptual model specifies the basic rules for a concrete architecture. The architecture developed within this work is the Middleware and Application Management Architecture (MAMA). This architecture concentrates on the two middle planes of the conceptual model. The rules of the Object Plane are used to define the basic components of MAMA. The rules of the Service Plane already identify services that need to be realized. The architecture is not going to define methods for business models. Those models are related to the business logic of application, which should be supported but cannot be defined by the architecture. The Technology Plane depicts important realizations and products. The architecture must recognize those technologies and should reuse existing approaches for interworking between different technologies.

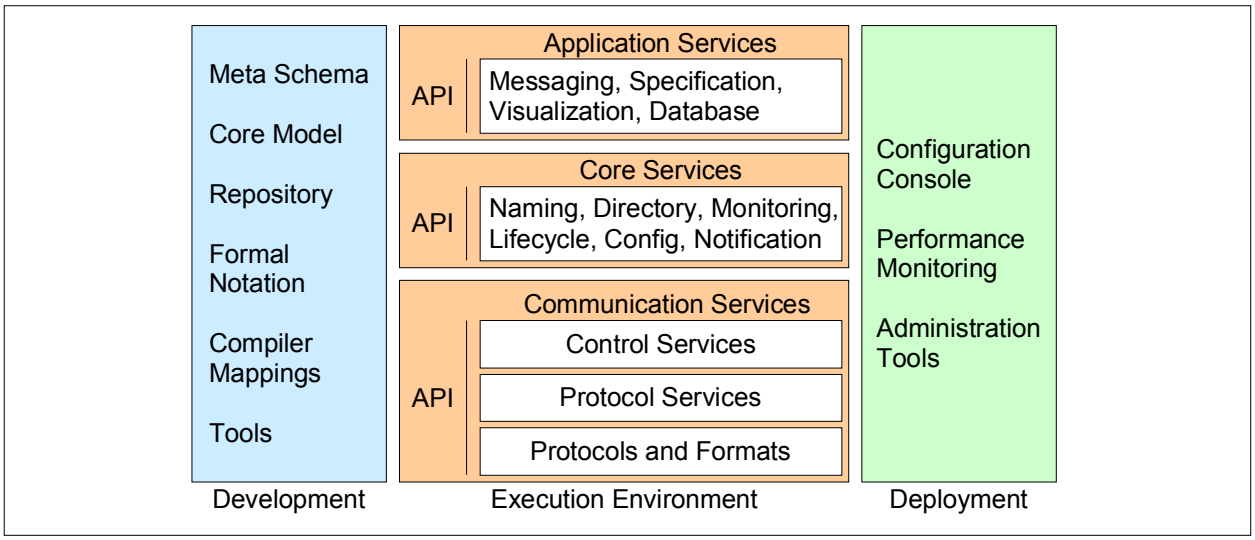

Figure 2-5: General Framework – Components

The components of MAMA belong to the three parts of development and operation of a distributed system. These three parts are depicted in Figure 2-5. They are *development*, *execution environment*, and *deployment*. The development step follows a business model. MAMA should support the development with an appropriate formal notation for the specification of objects. This specification is based on a meta schema (or object model), which needs to be designed in a way that reflects the needs of the target envi-

[8] PHP Hypertext Preprocessor

ronments (cf. section 2.2). A core model can be specified using the formal notation. Task of the core model is to provide basic definitions for a distributed system, such as often used data types and generic objects. The formal notation defines syntax and semantic of a language that is derived from an object model. Here, the mapping of specifications to concrete middleware technology, programming languages and compilers must be specified by MAMA. This mapping can be supported by tools for an automated processing of specifications.

The second step is the execution of the distributed system. This is supported by MAMA in form of an execution environment, which provides an interface to the Technology Plane. Figure 2-5 shows the execution environment in form of three layers. Each layer provides a unified access to its functionality via one or more APIs. The lowest layer realizes the communication between objects and between objects and services. For this communication, MAMA needs to define a protocol along with data formats, and control services. The protocol defines the communication behavior, including the transmission of data. The data formats are used by the protocol for the transmission of information specified in the formal notation. The control services can be used to include additional functionality for addressing, security, and transactions. The protocol must be specified in a technology independent way. This should allow the usage of many different technologies for the actual exchange of data.

The services of the Service Plane are further separated in two groups. The first group collects core services, which *must* be present in the execution environment for usage and operation of a distributed system. The second group depicts services that *might* be present. This second group of services should improve the architecture's ability to support control, administration, and maintenance.

The final step is the deployment of the objects and the distributed system itself. Here, a number of tools should be provided for the configuration of the system. These tools can be offered in form of an administration application. This application should be able to visualize information about the actual state of objects, including instantiated objects, request counts, runtime behavior, monitoring, and log information. The tools should be based on core and/or application services. Following this approach, an administration tool by itself is a distributed application that can be modeled using the conceptual model and the mechanisms of the architecture.

The following subsections discuss state of the art technologies for the components of the architecture. These technologies are parts of actually available middleware and management architectures. Section 2.3.4.1 starts discussion with an overview of object models and meta schemas. Section 2.3.4.3 concentrates on interface languages and syntax notations. An object model and a formal notation provide the basis for the definition of repositories. Section 2.3.4.2 shows that a repository represents a virtual data store that combines specifications (object classes and related definitions) and runtime information (object instances).

The sections 2.3.4.5 and 2.3.4.7 review approaches for protocols, data formats, communication services, and APIs to define the lowest layer of the execution environment. Section 2.3.4.6 is dedicated to core and application services. Tools and user interfaces to application services should support the deployment of applications. Other tools for the deployment of applications are not discussed explicitly. They belong to the concrete target environment. However, the specifications of MAMA should give recommendations for those tools.

2.3.4.1. Object Models

An object model is the composition of interacting objects that concentrates on clearly identified aspects of the real world. Object models can be abstract or concrete. An abstract object model identifies basic characteristics of objects [OMG-OMA]. The client/server paradigm ([Orfali96]) and the manager/agent relationship ([Hegering99]) can be denoted as abstract object models. A concrete object model adds specific recommendations and rules. A concrete object model identifies the semantic of objects. Furthermore, it can restrict the abstract model by eliminating entities or placing additional restrictions [CORBA].

The object models of [CORBA] and [TINA-CMC] define an object as presented in section 2.3.3.2. In general, these characteristics apply also to managed objects. In detail, managed objects are further restricted. They provide an abstraction from a physical resource (network component) or logical resource (e.g. profiles). The operations of a managed object are often predefined in the object model. SNMP declares access policies for attributes (read-only, read-write, read-create, not-accessible; [IETF-RFC2578])

that directly influence set and get operations of the protocol. X.700 has standard operations defined for all managed objects (create, delete, action) and standard operations for attributes (get, replace; [ITU-X720]).

Furthermore, managed objects emit notifications. Notifications are closely related to the resource that is modeled by the managed object. This is an important characteristic of the manager/agent relationship. [Hegering99]

The CIM Meta Schema allows describing object instances [DMTF-CIM]. This feature enables a designer to specify a distributed system in a very restricted way, including a set of object instances. This approach makes configuration management easier and solves some issues of an initial start-up of a system.

A Reference Model

The Reference Model for Open Distributed Processing (RM-ODP; [ITU-X901], [Raymond95]) can be used as a guideline for the design of distributed systems. The computational and the engineering viewpoint can be employed as a basis for concrete object models. The left side of Figure 2-6 shows a computational object, the right side an engineering object. [Linington95] states that "Objects can have any number of interfaces. This ability to partition the observable behavior of an object between multiple interfaces gives a valuable tool for structuring the specification of an object's behavior."

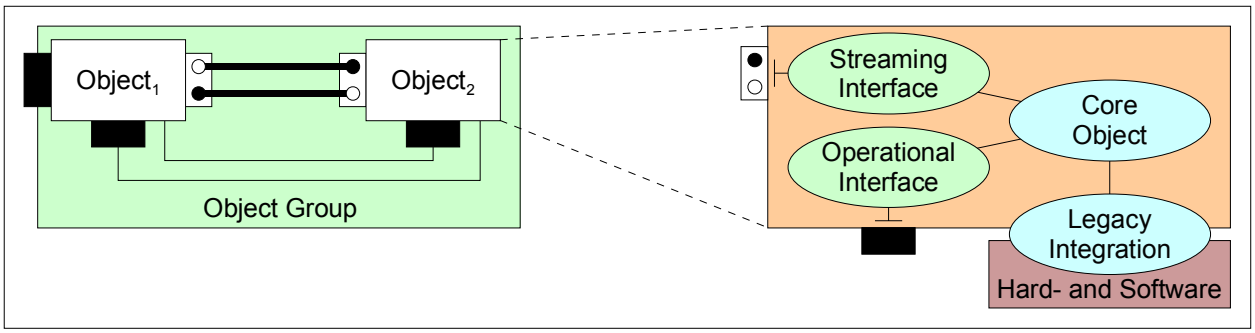

Figure 2-6: Computational and Engineering Objects

Figure 2-6 shows $Object_2$ with one computational and one streaming interface. Furthermore, the mapping to engineering objects is presented. These engineering objects decouple the interfaces (interface objects) from the implemented behavior (core object) and legacy technology (legacy integration).

Semantic Information

The concrete object models of middleware offer mechanisms that describe the syntax of an object. TINA adds the two textual descriptions *behaviorText* and *usage*, which should be used to explain behavior and usage in a natural language of choice [TINA-CMC] [TINA-ODL]. SNMP calls such a description a *definition*, which must be used to explain the minimum requirements on an object's implementation [IETF-RFC2578].

The CIM meta schema introduces qualifiers to characterize objects and other named elements. This mechanism allows the CIM meta schema to be "extensible in a limited and controlled fashion" [DMTF-CIM]. Adding new qualifiers increases the availability of meta data about a schema, which can be automatically processed in a particular management environment.

Attributes and Parameters

Attributes and parameters of interfaces belong to specific types. The permitted types are usually identified in the concrete object model. Middleware distinguishes between basic types and constructed types e.g. described in [CORBA]. Basic types are in many cases similar to basic types of programming languages, like different kinds of integers, floating point numbers, characters, and Boolean types. Constructed types can be records (struct), discriminated unions, sequences of other types, arrays, interfaces etc. The description of the semantic of types is mostly limited to ranges (e.g. for integers), minimum/maximum boundaries (e.g. for arrays), and the memory allocation (e.g. characters).

Management object models define a more restricted semantic of types. E.g., SMI uses one type for IP addresses, two types for counters, and one type for time ticks [IETF-RFC2578]. The CIM meta schema includes a type for date and time information [DMTF-CIM]. [ITU-X721] defines generic types for attributes, actions, and notifications. Generic attribute types include counters (simple and settable), gauge (dynamic variable), and tide-mark (minimum or maximum value of a gauge during measurement time). Each implementation of such an object model is expected to support the defined types along with their semantic. Interworking between management systems is only guaranteed when these types are used.

Object Identifiers

For management, administrative policies are used for assigning identifiers. These policies are either defined directly in the object model (SNMP in [IETF-RFC2578]) or belong to the characteristic of employed technologies (e.g. distinguished names for directories as defined in [ITU-X720]). These identifiers are further used to generate repositories and to enable interworking between different implementations.

Rules for Object Instantiation

In middleware, an interface represents a syntactical description of a service that is offered to clients. An object satisfies an interface when it provides the service according to the operations of this very interface. Management defines more restrictive rules for the instantiation of objects. An OSI managed object must support all the attributes, operations, behavior, and notifications specified in all mandatory packages. Furthermore, it exists, from a management point of view, if it has a distinguished name and supports the operations and notifications defined for its class. Otherwise, it does not exist from a management point of view, even if a physical counterpart exists [ITU-X720].

Object orientation

Object-orientation is based on at least two key issues: *encapsulation* and *inheritance*. Encapsulation reflects the fact that objects distinguish between their interfaces and their implementation. Inheritance provides the basis for code-reuse and for code-clarity [Stroustrup92]. The most management object models do not allow multiple inheritance [Hegering99]. As [CORBA-MAN] states: “Real embedded interfaces don’t differ in the services they offer, they differ in the entities they manage … In object-oriented terms, services represent a case in which inheritance is not appropriate.”

However, not all object models must follow an object oriented approach. The object model of SNMP does not include any kind of object-oriented design mechanism. SNMP managed objects are declared, implemented, and used as a set of variables. The variables can be structured in tables. A table might contain any number of columns and each column represents a single variable (managed object) [Zeltserman99].

2.3.4.2. Repositories

Management Information Bases and Interface/Object Repositories are (usually distributed) virtual data stores that manage information about an object-oriented system. A MIB defines the naming conventions for stored objects. In middleware, the term repository describes a database that holds interface/object signatures. For DCOM and CORBA, these repositories contain information about an object’s signature and its actual implementation. Middleware for the control of appliances, such as Jini and UPnP, add information about the semantic of interfaces/objects.

Management architectures define a central naming scheme in which names (or parts of names) are assigned by an authority. The names are arranged in a hierarchical structure reflecting a hierarchy of managed objects. SNMP uses an OBJECT IDENTIFIER that is constructed of a number of labels [IETF-RFC1157]. The SNMP standards track demands the implementation of several parts of the MIB at each agent. The implementation of the system group, e.g., is mandatory [IETF-RFC1213]. For OSI management, a name binding must be assigned to each object specification [ITU-X720]. Furthermore, an Object Identifier Tree (OIT) is defined as basis for a consistent object naming [ITU-X722].

The CIM mechanisms for naming and object databases facilitate the task of sharing management information between a variety of platforms. The major issue of naming is the enterprise-wide addressing of objects. Object databases based on CIM naming are employed to realize a MIB-like instrumentation

[DMTF-CIM]. The creation of different scope hierarchies, regarding the actually used models, must be able to be changed over times. This does not permit a single, standardized MIB as of SNMP and X.700 management systems.

2.3.4.3. Formal Notations

Formal notations are languages used for the specification of objects and their interfaces. They do not depend on actual implementations. Those languages are built of a number of definitions that explain syntax and semantic of an object model (or a meta schema). A formal notation is defined by a grammar, usually a variant of the Backus-Naur Form (BNF), such as Augmented BNF (ABNF, [IETF-RFC2234]) or Extended BNF (EBNF, [ISO14977]). Notations for managed objects are often based on the Abstract Syntax Notation 1 (ASN.1) [ITU-X208]. Almost all specifications are coded as plain text files. This gives a number of intrinsic characteristics:

- Platform independence (every platform is able to handle plain text files);
- Extendibility (every platform supports tools for editing plain text files);
- Automated processing (no additional re-formatting is necessary); and
- Readability (as long as the reader is familiar with the formal notation).

A formal notation defines conventions for permitted character sets (ASCII[9], Unicode, ISO[10] Latin-1), identifiers, keywords, comments (single line, multi-line), and preprocessing. With particular language elements, the definition of naming and scoping rules is established.

The formal notation that are of interest for MAMA can be categorized as follows.

- Interface Definition Language (IDL) – is used for the specification of interface signatures in DCE[11], CORBA, and DCOM. [DCE-RPC] [CORBA]
- Object Definition Language (ODL) – enables the description of ODP computational objects with interfaces, object groups, and a number of templates. [TINA-ODL]
- Languages for the specification of managed objects – are employed by management architectures. Examples for those languages are SMI, the Guidelines for the Definition of Managed Objects – GDMO, and DMTF Managed Object Format – MOF. These languages are either a subset of ASN.1 (SMI [IETF-RFC2578], GDMO [ITU-X722]) or an extension of IDL (DMTF MOF; [DMTF-CIM]).
- Languages for generic data exchange – XML and derived languages provide technology independent mechanisms for data exchange. As a meta language, XML offers the ability to specify domain specific languages that can be processed with standardized tools.

All languages are designed following a concrete object model. The applicability of the languages within this thesis depends on the concrete object model of MAMA. In general, languages from all categories can be used. However, each category has specific characteristics that need to be considered.

The object models of CORBA and DCOM do not distinguish between object and interface. Their IDL does not support more than one interface per object. With ODL, multiple interfaces per object can be realized. Both languages give no recommendations or restrictions for operations and type definitions. Languages from the area of management primarily support generic interfaces for specific management functionality. This includes meta information on semantic of objects, interfaces, and attributes as introduced by an object model. IDL and ODL do not offer such a facility.

Object identifiers are almost case-insensitive. This means, two identifiers collide when they differ only in the case of their characters. Case-insensitive identifiers already support scripting languages, which do not distinguish between upper-case and lower-case characters. [DMTF-CIM]

In IDL, attributes can be declared as read-only. Parameters of operations can be declared as in, out, or both; clarifying in which direction a parameter should be transmitted [ITU-X920]. Furthermore, map-

[9] American Standard Code for Information Interchange

[10] International Standardization Organization

[11] Distributed Computing Environment

pings from IDL to programming languages are defined [CORBA]. ODL adds templates for the creation of interfaces, objects, and object groups. These templates serve for “software distribution and modularity” [TINA-ODL]

SNMP goes a similar way offering an object-type macro that should be used to define SNMP managed objects [IETF-RFC2578]. The OSI management framework provides a complete set of guidelines. GDMO includes templates for objects, attributes, behaviors, and notifications [ITU-X722]. Each managed object can be assembled with packages. A package can be declared as conditional, which offers a policy for object instantiation. Each managed object is accompanied with a name binding [ITU-X722].

The DMTF MOF language adopts many IDL features (from DCE IDL). Special compiler directives are available to include paths in a global name space for MOF object classes and instances [DMTF-CIM].

The combination of SOAP and XML promises a mechanism to access distributed systems from the WWW. XML was originally defined as “[…] a method for putting structured data in a text file” [W3C-XML-10P]. Basically, XML defines no objects or interfaces but documents. Structure of information and content are separated. Structure of information can be declared with tags [ISO8879]. A Document Type Definition (DTD) defines constraints on storage layout and logical structure. XML further distinguishes between well-formed (XML conform) and valid (conform to a well-formed DTD) documents [W3C-XML].

The use of ASN.1 as basis for languages from the management area is currently discussed in the IETF. The discussion started with the work on a new version of SMI (SMIng; [IETF-RFC3216]), which incorporates management and policy schemes. [Draft-SMIng] defines a new language that depicts several features from object-oriented languages and from programming languages. On the other hand, [Draft-ASN1NG] proposes the usage of ASN.1 for the new version of the SMI. The IETF Draft is conforming to the requirements identified in [IETF-RFC3216].

2.3.4.4. Development Tools

Development Tools play an important role in the development process. They offer a Software Development Kit (SDK) or an Integrated Development Environment (IDE). Both are usually toolsets that include editors, debuggers, and documentation for a programming language, middleware, and/or management architecture. SDK and IDE usually allow the integration of external tools for specific tasks. Following this approach, a compiler for the formal notation of MAMA can be integrated in an existing development environment. Therefore, a specific development environment will not be depicted for MAMA.

2.3.4.5. Communication Services, Protocols, and Formats

Protocols

A protocol defines mechanisms for the exchange of data between distributed objects. [ITU-X210] recommends a reference model that includes the basic definitions of a protocol and protocol services. Information is transmitted in Protocol Data Units (PDU), which contain protocol information (header) and the actual data (payload). PDUs can be designed for specific service primitives [ITU-X210]. [Tannenbaum96] and [Badach94] explain many protocols in detail.

SNMP defines a protocol in [IETF-RFC1157] as “an application protocol by which the variables of an agent’s MIB may be inspected or altered. Communication among protocol entities is accomplished by the exchange of messages, each of which is entirely and independently represented within a single datagram using the basic encoding rules of ASN.1”. The OSI management framework recommends a complex set of entities, services, and protocols for the exchange of management information. [ITU-X701] [ITU-X710]

Another example of a protocol is HTTP. The HTTP is an application level protocol [IETF-RFC2616]. It is based on an asynchronous communication model and realized as a request/response protocol. Communication is organized with the simple exchange of messages in form of documents. Access to legacy systems is provided by gateways and proxies. This protocol can serve as communication protocol for other application protocols (such as the Simple Mail Transfer Protocol – SMTP or the File Transfer Protocol - FTP).

Data Encoding

An important issue for data transmission is data encoding. [CORBA] defines a transfer syntax (Common Data Representation – CDR) that maps ordinary IDL data types to a bi-canonical, low-level representation. The Basic Encoding Rule (BER) formulates encoding rules for the transmission of ASN.1 typed information. Application level protocols, such as FTP and HTTP, are often based on text messages. The actual transmission of these messages is realized with transport protocols. Special transport mappings are used to define how information "maps onto an initial set of transport domains" [IETF-RFC1906].

[Palme02] provides a comparison of ABNF, ASN.1 BER and DTD-XML. The excerpt compares characteristics regarding the level of coding, the encoded format, the readability of the code, and the efficiency of data packing, binary data, and layout facilities. One example is given for textual encoding measured in octets. This example shows that ASN.1 BER is the most efficient language (61%) followed by ABNF (19%) and XML (11%). [Palme02] has recently specified an encoding for XML data, which promises to optimize the encoding of XML data for transmission.

Message Formats

Message formats provide a facility to generate requests, to locate objects, and to manage communication channels [CORBA]. For management protocols, these message formats are predefined in form of a set of PDUs (SNMP; [IETF-RFC1905]) or a combination of PDUs and service elements (OSI, [ITU-X710]). For middleware protocols, these message formats realize a subset of interactions needed for a client/server relationship [CORBA] [DCE-RPC].

SNMP uses a mechanism called *variable binding* for the payload of a PDU. This is a simple list of variable names and corresponding values. Some PDUs are concerned only with the name of a variable and not its value (e.g., the GetRequest-PDU). In this case, the value portion of the binding is ignored by the protocol entity. [IETF-RFC1157]

Protocol Services

An SNMP request is atomic, which means it is either completely successful or not at all. More sophisticated transaction protocols are the 2 Phase Commit Protocol (2PC) and the 3 Phase Commit Protocol (3PC). They are able to operate a unit of work over multiple objects. Commit and abort messages are exchanged to signal a successful operation of a complete unit or to instruct a rollback [Heuer97]. The Common Management Information Service (CMIS) offers a parameter that indicates in which manner operations over multiple (managed) objects are to be synchronized. This parameter can be set to atomic (similar to SNMP) or best effort. [ITU-X710]

The Common Management Information Protocol (CMIP) allows the collection of managed objects for each operation [ITU-X710]. Such a selection involves two phases: scoping and filtering. *Scoping* entails the identification of the managed objects to which a filter is to be applied. *Filtering* entails the application of a set of tests to each member of the set of previously scoped managed objects to extract a subset. The subset of scoped managed objects that satisfy the filter is selected for the operation. Four specifications of scoping level are defined, enabling to address managed objects in a management hierarchy: the base object alone, the n^{th} level subordinates of the base object, the base object and all of its subordinates down to and including the n^{th} level and the base object and all of its subordinates (entire sub tree).

2.3.4.6. Core and Application Services

Naming and Directory Services

Naming services support the identification of objects. The naming service provides a name-to-object resolution. Each object must register itself at a naming service. Other objects can retrieve information about registered objects, at least their system specific address. Commonly used naming schemes are object references (in CORBA), hash identifiers (in Java Remote Method Invocation – RMI), or distinguished names (in OSI, TMN). IP networks use a different naming scheme. Here, a name is a set of labels with clear boundaries (SNMP, Domain Name System – DNS; [IETF-RFC2929]).

Naming schemes can be based on name spaces that are managed by an authority (Universal Resource Locator – URL [IETF-RFC2396], X.500 distinguished names [ITU-X501]). Other mechanisms, like the

CORBA Interoperable Object Reference – IOR ([CORBA]) and the DCE Universally Unique Identifier – UUID ([DCE-RPC]), do not involve such an authority. They are based on generic algorithms.

A certain protocol can be assigned to a name, which indicates how an object can be accessed. A CORBA IOR can be assigned to the Internet Inter-ORB Protocol IIOP, RMI, or DCE. Furthermore, a CORBA IOR can be expressed in form of a URL [CORBA-NS]. A URL can be assigned to any available protocol. Furthermore, the access to a resource addressed by a URL might involve more than one protocol. For access to an HTML[12] document a client needs to request a host name from the DNS and than the document via HTTP [IETF-RFC2396]. Names need not to be interpreted by an application. This is task of the implementation of the protocol.

Names can be further integrated into a directory service. Such a service employs directory names. An entry in a directory can be aliased, which means more than one entry can be assigned to the same object without replicating this very object. Access to a directory service is offered to clients via a protocol [ITU-X500] or an API [JNDI-API] [CORBA-NS].

State of the art directory services, such as LDAP, JNDI, and Active Directory (AD), are realized as distributed services. They offer a single point of access to clients and manage distributed databases, which can rely on different technologies [DMTF-CIM]. The management of these databases involves two important concepts: *referrals* and *replication*.

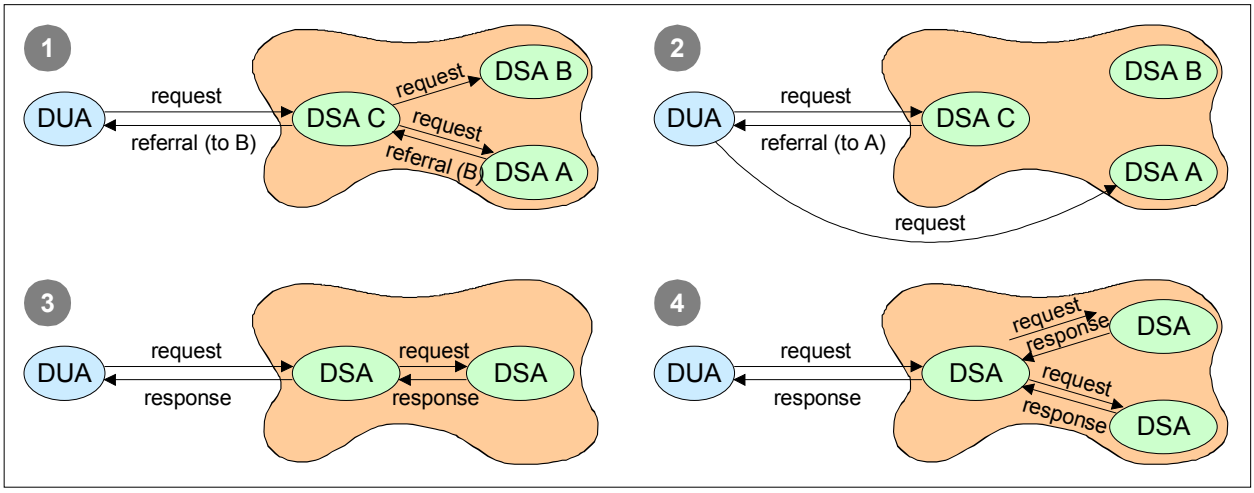

Figure 2-7: Distributed Directory and Referrals [ITU-X501]

The DNS, X.500, LDAP, and JNDI realize the search operations in a distributed directory with referrals. Figure 2-7 shows four cases for the usage of referrals as defined for X.500 and LDAP. In case 1, the Directory System Agent (DSA) C handles a referral from DSA A to retrieve the requested information from DSA B without further interactions with the client. In case two, the original client's request is answered with a referral to another directory server. The client is responsible for a consecutive request. The cases 3 and 4 show two approaches for chaining. Case 3 depicts uni-chaining (one request is passed through several DSAs). Case 4 shows a multi-chaining example (one request is forwarding it to two or more other DSAs). [ITU-X500]

The concept of replication was introduced to improve the availability and reliability of a directory service. Replication is employed under every circumstance that denies an LDAP server the appropriate processing of incoming requests. The concept comprises clients, masters, and slaves. The master, a special LDAP replication daemon, offers the replication service to slaves. Modifications made by clients on the master LDAP server are automatically forwarded to all relevant slaves. [Draft-LDUP]

An example for a directory service that covers multiple technologies is JNDI. Here, a name is related to a naming context, which is the starting point for exploring a namespace. An initial context contains a variety of name bindings from one or more naming systems [JNDI-API].

[12] Hypertext Markup Language

Visualization

Human-Computer Interaction (HCI) analysis defines design alternatives, data structures, and evaluation of human factors in terms of efficiency for a variety of tasks. Visualization and navigation on structured data must be seen as a single entity. In the Curricula of HCI [Hewett96], the work is described as: "Human computer interaction is a discipline concerned with the design, evaluation, and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them."

In the actual world of computing, human beings interact more with information objects and less with computers. Hereby the computer is a tool that is used for the interaction with information objects. HCI provides skills and techniques for information design and makes computers more and more invisible. Interactions are split into the human side and the mechanism side for one particular machine. The main fields of HCI are [Hewett96]:

1. the tasks by humans and machines;
2. the structure of communication between human and machine;
3. human capabilities to use machines (including the *learnability* of interfaces);
4. algorithms and programming of the interface itself;
5. engineering concerns that arise designing interfaces;
6. the process of specification and design; and
7. the implementation of interfaces.

The first and central question at the beginning of a HCI process should be: "What is the goal and how can it be achieved?" This question can be answered with a human-machine categorization followed by a task analysis. One analytical method follows the GOMS[13] model [John96]. It provides a guideline to investigate tasks in the terms of goal-directness, the amount of required skills, the control of interactions, and the interaction sequences.

The characterization of tasks in the context of other tasks is another key concern in HCI [Wright96]. Control of interaction can be done in an active or a passive context. In the active context, the machine requires a direct action (e.g. a video game). In passive contexts, tasks can be performed by the user at will because no direct response is necessary (e.g. web browser).

The intended functionality is by, for example, commands a user can enter and or/items he can selection completing a sequence of interactions. Input and output devices and techniques should be predefined. The relevant input techniques which best match with user requirements are to be determined. This can be keyboard-based (e.g. commands, menus), mouse-based (e.g. picking, rubber-band lines), pen-based (e.g. character recognition, gesture) or voice-based. Operating systems usually offer guidelines ([Schlosser97] [MSDN-UI01]) for graphical user interfaces that are supported by class libraries ([JAVA-J2SE]).

Dialogue is the technique for interaction with humans. [DIN96] describes sound issues that need to be considered. Dialogues should work adequate, self describing, controlled by hand, reaction conform, error tolerant, and individual to adapt. They should also assist in learning to control an application.

The interaction can be done in a primary or secondary dialogue with a modal or modeless category. A document based dialogue can be realized as a single document interface (SDI) or a multiple document interface (MDI) [Balzert99]. New developments and newly available technologies allow the application of avatars and voice-based interactions with the user.

Some basic concepts of computer graphics are useful for HCI, too, like the use of color, 2-D and 3-D spatial organization. The graphic representation of data can be form based, diagram oriented, iconic, or a combination of all three representations [Tiziana96].

Management Services

[ITU-X700] defines five functional areas for system management: Fault, Configuration, Accounting, Performance, and Security (FCAPS). A variety of system management functions is defined in the ITU recommendations X.730 up to X.799. These management functions can be applied to control, administra-

[13] Components of a design model : Goals, Operators, Methods, Selection rules

tion, and maintenance of applications, services, and resources. Service platforms enhance the five functional areas with e.g. functions for subscription and service management [TINAC]. Here, the management function is applied to the management of one particular layer.

Fault management encompasses fault detection, isolation, and correction of abnormal operation of the system. Faults cause systems to fail to meet their operational objectives. Faults can be persistent or transient. Faults manifest themselves as particular events. Error detection provides a capability to recognize faults. Fault management includes functions to maintain and examine error logs, accept and act upon error detection notifications, trace and identify faults, carry out sequences of diagnostic tests, and correct faults.

Configuration management identifies, exercises control, collects data, and provides data that allow management systems to initialize, start, operate continuously, and terminate services and network elements. This includes functions to control the routine operation, associate names with managed objects, initialize and close down managed objects, collect information on demand about the current condition, obtain announcements of significant changes in the condition of the system, and change the configuration.

Accounting management enables charges to be established for the use of resources in the system and for costs to be identified for the use of those resources. This includes functions to inform users of costs incurred or resources consumed, enable accounting limits to be set and tariff schedules to be associated with the use of resources, and enable costs to be combined where multiple resources are invoked to achieve a given communication objective.

Performance management enables to evaluate the behavior of resources in the system and the effectiveness of communication activities. This includes functions to gather statistical information, maintain and examine logs of system state histories, determine system performance under natural and artificial conditions, and alter system modes of operation for conducting performance management activities.

Security management supports the application of security policies by means of functions that include the creation, deletion, and control of security services and mechanisms, the distribution of security-relevant information, and the reporting of security-relevant events.

2.3.4.7. Application Programming Interfaces

An API is a convention by which applications gain access to operating systems or other services [FODC]. The first APIs have been defined at source code level, as an integral component of operating systems. Up to now, the intention of APIs has not changed. However, the concepts have evolved from technology dependent towards technology independent developments.

The Parlay API is an open and technology-independent specification for distributed telecommunication applications. The intention of the Parlay API is to help application developers to work out new application and service possibilities independent of network and infrastructure developments. The API facilitates the *migration of new protocols* as well as the *increase of portability* of telecommunication applications. [Parlay-API]

A middleware architecture and its APIs offer an abstraction from underlying hardware and software. Here, the API's major task is the translation of parameter lists from one format to another. Furthermore, the API is responsible for the interpretation of call-by-value and call-by-reference arguments in one or both directions. [Orfali96]

One advantage of XML is the availability of two different APIs for the processing of XML documents. The Document Object Model (DOM) is a widely used, tree structure-based API issued as a W3C recommendation [W3C-DOM98]. DOM is best applicable for applications that modify the structure of XML documents and for sharing access to the DOM tree with other applications. The Simple API for XML (SAX; [SAX]) represents an event-driven API, which is not supported by a standardization body. SAX is best applicable when an application deals with large XML documents that do not fit in memory, for counting the total number of elements in a document, and for extracting the content of specific elements. [Maruyama00]

Chapter 3
Approach

The general framework developed in the last chapter provides the basis for the Middleware and Application Management Architecture. This chapter starts with the definition of the architecture, which is assembled from six parts that cover the requirements of the framework. The sections 3.1 up to 3.6 are dedicated to one part each, which in combination form the architecture. Each part is discussed in a similar way. This begins with the objectives, followed by the relevant specifications and definitions, and finalized by additional issues and concluding remarks. Section 3.7 concludes with this chapter with recommendations for the design of applications using the advantages of the solutions.

The Middleware and Management Architecture (MAMA, cf. Figure 3-1) concentrates on the two middle planes of the conceptual model that was introduced in section 2.3.3. The *Object Plane* of the conceptual model provides concepts and mechanisms for this design process. The *Service Plane* adds the necessary core services and enhanced services for integrated management. Both planes offer technological transparency to the applications.

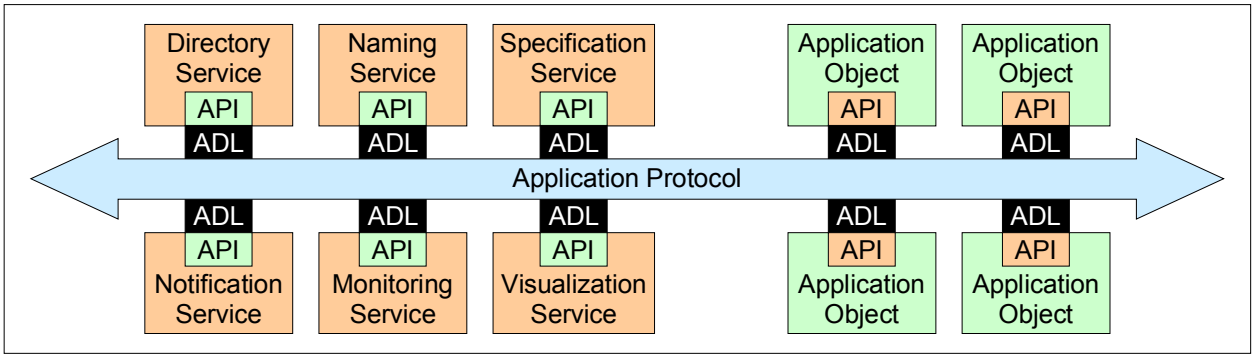

Figure 3-1: Middleware and Application Management Architecture

The applications are not bound directly to middleware and management technologies. Six recommendations form the basis to realize this transparency:

1. The basis of MAMA is an abstract object model. A concrete object model, called **Meta Schema**, is derived from the abstract object model. The Meta Schema defines the basic specification elements of a MAMA application.
2. The **Application Definition Language** (ADL) is a formal notation that is used to express the Meta Schema. This language combines characteristics from middleware interface definition languages, languages for the definition of managed objects, and languages used to specify data that is exchanged between applications.
3. Application objects are specified in ADL. MAMA defines a **Schema** based on ADL and a **Core Model** that is relevant for all applications and independent of domain specific specifications. The basic part of the Core Model is the identification of a reasonable set of qualifiers providing meta information about ADL typed application objects.
4. An **Application Protocol** is responsible for the transport of information among application objects including features that enable the construction of management hierarchies like addressing of hierarchies, scoping and filtering, and transactions.
5. An **Application Programming Interface** (API) decouples application objects from middleware technology and enables the seamless integration of management functionality into applications. The API offers a small and simple to use set of operations for the parameterization of the protocol. Tools

are responsible for the automatic generation of parts of the API code that is needed for the exchange of information between applications.

6. **Application services** realize the naming of objects; enable the mapping of ADL specified information to directories, and the usage of type and data repositories for applications and MAMA components. All application services can be accessed in a unified way similar to the access to other applications. The API implements standard jobs like lookup for available services and registration.

The Meta Schema, the ADL and the Core Model take advantage by using existing mechanisms and techniques and, simultaneously, enhancing them to meet the requirements and objectives of the general framework. They are neither specific to middleware nor management. The Application Protocol and the API provide a generic interface to the application. Internally, they map this generic interface to middleware specific interfaces that need to be adapted to any employed concrete middleware technology.

The protocol and the internal realization of the API are transparent to the applications. Any application developed with MAMA is ready to run on any middleware that MAMA supports. The support of a new kind of middleware has some requirements. First, the MAMA API must support the new kind of middleware. This is realized with a specific implementation of the API that recognizes the available facilities of the middleware. Second, the new kind of middleware must be integrated in the existing middleware technologies. The integration reflects the Technology Plane of the Conceptual Model. MAMA applications that are designed to communicate with each other must be supported by gateways between the middleware technologies (cf. example of section 2.3.3.4). When these two requirements are fulfilled, the new kind of middleware is available for all MAMA applications. It is also possible to substitute the actually employed type of middleware without changing the applications.

3.1. Object Model

The MAMA object model builds the basis for the design of a MAMA application. It combines characteristics from middleware object models and from object models from management architectures. The object model has the following objectives:

- Define an abstract object model.
- Derive a concrete object model.
- Provide the basic functionality for a formal notation.

The concrete object model identifies the basic characteristics of a MAMA application. An application is a piece of software that solves a specific task. An application consists of one or more components, namely application objects. These objects are modeled following the abstract object model.

The concrete object model represents a formal description of the abstract object model. It follows the facilities of the object model of the Common Object Request Broker Architecture (CORBA; [CORBA]) and the Meta Schema of the Common Information Model (CIM; [DMTF-CIM]). For MAMA, the concrete object model is expressed in form of a meta schema in the Unified Modeling Language (UML). The UML specification provides the basis for a formal notation, which is used to specify application objects.

3.1.1. Abstract Object Model

Application objects and core services are handled in the same way. That is, core services profit from the advantages of MAMA similar to regular applications. Application objects are modeled as computational objects by means of the computational viewpoint of the Reference Model for Open Distributed Processing (RC-ODP; cf. section 2.3.4.1). Their interfaces are specified in ADL. This approach of ADL-typed, middleware independent interfaces does not change the design of computational objects. In fact, the mapping of computational objects to engineering objects and the basic engineering objects themselves follow a new approach.

The left side of Figure 3-2 shows a classic ODP computational object with its engineering objects for interfaces and the core object. The interface objects realize middleware specific interfaces such that are

implemented using a specific CORBA product. The right side of Figure 3-2 shows a MAMA computational object. Here, the core object is supported by the API and MAMA interface objects.

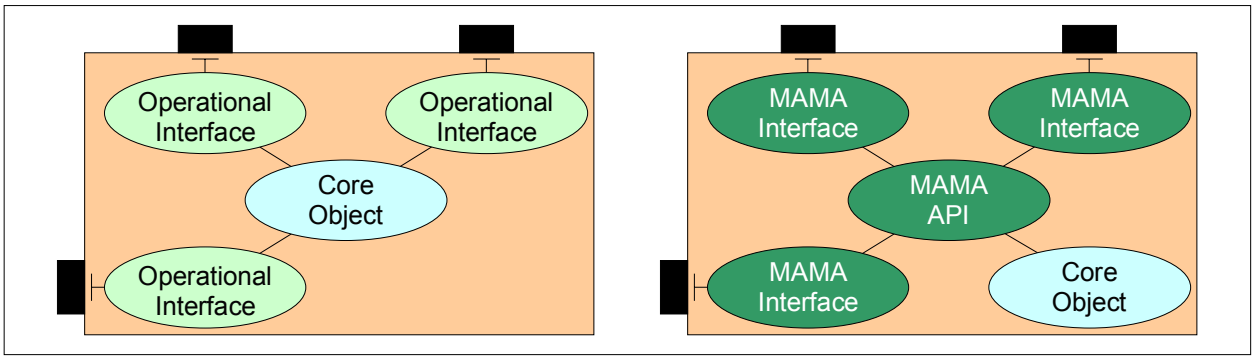

Figure 3-2: Object Model – ODP vs. MAMA Computational Object

The API is a supporting engineering object modeling an abstract interface for the core object to any offered features of the architecture. The MAMA interface objects are responsible for the realization of the Application Protocol. They are further mapped to concrete middleware technologies. The core object is completely independent of concrete middleware as long as the API is able to handle the middleware. Furthermore, the Application Protocol is completely hidden from the core object.

3.1.2. Meta Schema

The MAMA Meta Schema is a concrete object model. The Meta Schema – as shown in Figure 3-3 – starts with the element *specification*. A MAMA specification includes every statement that belongs to the specified application:

- A **specification** contains all statements that belong to the same schema. A specification includes zero or an infinite positive number of declarations for qualifiers and definitions. It can be distributed over multiple source files.
- A **definition** can be a module, an object, or a user-defined type, called type definition.
- **Qualifiers** need to be defined (*qualifierdef*) before any other statement within the specification. They are used within any declaration. Specified qualifiers can be assigned to the elements module, object, interface, attribute, action, and parameter.
- The element **module** aggregates any infinite non-negative number of definitions that is type definitions, objects, or (sub-) modules.
- An **object** has zero or more interfaces and zero or more type definitions. Objects can inherit characteristics from other objects. Multiple inheritance is not supported. Inherited characteristics from other objects cannot be changed nor specialized.
- An **interface** embraces zero or more actions, attributes, and type definitions. Interface inheritance is not supported.
- An **attribute** can be of a certain type, either basic or user-defined. Each attribute is additionally handled as a new type definition within the specification.
- An **action** may contain any number of parameters.
- **Parameters** are a part of an action. A parameter is treated like an attribute. The separation of parameter and attribute has been introduced to allow the definition of specific qualifiers for stand-alone attributes and action-dependent parameters.
- A **typedef** can be used to introduce new data types in a specification. New type definitions can be based on basic types or any other already defined type. A **member** represents an element of a constructed type.

The Meta Schema defines five core elements: *module*, *object*, *interface*, *attribute*, *action*, and *parameter*. The two additional elements are *type definitions* and *qualifiers*. Type definitions serve as a mechanism to extend the basic data types by adding new types to a specification. Qualifiers are related to the generation of meta information about the described application.

A module groups zero or more objects. It may also contain any number of other modules. An object represents an object following the definition of a computational object of the Telecommunication Information Networking Architecture (TINA; [TINA-CMC]). Objects include any number of interfaces. The interface of an object collects a number of operations, which are called actions in the Meta Schema, and any number of attributes. An action may contain any number of parameters defined for this action only or reused from formerly defined attributes.

A type definition represents an identifiable entity with an associated predicate. “An entity satisfies a type if the predicate is true for that entity. An entity that satisfies a type is called a ‘member of the type’” [CORBA]. The Meta Schema defines a set of basic types and provides a mechanism to define new types based on those basic types, structures, and attributes.

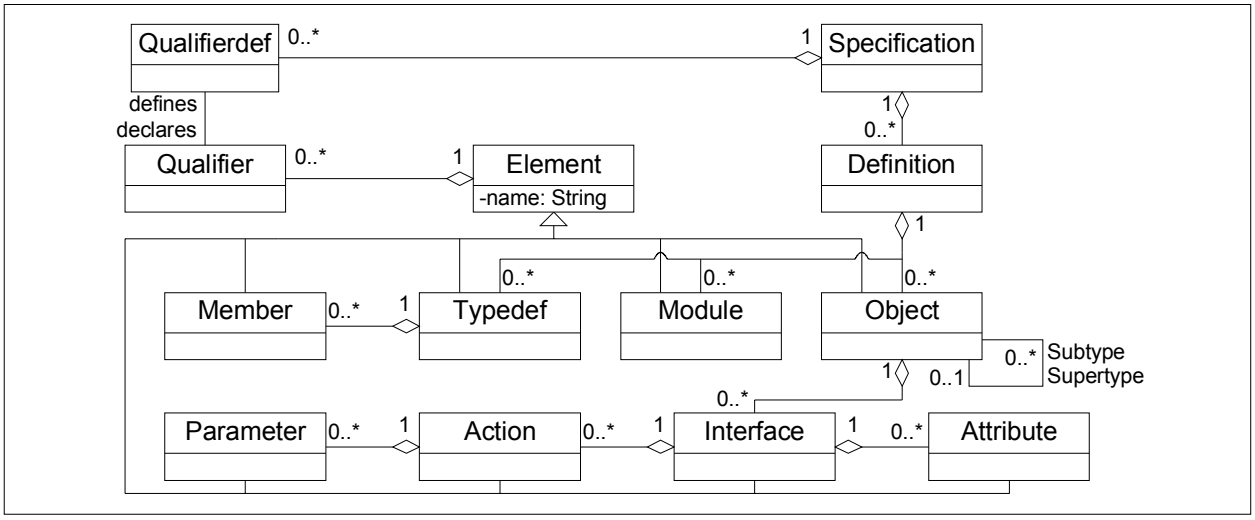

Figure 3-3: Object Model – Meta Schema

The element qualifier is adopted from [DMTF-CIM]. A qualifier is used to characterize the instantiation of core elements within specifications. The introduction of this element provides a mechanism that makes the object model extensible in a limited and fashionable way by applying an appropriate set of meta information. Each environment that employs the Meta Schema can define its own set of qualifiers. The Meta Schema itself defines no specific qualifier at all.

3.2. Application Definition Language

The Application Definition Language follows the specifications of the MAMA meta schema. Its ability to function as a language for data exchange is based on a specified mapping of ADL to the eXtensible Markup Language (XML). This mapping can be done in both ways. An XML document valid against the ADL Document Type Definition (DTD) can be transformed to an ADL specification and vice versa. ADL offers a solution for common tasks for the specification of a manageable distributed system:

- the definition of distributed applications and objects;
- the definition of interfaces of objects that client objects call and object implementations provide;
- the generation of object- and interface repositories;
- the definition of communication protocols for objects and applications on top of middleware;
- the generation of data exchanged by objects;
- the definition of events sent by objects;

- the definition of derived types;
- the definition of managed objects that describe managed resources;
- the definition of Management Information Bases; and
- the definition of information beyond the scope of this document.

ADL is independent of actual business models, communication protocols, programming languages, and languages that are used to describe information exchanged among applications. Protocol mappings, programming language mappings, and data exchange language mappings can be specified for specific environments in an individual way. Mappings from ADL to programming languages or markup languages depend on the facilities of the target languages. This document defines language mappings for the Interface Definition Language (IDL) as defined by the Object Management Group (OMG), specifies an XML notation for ADL, and provides instructions for the development of other mappings.

3.2.1.1. Lexical Conventions

An ADL specification contains five different lexical elements: keywords, identifiers, operators, literals, and other separators. The following rules apply to any ADL specifications:

- an ADL file contains a sequence of ADL statements and/or preprocessor directives;
- every ADL statement starts with a keyword followed by a number of arguments;
- arguments that are allowed to follow a keyword depend on the given keyword;
- every statement is terminated by a semicolon;
- white spaces are ignored unless they are used to separate tokens; and
- comments are ignored and will not be translated within a language mapping.

3.2.1.2. Comments

```
SL_COMMENT              = "//" (~'\n')* '\n';
ML_COMMENT              = "/*"
                          (STRING_LITERAL
                          | CHAR_LITERAL | '\n' | '*' ~'/' |~'*')*
                          "*/";
```

Comments are part of lines, lines, or complete paragraphs that must not be interpreted as part of the specification. Two different types of comments can be used within ADL:

- A *traditional comment*, similar to C++, is used to mark the start and the end of a comment. The start is marked with ‘/*’ and the end is marked with ‘*/’. All statements between start point and end point of the comment are ignored. Traditional comments cannot be nested.
- An *end-of-line comment* is indicated by ‘//’. From this sign, every other character up to the end of the current line is ignored.

Both types of comments are defined in [ISO14882]. Additionally, the *documentation comment* might be used by any ADL specification. This comment is similar to the traditional comment, except that the start sign includes an additional asterisk character (‘/**’). Characters between those two signs can be processed by specialized tools to prepare automatically generated documentation of a specification.

3.2.1.3. Identifiers

```
IDENT                   = ('a'..'z'|'A'..'Z'|'_')
                          ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Identifiers are sequences of characters from the American Standard Code for Information Interchange (ASCII), as listed in the left row of Table B-1, digits from Table B-4, and the underscore character as listed in Table B-2. The first character must be an ASCII alphanumeric character. All characters are significant. Identifiers are treated as case sensitive. That is, two identifiers that differ only in case do not

collide. Once defined, an identifier cannot be overwritten by a proceeding definition within the same name scope.

3.2.1.4. Keywords

The identifiers listed in Table 3-1 are reserved keywords. They cannot be used otherwise. Keywords must be written exactly as shown in the table.

action	alterable	attribute	boolean	char	descr	double	FALSE
float	interface	long	mandatory	module	NULL	object	octet
optional	parameter	qualifier	required	scope	short	string	signed
struct	TRUE	type	typedef	unsigned	void		

Table 3-1: ADL – Keywords

3.2.1.5. White Spaces

```
WS_                     = (' ' | '\t' | '\n' | '\r');
```

White spaces are characters to separate individual tokens. The following ASCII characters are treated as white spaces: space, horizontal tab, form feed, and the line terminators line feed (LF) and carriage return (CR) (cf. Table B-3).

3.2.1.6. Preprocessing

```
PREPROC_DIRECTIVE       = '#' (~'\n')* '\n';
```

For preprocessing, ADL adopts the specifications of [ISO14882] that is the ISO[1] standard for the C++ programming language. Following this approach, the individual preprocessors from any development environment can be used to parse ADL specifications.

3.2.2. Elements

This subsection describes the specifications of the ADL elements. ADL is defined using the Extended Backus-Naur Form (EBNF). A short description of EBNF can be found in Appendix A.3. EBNF is an ISO standard completely described in [ISO14977]. A good overview of this notation with motivation and examples is given in [Scowen93]. The complete ADL grammar is presented in Appendix B.3. Specifications starting with a number belong to the EBNF for a parser, specifications without a starting number belong to the EBNF for a lexicographic analysis.

3.2.2.1. Module

```
20 module               = module_header LCURLY {definition} RCURLY;
21 module_header        = module_literal identifier;
```

The element module provides a simple mechanism to group a configuration of objects for different use cases:

- separation of name spaces within complex specifications;
- geographical or administrative distinction of objects similar to, for example, the X.500 directory tree;
- separation of objects regarding the organizational structure of a company; and

[1] International Standardization Organization

- grouping of objects that offer similar services or solve a specific task in combination.

A module can have zero or more type definitions. A module can include any number of other modules and/or any number of objects.

3.2.2.2. Object

```
22 object            = object_header LCURLY object_body RCURLY;
23 object_header     = object_literal identifier [inheritance_spec];
24 object_body       = {object_export};
25 object_export     = [qualifier_list] (interface SEMI | type_def SEMI);
26 inheritance_spec  = COLON scoped_name;
27 scoped_name       = [[SCOPEOP] identifier {SCOPEOP identifier} SCOPEOP]
                       identifier;
```

An object represents a computational object as described in [TINA-CMC]. The object header contains an optional list of qualifiers, the object literal, and a unique identifier for the object. Furthermore, inheritance definitions are included in the object header. The object body might contain any number of interfaces and/or type declarations. A scoped name must denote a previously defined object.

3.2.2.3. Interface

```
28 interface         = interface_header LCURLY interface_body RCURLY;
29 interface_header  = interface_literal identifier;
30 interface_body    = {interface_export};
31 interface_export  = [qualifier_list]
                       (attribute SEMI | action SEMI | type_def SEMI);
```

An interface is always part of an object. It is constructed by a header and a body. The interface header includes an optional list of qualifiers, the interface literal, and a unique name of the interface (identifier). The interface body can contain any number of attributes, actions, and type declarations.

3.2.2.4. Attribute

```
32 attribute         = attribute_header;
33 attribute_header  = attribute_literal simple_type_spec identifier;
```

An attribute is part of an interface. For each attribute, the ADL compiler will create appropriate operations to read the current value of this attribute (get) and to alter it (set). An attribute satisfies the following definition: It consists of an optional list of qualifiers, a type specification, and a unique identifier. The operations set and get are constructed by using the prefix ‘set_’ and ‘get_’ before the attribute’s identifier.

For the declaration of an attribute, only basic types and scoped names are allowed. That is, any prior declared type (including structures) can be used. An attribute defines a new type itself that can be used in the following specifications.

3.2.2.5. Action

```
34 action            = action_header
                       LPAREN [param_decl {COMMA param_decl}] RPAREN;
35 action_header     = action_type_spec identifier;
36 action_type_spec  = simple_type_spec | void_literal;
```

An action is an operation of an interface, in object-oriented programming languages also known as method. The definition of an action consists of an action header and a list of parameters. The action header may contain a list of qualifiers. The header must contain a type specification for the action. This specification defines the type of the return value of the action that can be of any previously defined type or empty (void).

The list of parameters is either empty (no parameters at all), or contains any positive number of parameters. Some ADL compiler might limit the number of actually allowed parameters of an action. Each compiler should at least recognize the first 32 parameters.

3.2.2.6. Parameter

```
37 param_decl        = [qualifier_list] simple_type_spec identifier;
```

A parameter can contain a list of qualifiers. A parameter must contain a type specification, which can be any previously defined type, a unique identifier, and an optional array declaration.

3.2.2.7. Qualifier

A qualifier supplies additional or meta-information about modules, objects, interfaces, actions, attributes, parameters, and type definitions. Additionally, the members of constructed types are applied with qualifiers. Qualifiers are defined by:

- a *name* that is unique within the specification;
- a *scope* that identifies the ADL elements where this qualifier is applicable;
- a *type* indicating the ADL basic type for the qualifier;
- a *default value* for the initialization;
- a number of *status* attributes associated to each element of its scope to specifying the level of application for of the qualifier for this element (see Table 3-2);
- an attribute *alterable* specifying whether the value of the qualifier can be altered or not; and
- an attribute containing a *description* or a specific scheme of a Uniform Resource Locator (URL) of type *httpurl*, *ftpurl*, or *fileurl* according to and usable as described in [IETF-RFC1738] section 3 as link to the actual description of the qualifier.

The scope of a qualifier is a list of pairs that are constructed by an element and a status. The element specifies the ADL element the qualifier is defined for. The status gives a requirement level for the qualifier following Table 3-2.

Status	Description
Required	The qualifier needs to be defined with a non-zero default value.
Mandatory	The qualifier needs to be defined, with unspecified default value.
Optional	The qualifier might, but has not to be defined.

Table 3-2: ADL – Statuses of Qualifiers for a certain Scope

A system specified in ADL must provide a mechanism to alter the value of qualifiers that are alterable. The common approach is to define an abstract object with an interface that offers such operations. Qualifiers that are used to characterize type definitions are not alterable. Additionally, those qualifiers are not inherited by attributes, actions, and parameters that are based on the type definitions. Qualified type definitions represent a recommendation on how to use the new introduced types in the following specification.

A system may define qualifiers that contain any kind of text. The XML mapping of ADL indicates that such pieces of text should not include statements that can be misinterpreted by XML parsers. Furthermore, if such text is intended to be used in Hypertext Markup Language (HTML) document it should not include any special HTML character such as the ampersand.

The definition of qualifiers has to be done before any other ADL statement. Qualifiers cannot be specified after the first type definition, module, or object occurred. The identifiers of qualifiers are case-sensitive and cannot be used for other ADL statements or as identifier for other ADL elements. The following EBNF code specifies the definition of qualifiers.

```
03 qualifier_def     = qualifier_header qualifier_body SEMI;
04 qualifier_header  = qualifier_literal identifier COLON;
05 qualifier_body    = qualifier_type COMMA qualifier_alt COMMA
                       qualifier_scope COMMA qualifier_descr;
06 qualifier_alt     = alterable_literal LPAREN alterable RPAREN;
07 alterable         = true_literal | false_literal;
08 qualifier_scope   = scope_literal
                       LPAREN scope_rank {COMMA scope_rank} RPAREN;
09 scope_rank        = LBRACK element COMMA rank RBRACK
10 element           = (module_literal    | object_literal
                       |interface_literal | action_literal
                       |attribute_literal | parameter_literal);
11 rank              = required_literal | mandatory_literal
                     | optional_literal;
12 qualifier_descr   = descr_literal LPAREN string_value RPAREN;
13 qualifier_type    = type_literal LPAREN
                       base_type_spec {array_declarator} default_value RPAREN;
14 default_value     = ASSIGN constant_value;
```

Once defined, qualifiers can be used. The scope of qualifiers identifies the ADL elements it can be applied to and the level of requirement called rank. Qualifiers that are ranked as *mandatory* or *required* have to be applied to all ADL definitions of the regarding ADL element. Qualifiers that are not scoped for a specific ADL element cannot be used with this element. The following EBNF code shows how qualifiers are collected and surrounded by square brackets.

```
15 qualifier_list    = LBRACK qualifier {COMMA qualifier} RBRACK;
16 qualifier         = identifier [qualifier_param];
17 qualifier_param   = LPAREN constant_value {COMMA constant_value} RPAREN;
```

3.2.3. Types and Values

ADL offers a number of basic types and a mechanism to introduce new, environment specific types. Basic types are related to values that are permitted for those types.

3.2.3.1. Basic Types

Similar to programming or interface languages, ADL offers a number of basic types that build the foundation of all data types available for a specification.

```
40 simple_type_spec  = (scoped_name | base_type_spec) {array_declarator};
41 base_type_spec    = (numeric_type | char_type | string_type
                       |boolean_type | octet_type);
43 numeric_type      = integer_type | floating_pt_type;
```

Integers can be short, long, or extra long with a signed or unsigned characteristic.

```
45 integer_type      = [unsigned_literal | signed_literal]
                       (short_literal | (long_literal [long_literal]));
```

Floating numbers are floats, double precise, and long double precise. The corresponding types are *float*, *double*, and *long double*. They follow the Institute of Electric and Electronic Engineers (IEEE) standard for single-precision, double-precision, and double-extended floating point numbers as described in the American National Standards Institute (ANSI) standard 754-1985. C++ and OMG IDL follow the same standard. Adopting this for floating points is a necessity for acceptance and interoperability of ADL.

```
44 floating_pt_type  = float_literal | [long_literal] double_literal;
```

Characters can be expressed by the type *char*, which is an 8-bit quantity that encodes a single byte character from any byte oriented code set or, when used in an array, a multi-byte character from a multi-byte code set. The ISO 8859-1 (Latin1) character set standard defines meaning and representation of all possible graphic characters used in ADL. These are space and alphabetic characters given by Table B-1, the graphic characters of Table B-2, the digits of Table B-4, and the escape characters of Table B-5. Additionally, the meaning of null and formatting characters (cf. Table B-3) is the numerical value of the character as defined by ASCII (ISO646) standard. All other characters and their meaning is implementation dependent.

```
46 char_type         = char_literal;
```

The data type *boolean* is used for elements that can only take the value *true* or *false*. No other values are permissible for a Boolean-typed element.

```
48 boolean_type      = boolean_literal;
```

The type *string* can consist of all possible 8-bit quantities in form of a sequence of characters

```
47 string_type       = string_literal;
```

The data type *octet* is an “8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.” [CORBA]

```
49 octet_type        = octet_literal;
```

3.2.3.2. Type Definitions

```
18 type_def          = (typedef_literal type_spec | struct_type_spec);
19 type_spec         = simple_type_spec identifier;
38 struct_type_spec  = struct_literal identifier LCURLY {member}- RCURLY;
39 member            = [qualifier_list] type_spec SEMI;
```

The language construct *typedef* is included to allow the definition of new environment specific types. A type definition can be simple or complex. Simple type definitions are variations of existing basic types or already defined new types. Complex type definitions are structures. This mechanism is the same as in C++ and OMG IDL. Qualifiers that are defined for attributes apply also for type definitions. Here, type definitions and attributes are handled equally.

The definition of a simple type is simple. The *typedef* literal is followed by a simple type (basic type or predefined other simple type) and a unique identifier. Complex type definitions are identified by the literal *struct*, which is followed by a unique identifier and the members of this structure surrounded by curly brackets. *Members* can be basic types and earlier simple or complex type definitions. The definition of new structures within structures is not supported.

3.2.3.3. Values

```
80 constant_value    = integer_value | char_value   | string_value
                     | boolean_value | binary_value | float_value;
```

Values are expressed with the EBNF token *constant_value* for integer, character, string, boolean, binary, and float types. Additional definitions, also given in EBNF, provide a formal specification that can be used for the assembly of compilers or interpreters for ADL.

Binary values are used for bit fields and for Boolean data types. Only the values *true* and *false* are permitted. Both values are placeholders for the two binary values *0* and *1*. A binary value other than a Boolean is interpreted as an integer. To distinguish a base 10 integer from a base 2 binary value, the binary value must be preceded by the character *b* or *B*.

```
81 binary_value      = BINARY;
BINARY               = ("b" | "B") (BINARYDIGIT)+;
BINARYDIGIT          = ('0' | '1');
```

An integer value can be either of base 10 (decimal), base 8 (octal value), or base 16 (hexadecimal value). A decimal value is just the number, octal values are indicated by an *o*, and a hexadecimal value can be identified by the preceding *0x* or *0X*.

```
82 integer_value     = INT | OCTAL | HEX;
DIGIT                = '0'..'9';
OCTDIGIT             = '0'..'7';
HEXDIGIT             = ('0'..'9' | 'a'..'f' | 'A'..'F');
HEX                  = ("0x" | "0X") (HEXDIGIT)+;
INT                  = (DIGIT)+                  // base-10
                       ['.' (DIGIT)*
                       [('e' | 'E') ['+' | '-'] (DIGIT)+]
                       |('e' | 'E') ['+' | '-'] (DIGIT)+
                       ];
```

Integer values express numbers regarding the following rules for ranges [CORBA]:

Integer type	Range of values
short	$2^{15} \dots 2^{15} - 1$
long	$2^{31} \dots 2^{31} - 1$
long long	$2^{63} \dots 2^{63} - 1$
unsigned short	$0 \dots 2^{16} - 1$
unsigned long	$0 \dots 2^{32} - 1$
unsigned long long	$0 \dots 2^{64} - 1$

Table 3-3: ADL – Integer Types and their Value Range

String values are defined as a *constant_value* that is surrounded by double quote characters. For better readability of specifications, ADL allows the separation of one string into several parts where each part is surrounded by double quote characters and a white space is used to concatenate those parts.

```
85 string_value      = strings {strings} | null_literal;
86 strings           = STRING_LITERAL;
STRING_LITERAL       = '"' (ESC | ~'"')* '"';
```

The other values follow the rules of the basic data types they are defined for.

```
83 float_value       = FLOAT;
84 char_value        = CHAR_LITERAL;
87 boolean_value     = true_literal | false_literal;
FLOAT                = '.' (DIGIT)+ [('e' | 'E') ['+' | '-'] (DIGIT)+];
CHAR_LITERAL         = '\'' ( ESC | ~'\'' ) '\'';
```

3.2.3.4. Arrays

```
42 array_declarator  = LBRACK RBRACK;
```

Every basic type and every new type definition can be declared as an array by applying square brackets to the type. An array cannot comprise more than one type. The length of an array is not to be specified. An API is responsible for handling array boundaries. Additionally, an ADL specification can restrict the us-

age of arrays with appropriate qualifiers that indicate minimum and/or maxim members of the array. Multiple-dimensional arrays are permitted.

3.2.4. Scopes and Naming

ADL employs named and unnamed scopes. A named scope is defined by the elements module, object, and interface. An unnamed scope is defined by the element action and by the constructed type struct (structure). A qualified (scoped) name can be used to resolve names. A scoped name consists of names separated by two colon characters '::'. Unnamed scopes cannot be resolved. They are used to enable identical identifiers for action parameters and structure members.

A scope starts immediately after the opening ('{') and terminates immediately after the closing ('}') bracket of the respective ADL element. Identifiers can only be defined once within a scope. The identifiers of the elements module, object, interface, action, and struct can be redefined in the immediate scope. The following ADL example shows a number of declarations for type definitions, attributes, and actions with the use of qualified names (qualifiers are not shown):

```
01  module m1{
02    typedef unsigned short t1;
03    typedef t1 t2;
04    typedef string t3;
05
06    object o1{
07      typedef short t1;
08      typedef t1 t2;
09      typedef string t3;
10
11      interface i1{
12        typedef short t1;
13        typedef t1 t2;
14        typedef string t3;
15        attribute t3 a3;
16        attribute m1::o1::t3 a2;
17
18        long act1(long p1, unsigned short p2);
19        m1::t3 act2(string p1, long p2, short p3);
20      };
21    };
22  };
```

Line four shows a type definition that binds the identifier *t3* to the type string. The same identifier is used in the object *o1* and in the interface *i1*, also bound to the type string. In *i1*, all three type definitions are used (line 15, 16, and 19). The first occurrence references the type definition done in the interface *i1* (line 14), the second the one done in the object *o1* (line 9), and the last the one done *m1* (line 3).

3.2.5. xADL – XML for ADL Data Exchange

The eXchange ADL (xADL) format is an exact representation of ADL in the XML language. It has been developed to enable the exchange of data between MAMA applications. The conversion between ADL and xADL and vice versa can be done without information loss. Table 3-4 shows the elements of xADL with their attributes contained elements.

xADL definitions are collected by the element *collection*. This allows grouping any kind of xADL definition into single XML documents. Information that is included in a collection cannot be converted to ADL, since it follows not the rules of the EBNF specification of ADL. The element collection has been added to allow the exchange of information for application services.

xADL Element	Attributes	Elements
collection	-	specification, qualifierdef, module, object, interface, attribute, action, parameter, typedef, member, qualifier, scope, description, constant_value
specification	name, distinguished_name, uuid	qualifierdef, module, object, typedef
qualifierdef	name, distinguished_name, base_type, signed, default_value, alterable, array_dim	scope, description
module	name, distinguished_name	qualifier, module, object, typedef
object	name, distinguished_name, extends	qualifier, typedef, interface
interface	name, distinguished_name	qualifier, typedef, attribute, action
attribute	name, distinguished_name, type, base_type, array_dim, signed	qualifier
action	name, distinguished_name, type, base_type, array_dim, signed	qualifier, parameter
parameter	name, distinguished_name, type, base_type, array_dim, signed	qualifier
typedef	name, distinguished_name, type, base_type, array_dim, signed	qualifier, member
qualifier	name, distinguished_name	constant_value
constant_value	-	-
member	name, distinguished_name, type, base_type, array_dim, signed	-
scope	element, rank	-
element	-	-
rank	-	-
description	-	-

Table 3-4: xADL – Elements and Attributes

The name of the master ADL file defines the name of the specification. Some elements have an additional attribute distinguished_name. This attribute has been added to support naming and directory services. This attribute need not to be converted to ADL and can be ignored by a parser or interpreter. All other definitions of xADL follow the specifications of ADL. The complete XML DTD for xADL and the EBNF are presented in Appendix B.4.

3.2.6. Development Process

The process of developing applications with ADL is depicted by Figure 3-4. The system designer creates an ADL specification with any tool that is able to save this in form of a plain text. Usually, this tool would be a more or less featured text editor. The specification itself can be split-up over multiple files. In this case, the preprocessor directive *#include* must be used to concatenate the respective files virtually. This mechanism has to be employed because ADL requires complete specifications that start with qualifier declarations prior to any other definition, followed by type definitions, modules, and/or objects.

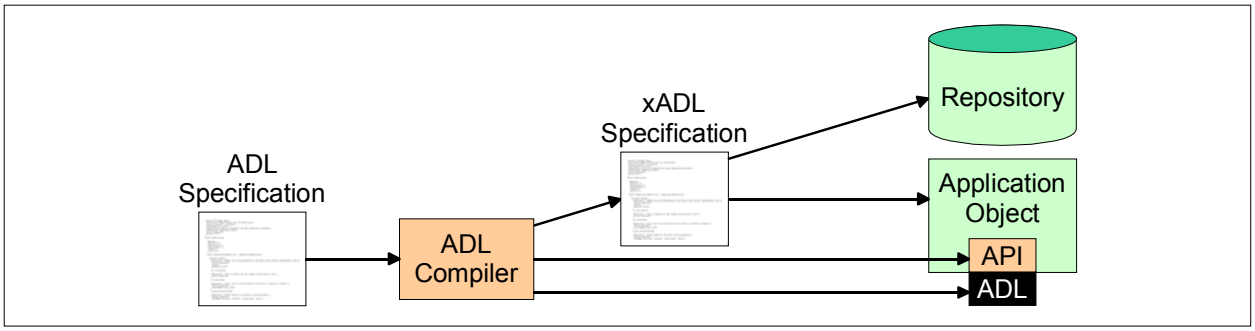

Figure 3-4: ADL – Development Process

The specification is handed over to an ADL compiler. This compiler is responsible to analyze the specification. Any declaration that does not follow the rules defined by ADL should be clearly identified and marked as an error. In case errors are detected, no further action should be undertaken by the compiler. When the specification is error-free, the compiler has the option to generate several transformations. First, it must be able to transform the ADL specification into xADL. Second, the compiler can produce programming language specific code that represents an API for the system programmer. Furthermore, the compiler can support the ADL-typed interface of an application object by extracting this very description for each object in the specification.

The actual offered transformations and output formats depend on the compiler. The minimum functionality of a compiler should comprise the ability to transform ADL to xADL and vice versa, to examine a given specification in order to detect any kind of error (including reuse of identifiers, multiple declarations, exact use of naming conventions, etc.), and to generate statistical information about a given specification that can be used in repositories. The compiler itself can be integrated into development environments that offer additional tools for debugging, tracing, and other features that support an effective design and implementation of MAMA applications. The characteristics of such a Software Development Kit (SDK) are not part of the ADL language specification.

3.3. MAMA Core Model

The MAMA Core Model is an information model that captures issues that are applicable to all MAMA applications. It is a small set of type definitions, classes, and other specifications that, in combination, provide the basic vocabulary for analyzing and describing MAMA applications and services. The Core Model is a specialization of the ADL schema. While the Core Model might be enhanced, major changes to the specifications presented in this section are not anticipated. The Core Model follows four objectives:

1. to identify a reasonable set of qualifiers to extend the specification of ADL elements with meta information for repositories;
2. to specify basic type definitions for the MAMA protocol, the MAMA API, MAMA services, and applications;
3. to declare abstract base classes that can be inherited from by any MAMA application object; and
4. to define basic rules that apply to all MAMA specifications, such as the specification of events and exceptions, recommendations for naming conventions, and possible enhancements of specifications for the Core Model itself.

Figure 3-5 shows the individual specifications of the MAMA Core Model. The Core Model defines a set of qualifiers, which are scoped to the eight ADL elements. Regarding ADL, qualifiers can be ranked as required, mandatory, and optional.

The Core Model assigns seven mandatory and only two optional qualifiers to the ADL element module. Therefore, a module has an almost fixed set of meta information that must be specified for each module in a MAMA specification. Objects and interfaces have one mandatory qualifier less, six, but eight optional qualifiers. Here, the flexibility is enhanced reflecting the different possible target environments and their

specific requirements on object and interface definitions. For actions, the Core Model reserves five mandatory and five optional qualifiers.

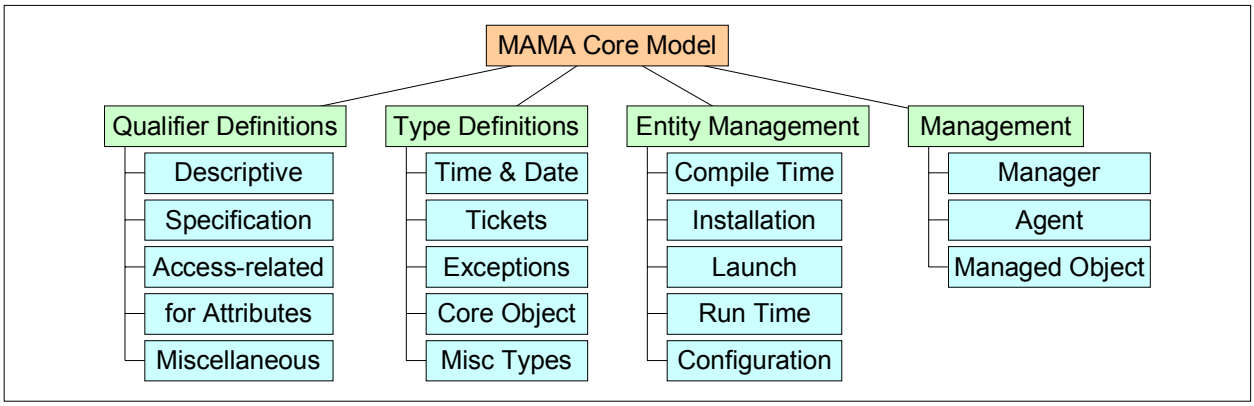

Figure 3-5: MAMA – Core Model

Attributes and type definitions are handled equally regarding qualifier specifications. Both elements require only three, but can be accompanied with up to twenty-three qualifiers. The last element, parameter, requires no specific qualifier. The total number of fifteen qualifiers has been identified applicable to the element parameter. Since not all qualifiers are applicable simultaneously, the Core Model describes the dependencies among them with regulations and recommendations for their use.

The qualifier definitions for ADL elements module, object, and interface are more restrictive than for the other five ADL elements. Repositories build out of the meta information provide general agreed and fixed information on the first type of ADL elements since their role for distributed applications is almost fixed. The second type of elements is more flexible to enable expandable application-specific and domain-specific specifications.

Furthermore, the Core Model identifies basic types that are useful for MAMA applications. Those types comprise a unified specification for time and data information, tickets, and exceptions. Additionally, an abstract core object is included. This object can be used as base class for MAMA applications. The type definitions are concluded by a number of miscellaneous definitions.

The Core Model acknowledges definitions for management. For this reason, some general specifications for management of installation and configuration of MAMA applications are included (Entity Management). Additionally, generic objects for the management roles Manager, Agent, and Managed Object are made to simplify the specification of management hierarchies.

3.3.1. Naming Conventions

To simplify the application of the MAMA Core Model, declarations follow a naming convention.

- The identifiers of modules start with the character *m*.
- The identifiers of objects starting with the character *o*.
- The identifiers of interfaces starting with the character *i*.
- The identifiers of attributes starting with the character *a*.
- The identifiers of type definitions starting with the character *t*.
- The identifiers of structures starting with the character *s*.

No special recommendations are given for the ADL elements *action* and *parameter*. Both elements declare identifiers that are only applicable in the naming scope of a single interface and cannot reused later in a specification file.

3.3.2. Qualifiers

The Core Model defines a set of qualifiers as basis for all specifications. They must be declared as described in this document. It is not valid to change any of this information except the default value that is the assumed initial value. The complete matrix of qualifiers is listed in Table C-1 in Appendix C.1.1.

Each qualifier is assigned with a scope that identifies the ADL elements where this very qualifier is applicable to. Required and mandatory qualifiers must be given to any declaration of a specific ADL element, optional qualifiers might be. Some optional qualifiers are further constrained because they are mutually exclusive or the use of one qualifier implies restrictions on the value of another qualifier. Those dependencies among qualifiers are explained at the end of this section.

Qualifiers are not inherited or passed from one declaration to another. A set of qualifiers assigned to a type definition is not valid for attributes and parameters using this new type. Qualifiers assigned to objects are not inherited by a specialized object.

The MAMA Core Model includes a generic object that is automatically inherited by all new object declarations. This generic object includes operations to alter qualifiers when qualifiers are declared *alterable*. A MAMA application might decide that qualifiers cannot be altered even if specified so. This functionality is provided by the MAMA API.

The next subsections describe the legal MAMA qualifiers. The qualifiers are grouped using the following classification.

- Descriptive qualifiers add description in a natural language: *Behavior, Contact, Description, History, Organization,* and *Usage.*
- Specification qualifiers clarify the current status of a specification: *Revision, SpecStatus, Status,* and *Version.*
- Access-related qualifiers explain access policies to ADL elements: *Permissions, Group,* and *Owner.*
- Qualifiers for attributes and parameters comprise meta information for the generation of repositories: *ArrayType, Counter, BitMap, BitValues, DisplayHint, DisplayName, MaxLen, MaxValue, MinLen, MinValue, StepIndex, ValueMap, Values, Wildcards,* and *Units.*
- Miscellaneous qualifiers belong not to another classification: *Abstract, In, Out, Quality, RegisteredAs,* and *xmlDTD.*

Each qualifier is accompanied with its ADL specification. The general form of this specification is:

```
qualifier Q1: type(string = NULL), alterable(TRUE|FALSE),
              scope([element1, status], [element2, status]),
              description("text");
```

Furthermore, the particularities of the qualifiers are described, the possible ways of using the qualifiers are shown, and relationships to standards are presented.

3.3.2.1. Descriptive Qualifiers

Descriptive qualifiers include information that is usually contained in manuals only. Such information is useful for an application programmer to understand the meaning of a specification. On application runtime, these qualifiers offer information that can be employed to generate on-line help systems for human operators and users.

The qualifiers are intended to comprise descriptions in a natural language of choice, a formal language specific to an application, or links to such descriptions. Three formats are recognized and will be automatically processed by a MAMA environment:

- Plaintext is text that can be built out of any character listed in Table B-1 (alphabetic characters), Table B-2 (graphic characters), Table B-3 (formatting characters), Table B-4 (numeric characters), and Table B-5 (escape characters). This text might be either human readable plaintext or any kind of

formal language for automated processing. White spaces shall be ignored displaying or processing this type of text.

- HTML is formatted text according to [W3C-HTML] that serves for an easy integration in web enabled applications.
- A URL as link to either plaintext or HTML formatted text can be used instead of the actual description to minimize the overhead that is produced by the descriptions. A URL must follow a specific scheme of type *httpurl*, *ftpurl*, or *fileurl* according to and usable as described in [IETF-RFC1738].

Behavior

```
qualifier Behavior: type(string = NULL), alterable(FALSE),
    scope([object, mandatory], [interface, mandatory],
          [action, mandatory], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/behavior.html");
```

This qualifier explains the behavior of the ADL elements *object*, *interface*, and *action*. For an object, the behavior describes the change of a stable internal condition. An operation (in ADL an action) represents a behavior. When this qualifier is used for an *object* or an *interface*, it describes the collection of behaviors for the ADL element. When this qualifier is used for an *action*, it describes this very action. In general, the change of a state can occur on the change of values on external or internal stimuli. Each attribute of an interface can be accompanied with a description of its behavior with this qualifier.

Contact

```
qualifier Contact: type(string = NULL), alterable(FALSE),
    scope([module, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/contact.html");
```

For each module, a contact has to be specified that names a person that is responsible for the specification of this module. It should contain at least one name, affiliation (if not included in or different to organization), and optional postal address, telephone and fax number, WWW address, and email address.

Description

```
qualifier Description: type(string = NULL), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory], [action, optional],
          [attribute, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/description.html");
```

It is an obligation that each ADL element (except *parameter*) is connected to a textual description. This description is used for the visualization of applications. This can be an online help system, manual, or documentation. The description of an element should be done in a natural language of choice. The description might include information similar to the qualifiers *Behavior*, *Contact*, *Organization*, and *Usage*.

History

```
qualifier History: type(string = NULL), alterable(FALSE),
    scope([module, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/history.html");
```

This qualifier explains the history of the module. Any progress of the module should be stated here by date, person, and actual change. Entries should be sorted by time in a descendant order. The actual string can be automatically generated by code revision software, such the Concurrent Versions System (CVS) or the Revision Control System (RCS).

Organization

```
qualifier Organization: type(string = NULL), alterable(TRUE),
    scope([module, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/organization.html");
```

The organization qualifier contributes information on the organization that is responsible for the module. It also might contain copyright information.

Usage

```
qualifier Usage: type(string = NULL), alterable(FALSE),
    scope([object, mandatory], [interface, mandatory],
          [action, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/usage.html");
```

The content of the qualifier *Usage* should provide programmers with information on how to use an object, interface, or attribute (i.e. description of values for parameters of actions, the intention of actions inside of an interface, and so forth). When appropriate, this comprises also references to other qualifiers. The content of this qualifier can be used for example to assemble documentation about a specification.

3.3.2.2. Specification Qualifiers

Revision

```
qualifier Revision: type(short = 0), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/revision.html");
```

This qualifier describes the minor number of the versions of a module and optional of objects and interfaces. Permitted values are zero or any infinite positive number within the range of an integer type.

SpecStatus

```
qualifier SpecStatus: type(string = "current"), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory], [action, mandatory],
          [attribute, mandatory], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/specstatus.html");
```

It is an obligation that a status is defined for each ADL element (except parameter). This qualifier is similar to the status clause of the Structure of Management Information (SMI), “which must be present, indicates whether this definition is current or historic.” [IETF-RFC2578] The default value is current.

Value	Description
current	“The specification is current and valid.” [IETF-RFC2578]
obsolete	“The specification is obsolete and should not be implemented and/or can be removed if previously implemented.” [IETF-RFC2578]
deprecated	“It permits new/continued implementation in order to foster interoperability with older/existing implementations.” [IETF-RFC2578]

Table 3-5: MAMA Core Model – Values for the Status Qualifier

Status

```
qualifier Status: type(string = "optional"), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory], [action, mandatory],
          [attribute, mandatory], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/status.html");
```

It is an obligation that a status is defined for each ADL element. This qualifier indicates whether the specific element needs to be implemented or not. Permitted values are *required*, *mandatory*, and *optional*. The default value is *optional*.

Value	Description
required	The element needs to be implemented with a meaningful value.
mandatory	The element needs to be present but might have a non-meaningful value.
optional	The element might be implemented or instantiated.

Table 3-6: MAMA Core Model – Values for the Status Qualifier

Version

```
qualifier Version: type(short = 0), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/version.html");
```

This qualifier describes the major number of the versions of a module and optional of objects and interfaces. Permitted values are zero or any infinite positive number within the range of a short type.

3.3.2.3. Access-related Qualifiers

Permissions

```
qualifier Permissions: type(octet = 0755), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/permissions.html");
```

Permissions are access policies for objects, interfaces, attributes, and actions. The permissible values for this qualifier are adopted from UNIX operating systems as described in [SunOS-chmod]. For simplification, MAMA employs read, write, and execute permissions only. Advanced permissions like mandatory locking, set-identifier, and sticky bits are ignored. The given permissions can be pre-defined with this qualifier. However, this qualifier is alterable to enable changes of permissions at run-time.

Permission consists of attributes for read, write, and execute. Permissions can be defined for the owner of an ADL element; a group the element is associated to, and for all other clients. Following this, permissions can be expressed by three sequences each having three characters:

```
User    Group    Other
rwx     rwx      rwx
```

The character *r* marks read permission; *w* is used for write permission, and *x* for execution permission. When the character is missing (indicated by '-'), the respective permission is not granted. For automated processing, the permissions are coded in form of a three digit octet where the first digit represents permissions for the owner, the second for the group, and the last for all others. Each digit can have one of the following meanings:

- 0 – no access permissions at all;
- 1 – execute permission;
- 2 – write permission;
- 3 – execute and write permissions;
- 4 – read permission;
- 5 – read and execute permissions;
- 6 – read and write permissions; and
- 7 – read, write, and execute permissions.

Only the owner of an ADL element should be able to change the actual permissions of this very element. Therefore, this qualifier should be used at least in combination with the qualifier *Owner* in order to identify privileged objects.

Group

```
qualifier Group: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/group.html");
```

This qualifier defines the group of an object, interface, attribute, or action in form of a string. The actual interpretation of the string is out of scope of MAMA core definitions. However, MAMA recommends using this qualifier according to the group management of UNIX operating systems.

Owner

```
qualifier Owner: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/owner.html");
```

This qualifier defines the owner of an object, interface, attribute, or action in form of a string. The actual interpretation of the string is out of scope of MAMA core definitions. However, it is recommended to use this qualifier according to the user management of UNIX operating systems.

3.3.2.4. Qualifiers for Attribute and Parameter

ArrayType

```
qualifier ArrayType: type(string = "bag"), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/arraytype.html");
```

This qualifier is derived from [DMTF-CIM]. It is used for array-typed attributes to qualify the type of the array. Permitted values are *bag*, *ordered*, and *indexed*. The default value is *bag*.

Values	Description
Bag	Unordered and multi-valued array, allowing duplicate entries
Ordered	Specialized bag with the same characteristic, values are returned in an implementation dependant way with a fixed order
Indexed	Maintains the order of elements e.g. with an integer index for each value of the array

Table 3-7: MAMA Core Model – Values for the ArrayType Qualifier

BitMap

```
qualifier BitMap: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/bitmap.html");
```

This qualifier is derived from [DMTF-CIM]: "Indicates which bit positions are significant in a bit map. The position of a specific value in the *BitMap* array defines an index that is used in selecting a string literal from the *BitValues* array."

```
qualifier BitValues: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/bitvalues.html");
```

BitValues

This qualifier is derived from [DMTF-CIM]: It "… provides translation between a bit position value and an associated string. See the description for the *BitMap* qualifier."

Counter

```
qualifier Counter: type(boolean = FALSE), alterable(FALSE),
    scope([attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/counter.html");
```

This qualifier indicates that the given attribute is used as a counter.

DisplayHint

```
qualifier DisplayHint: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/displayhint.html");
```

This qualifier, when used for attributes, gives a hint for graphical user interfaces on how to display the value of the given attribute. This usages adopts the definitions of the Simple Network Management Protocol (SNMP) found in [IETF-RFC2579] on page 21 and 22. When this qualifier is used for objects or interfaces, it should contain a reference in form of an URL to code that can be used for displaying the capabilities of the relative element. This code should be a graphical user interface by nature, which is HTML, Java, or native code for any operating system platform.

DisplayName

```
qualifier DisplayName: type(string = NULL), alterable(TRUE),
    scope([attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/displayname.html");
```

This qualifier gives a name different from the specified name of the attribute that can be used in a graphical user interface.

MaxLen

```
qualifier MaxLen: type(long = 1024), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/maxlen.html");
```

This qualifier defines the maximum length of an attribute that is typed as a *string* or an *array* with an infinite positive number. This qualifier is not applicable for all other basic types. When used for an attribute of type string, this qualifier indicates the maximal allowed length of the string in single byte characters. When used for an attribute of type array, this qualifier indicates the maximal allowed members of the

array. Both numbers might be related to memory issue. However, strings and arrays can extend the maximum length, but overhead information will not be processed.

MaxValue

```
qualifier MaxValue: type(long = 1024), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/maxvalue.html");
```

This qualifier defines the maximum value for attributes typed as integers or real numbers according to the ADL EBNF. This qualifier is not applicable for all other basic types.

MinLen

```
qualifier MinLen: type(long = 0), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/minlen.html");
```

This qualifier defines the minimum length of an attribute that is typed as a string or an array with an infinite positive number or zero. This qualifier is not applicable for all other basic types. When used for an attribute of type string, this qualifier indicates the minimum allowed length of the string in single byte characters. When used for an attribute of type array, this qualifier indicates the minimum allowed members of the array. Both numbers might be related to memory issue. However, strings and arrays can extend the minimum length, but overhead information will not be processed.

MinValue

```
qualifier MinValue: type(long = 0), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/minvalue.html");
```

This qualifier defines the minimum value for attributes typed as integers or real numbers according to the ADL EBNF. This qualifier is not applicable for all other basic types.

StepIndex

```
qualifier StepIndex: type(long = 1), alterable(FALSE),
    scope([attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/stepindex.html");
```

This qualifier specifies the step index for an attribute of type integer that is used as a counter. It should not be used without the qualifier *Counter*. *StepIndex* is of type signed long, indicating that the value for the associated counter is incremented or decremented. Permitted values are any negative or non-negative number.

Units Qualifier

```
qualifier Units: type(string = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/units.html");
```

This qualifier is derived from [DMTF-CIM]. It provides units in which data of the associated attribute is expressed. Recommended values for this qualifier are listed in Table C-2.

ValueMap

```
qualifier ValueMap: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/valuemap.html");
```

This qualifier is derived from [DMTF-CIM]. It defines a set of permissible values for the associated attribute. A *ValueMap* can be used alone or in combination with the qualifier Values. “When used in combination with the Values qualifier, the location of the value in the *ValueMap* array provides the location of the corresponding entry in the Values array. *ValueMap* may only be used with string and integer values.” [DMTF-CIM]

Values

```
qualifier Values: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/values.html");
```

This qualifier is derived from [DMTF-CIM]. This qualifier provides a map for translations of integer values (enumerations) to associated strings.

Wildcards

```
qualifier Wildcards: type(boolean = FALSE), alterable(FALSE),
    scope([attribute, required], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/wildcards.html");
```

This qualifier indicates that wildcards are allowed for the given attribute. It should be used for string and character typed attributes. The precise definition of the allowed wildcard syntax and semantic is up to the distributed system. MAMA uses extended regular expressions as defined by the Portable Operating System Interface (POSIX; [IEEE-1003.2]).

3.3.2.5. Miscellaneous Qualifiers

Abstract

```
qualifier Abstract: type(boolean = FALSE), alterable(FALSE),
    scope([object, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/abstract.html");
```

This qualifier is derived from [DMTF-CIM]. It indicates that the object is abstract and serves only as base class. An instance of an abstract element cannot be created.

RegisteredAs

```
qualifier RegisteredAs: type(string = NULL), alterable(FALSE),
    scope([module, optional], [object, optional],
          [interface, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/registeredas.html");
```

Indicates the registration name of the given module, object, or interface regarding the employed naming scheme. This qualifier is optional. That implies, that the respective ADL element can be specified without a pre-described name. In this case, the naming service is responsible for selecting an appropriate name and node within the naming tree. An MAMA object might suggest an appropriate name for its instances to the naming service. This qualifier will not be used to derive the name of classes since that is generated directly out of the specification.

In

```
qualifier In: type(boolean = TRUE), alterable(FALSE),
    scope([parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/in.html");
```

The declaration of a parameter of an action must have a directional attribute. This attribute informs the protocol and the API in which direction the parameter has to be passed. The *In* qualifier identifies that a

parameter is passed from client to server (or from calling object to a called object respectively). When it is used in combination with the *Out* qualifier, the parameter will be passed in both directions. This behavior is similar to the parameter attributes defined by [CORBA]. While the status of this qualifier is optional, it is necessary to use at least one qualifier for the direction of a parameter.

Out

```
qualifier Out: type(boolean = FALSE), alterable(FALSE),
    scope([parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/out.html");
```

The declaration of a parameter of an action must have a directional attribute. This attribute informs the protocol and the API in which direction the parameter has to be passed. The *Out* qualifier identifies that a parameter is passed from server to client (or from calling object to a called object respectively). When it is used in combination with the *In* qualifier, the parameter will be passed in both directions. This behavior is similar to the parameter attributes defined by [CORBA]. While the status of this qualifier is optional, it is necessary to use at least one qualifier for the direction of a parameter.

Quality

```
qualifier Quality: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/quality.html");
```

This qualifier provides information for both programmers of objects and users on Quality of Service (QoS) that can be used i.e. for accounting. Regarding objects, this qualifier indicates whether the implementation of this object might become simple or large. From the viewpoint of interfaces and actions, this qualifier describes whether the usage of those elements might result in expensive or cheap accounting. With this mechanism, a MAMA environment can define constraints for object deployment. E.g., large objects will not be useful on settop boxes with limited processing power and simple objects cannot solve monitoring of thousands of employees of a company. Similar, the user (client software) can be informed that the activation of a certain interface is expensive so that he can decide to try other interfaces that provide similar services but need maybe more time.

The actual definition of large, simple, expensive, and cheap has to be done within the concrete MAMA environment. This task is out of scope of the MAMA core specifications. Allowed values are *large*, *small*, *expensive*, and *cheap*. The basic intention for a *large* or *small* object is to give enhanced information on its implementation. *Expensive* and *cheap* reflect the usage of resource and can affect accounting and billing of operation calls.

xmlDTD

```
qualifier xmlDTD: type(string = NULL), alterable(FALSE),
    scope([interface, optional], [attribute, optional],
          [action, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/xmlDTD.html");
```

This qualifier indicates the URL to a DTD that is used to code the XML statements that are used within the interface, attribute, or action. When used for the ADL element attribute, this very attribute has to be of the type string. The content of this qualifier can be plain text or a URL as a link to the actual DTD. A URL must follow a specific scheme of type *httpurl*, *ftpurl*, or *fileurl* according to and usable as described in [IETF-RFC1738].

3.3.2.6. Dependencies among Qualifiers

Several qualifiers have dependencies to other qualifiers. Those dependencies can only be processed by a MAMA specific ADL parser, which has the knowledge on the semantics of qualifier definitions. Therefore, the ADL parser is a tool specific to a certain Core Model, in this case to the MAMA Core Model.

Qualifiers	Dependencies
Counter, StepIndex	*StepIndex* can only be used when *Counter* is present.
Abstract, Status	When *Abstract* is used, the qualifier *Status* defines under which conditions an instance of the object should exist. Required defines, that at least one instance must be present. Mandatory indicates that basic services depend on an existing instance. Optional means that no instance must be present.
In, Out	These two qualifiers can be used alone or together, but at least one of them needs to be present for any parameter.
MaxLen, MaxValue, MinLen, MinValue, StepIndex	Here, *StepIndex* is not used as indication for a counter, but to define the permitted set of values for an attribute or parameter. When *StepIndex* is used, it characterizes the intended values. Otherwise, the ADL element accepts any (analog) value in the given maximum and minimum borders.
Group, Owner, Permission	The qualifier *Permission* should be used in compartment of *User* and *Group* These qualifiers can be used to define an element as read-only. With the permissions set to '0444' and a given owner, only this very owner can change the permissions to another value.
SpecStatus, Status	When *SpecStatus* is set to current, *Status* can take all defined values. Otherwise, the status of a specification should be changed to optional.
ValueMap, Values	When *Values* is used alone, it defines the permitted values of an attribute, typedef, parameter, or structure member. It must contain only values that can be carried by the type the ADL element is assigned to. When both qualifiers are used in combination, *ValueMap* defines the location of the corresponding entry in the array of *Values*.

Table 3-8: MAMA Core Model – Dependencies among Qualifiers

When new qualifiers are added, the parser must be supplied with the semantic information about these new qualifiers. Dependencies should be avoided when possible. Table 3-8 lists the dependencies of qualifiers of the MAMA Core Model.

3.3.3. Type Definitions

This section introduces the basic type definitions of the Core Model. Some specifications are not presented here but in the sections that deal with the MAMA application protocol, the MAMA API, and with the Application Services. The complete set of type definitions is included in Appendix 5.2.2.4.C.1.3. No descriptive qualifiers are given for specifications.

3.3.3.1. Time and Date

Time and date is important information that is not yet standardized among distributed systems. The ISO standard [ISO8601] defines a common format to express certain moments in time. This standard cannot be used for the definition of time durations as used by personal schedules. However, it is applicable for time stamps as they occur whenever an event is to be reported or any kind of action needs to be logged.

Furthermore, the dates specified in [ISO8601] are going to be used as general time and date format for Internet protocols making an adoption of this standard a significant issue for interworking. The standard gives the syntax description in the Augmented Backus-Naur Form (ABNF, cf. [IETF-RFC2234]):

```
date-fullyear  = 4DIGIT
date-month     = 2DIGIT  ; 01-12
date-mday      = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31
time-hour      = 2DIGIT  ; 00-23
time-minute    = 2DIGIT  ; 00-59
```

```
time-second    = 2DIGIT  ; 00-58, 00-59, 00-60
time-secfrac   = "." 1*DIGIT
time-numoffset = ("+" / "-") time-hour ":" time-minute
time-offset    = "Z" / time-numoffset

partial-time   = time-hour ":" time-minute ":" time-second
                 [time-secfrac]
full-date      = date-fullyear "-" date-month "-" date-mday
full-time      = partial-time time-offset

date-time      = full-date "T" full-time
```

The characters *T* and *Z* might also be used in the lower cases *t* and *z*. Although time and date information can be combined, separated by the character *T*, applications might choose to specify full date and full time separated by a space character for the sake of readability. The token *date-mday* can receive the following values:

- 0-28 for the month February in a normal year;
- 0-29 for the month February for a leap year;
- 0-30 for the months April, June, September, and November; and
- 0-31 for the months January, March, May, July, August, October, and December.

Special care has to be taken for months in which a leap second occurs. Here, the *time-second* token is permitted to take the value *60*. It is also possible for a leap second to be subtracted, at which times the maximum value of *time-second* is *58*. In all other cases, the maximum value for *time-second* is *59*. Further, in time zones other than *Z*, the leap second point is shifted by the zone offset in order to reflect the fact that it happens at the same time around the globe.

Leap seconds cannot be predicted far into the future. The International Earth Rotation Service (IERS) publishes bulletins that announce leap seconds with a few weeks warning. Applications should not generate timestamps involving inserted leap seconds before the leap seconds are announced.

The token *time-hour* is allowed to have the value *24* following [ISO8601]. However, this specification does only permit values in the interval *0* and *23* for the actual hour.

The following code shows the definitions for time and date information. The specification defines two complex structures, one for time information and one for date information. Furthermore, three simple strings are specified that provide *stringified* information.

```
struct sTime{
  [MinValue(0), MaxValue(23), StepIndex(1)]
  unsigned short hour;

  [MinValue(0), MaxValue(59), StepIndex(1)]
  unsigned short minute;

  [MinValue(0), MaxValue(60), StepIndex(1)]
  unsigned short second;

  [MinValue(0), MaxValue(9), StepIndex(1)]
  unsigned short secFrac;

  signed short numOffset;
  signed short offset;
  signed short partialTime;
};

struct sDate{
  [MinValue(1), MaxValue(12), StepIndex(1)]
  unsigned short month;
```

```
  [MinValue(1), MaxValue(31), StepIndex(1)]
  unsigned short day;

  [StepIndex(1)]
  signed short fullyear;
};
struct sTime{
 unsigned short hour;
 unsigned short minute;
};
typedef string tTime;
typedef string tDate;
typedef string tTimeDate;
```

The structures *sTime* and *sDate* follow directly the specifications of [ISO8601]. The strings *tTime*, *tDate*, and *tTimeDate* are introduced to simplify the exchange of time information and to allow simple time stamps for notifications and logs. The content of the three strings will only be processed by MAMA components (protocol, API, services) when it strictly follows the time specifications of [ISO8601].

3.3.3.2. Tickets and Exceptions

Tickets are used within the system to notify any interested component about events that have occurred. The following ADL code specifies a generic ticket structure that shall be used for the exchange of notifications. The structure *sTicket* consists of two parts. The first part represents the header. It goes up to the structure member *optionalHeaderFields*. The second part of a ticket – called body – contains additional information.

```
[ValueMap("0", "1", "2", "3", "4", "5", "6"),
 Values("unknown", "Information", "Warning", "Error", "Exception",
        "Accounting", "Notification")]
typedef unsigned short ticketCategory;

struct sTicket {
  ticketCategory category;
  tTimeDate time;
  unsigned long ticketPriority;
  string ticketType;
  string ticketOriginator;
  string ticketDescription;
  tNameValueList optionalHeaderFields;
  tNameValueList filterableBody;
  tNameValueList anyThingElse;
};
```

The first part starts with a category. The category of a ticket can be specified by means of *unknown*, *information*, *warning*, *error*, *exception*, *accounting* and *notification*. The categories should be used as described below.

- *Unknown* should be used when the ticket cannot be categorized with any other value, e.g. for application specific categories included in the optional part of the ticket.
- *Information* should be used to describe low prioritized status changes to all interested application objects. Tickets of this category might require a response of other objects, but this response must not be done immediately.
- A *notification* should be used to notify a group of application objects about important changes that might require notice but not necessarily further action. A notification can be used to realize scenarios as the trap-directed polling of SNMP.
- A *warning* indicates that a certain condition has reached a level where the attention of other application objects is needed to monitor further developments. The monitoring can be realized directly on

the object that has emitted the warning. Otherwise, the event service might be overloaded with warnings during critical phases of a distributed system.

- An *exception* is an event that requires the immediate attention of application objects that receive it. Furthermore, an exception can be used to inform calling objects about an exception that has been recognized during the processing of an action. This mechanism can be used to realize programming language exceptions similar to C++.
- *Accounting* events are used to generate information useful for accounting and performance management, as well as for generating statistically information about the runtime behavior of application objects. The content of those tickets is application specific and should not be processed by the event services.

The time in the ticket structure indicates the time the ticket was originally created and sent by an object, not the time it was received at the event service. The priority is not further processed by MAMA. Applications can define a specific policy for handling priorities. Only the type long is predefined for priorities.

The member *ticketOriginator* should contain the address of the object that has created the ticket. This member is specified as string. Therefore, the address can be a middleware specific object identifier (e.g. CORBA IOR) or an address from the MAMA naming service. The member *ticketDescription* should contain a description of the message following the rules that are given for descriptive qualifiers in section 3.3.2.1. The member *optionalHeaderFields* is included to allow applications the specification of specific, non interpreted parameters for the ticket header.

The body of a ticket will not be processed by MAMA. This part comprises application specific information. This information can be separated into a filterable part (*filterableBody*) and a non-filterable part (*anyThingElse*). This approach follows the definition of a structured event in the CORBA notification service [CORBA-NotS].

3.3.3.3. Specifications for MAMA Core Objects

Each MAMA object must provide functionality to alter qualifiers and to send specification information on request. The Core Model declares an abstract object *oMamaCore*, which serves for those two use cases. The functionality of this object is automatically inherited by all MAMA objects. The object class *oMamaCore* serves as abstract base class for all MAMA objects. The MAMA API is responsible for the implementation of the functionality. Therefore, requests for the actual specification and the change of qualifiers is transparent to the application object,

```
[ValueMap("0", "200", "201", "300", "400", "500", "600", "700", "800"),
 Values("unspecified", "ADL", "xADL", "CORBA-IDL", "DCOM-IDL", "TINA-ODL",
        "JAVA", "SNMP-SMI", "OSI-GDMO")]
typedef short tSpecLanguage;

[Abstract]
object oMamaCore{
  interface iMamaCore{
    boolean changeQualifier([In] string name, [In] string value);
    string getSpecification([Out] MAMA::tSpecLanguage language);
  };
};
```

The object contains one interface, which declares two actions. The first action can be used to change the values of qualifiers when those qualifiers are explicitly declared as alterable. The application object can override this characteristic for each qualifier. The second action returns the specification of the application object in the requested language.

3.3.3.4. Miscellaneous Definitions

Table 3-9 comprises miscellaneous type definitions that are not discussed with their actual specification. Each definition in this table is further used by the MAMA protocol, the MAMA API, or application services. Furthermore, the definitions are intended to provide a basic set for MAMA applications.

Definition	Description
sNamedValue	This definition is the basic of a name-value list. It combines a name as a unique identifier with a value, a type, and a flag defining access policies.
tAccessFlag	The access flag is defined for the name-value lists used in the protocol.
tElementType	This is the declaration of enumerates for each ADL element. It is used in the directory service to identify groups of specifications.
tEntityStatus	Each application object has an actual status. This information is important for management issues, such as monitoring.
tEntityType	Entity types are used to qualify a MAMA object that represents a certain role of the manager/agent paradigm or a MAMA specific service. Values are manager, sub-manager, agent, sub-agent, dynamic managed object (one that is controlled completely by MAMA, including start and stop), static managed object (one that is only accessed by MAMA to control its behavior), Graphical User Interface (GUI), event server, and directory server.
tMiddleware	This definition is included to identify supported types of middleware. It is used for application management to specify the middleware an application object supports for communication.
tNameValueList	A list of name-value pairs, mostly used in the protocol.
tOperatingSystem	Each managed application object is accompanied with information about the operating system it is able to run on. This definition includes all known operating systems in form of enumerates. The list of operating system is provided by the Internet Assigned Number Authority (IANA) in [IANA-OS].
tPath	A path is a MAMA address of an application object. The definition is used in the protocol to generate complex structures for addressing single objects and objects in hierarchies. Multiple addresses are separated by the character semicolon.
tSecurityLevel	Specifies the security level for the protocol options.
tSpecLanguage	Each application object offers information about its specification in a certain language. Although MAMA initially supports ADL and xADL only, other languages might be interesting for specific use cases.
tURL	A URL is a basic addressing scheme. This type has been introduced to provide a generic description of URLs with minimum permitted values. Values are URL schemes defined e.g. in [IETF-RFC1738].

Table 3-9: Core Model – Miscellaneous Type Definitions

3.3.4. Entity Management

The Core Model provides basic specifications for the management of MAMA core objects. These specifications are collected by the abstract base class *oEntityMgmt*. Each MAMA core object that wants to offer status information just needs to inherit the functionality from this class. The MAMA API is responsible for the implementation of the functionality. The class *oEntityMgmt* identifies four different types of information:

1. information available at the time the application was compiled;
2. information about the installation of the application on a specific node or host computer;
3. information available at the time the application was launched, that is started; and
4. information about the runtime status of the application.

The information on runtime is further divided into general information, URLs that lead to external configuration files, fixed configuration information, variable configuration information, and information about requests that are stored by the API.

The information on each type is collected by a dedicated structure. The class *oEntityMgmt* declares each of the eight structures as an attribute with appropriate access permissions to enable other application objects to access the stored information.

3.3.4.1. Compile Time Information

```
struct sCompileTime{
  unsigned short    numberOfInterfaces;
  MAMA::tTime       compileTime;
  unsigned short    version;
  unsigned short    revision;
  string            cvs;
  MAMA::tEntityType type;
};
```

The compile time information can be generated automatically by an ADL compiler. They include the number of interfaces, the actual time the application was compiled, and the version and revision number of the compilation result (usually an executable). Additionally, a string that describes the identifier of an automatic code revision system for that very compilation and the type of application is included in this information.

3.3.4.2. Information about the Installation

```
struct sInstallation{
  MAMA::tPath pkgLocation;
  MAMA::tDate pkgDate;
  string pkgSerialNumber;
  string pkgProductName;
  string pkgVersion;
  string pkgManufacturer;
};
```

The information about the current installation of the application is derived from [IETF-RFC2287]. They comprise the location of the installation in form of an operating system specific path, the date of the installation. Furthermore, the structure contains information about the software package as there are an optional serial number for the package, the product name of the package that might differ from the application's name, the package version, and the manufacturer of the package. This information should be read-only to avoid manipulation.

3.3.4.3. Information about the Application Launch

```
struct sLaunch{
  string launchUser;
  string launchParameters;
  string launchTime;
};
```

An application is usually launched by a specific user (regarding the domain of the system where the application was started), with a number of parameters, at a specific time. All this information is included in the structure that deals with launch-time information. This information must be read-only to avoid manipulation.

3.3.4.4. General Runtime Information

```
struct sRuntimeGeneral{
  string supportContact;
  string physicalLocation;
  string ID;
  unsigned short boots;
  MAMA::tTime time;
  MAMA::tTime uptime;
  MAMA::tTime localTime;
  MAMA::tOperatingSystem OS;
  MAMA::tMiddleware middleware;
  string host;
  MAMA::tEntityStatus status;
};
```

The general information about a running MAMA application starts with the identification of a person responsible for support. This includes contact information like email and telephone number, and the physical location where the application is running (or the host computer respectively) by means of floor, room, or other appropriate location information.

Each application should be accompanied by a unique identifier. MAMA recommends the usage of a Universally Unified Identifier (UUID; [DCE-RPC]) for this purpose.

Each application can be booted and restarted several times. The time of the last boot can be stored, and the total runtime of the application can be measured. All this information might become useful for performance monitoring during long runtimes.

A running application depends on the middleware it is based on and the operating system of its host computer. The host computer has usually a local time and an IP[2] address. This information is also handled by this structure.

3.3.4.5. Configuration Files

```
struct sRuntimeConfigUrls{
  MAMA::tURL configUrl;
  MAMA::tURL persistentUrl;
  MAMA::tURL logUrl;
  MAMA::tURL eventServerUrl;
  MAMA::tURL entityUrl;
};
```

Application specific parameters that describe the actual configuration of an application are (most preferable) contained in configuration files. These files need to be accessed by the application at startup. The file locations of MAMA applications should be given to the applications in form of a URL (cf. [IETF-RFC1738]). The configuration information is grouped to general information (*ConfigUrl*), information about or location of persistence and log files, the address of the event service, and the application itself.

3.3.4.6. Fixed Configuration Information

```
struct sRuntimeConfigFixed{
  string serialNumber;
  string vendor;
  string manufacturer;
  string modelName;
  string languageEdition;
};
```

[2] Internet Protocol

Fixed configuration information does not change. Most of them are fixed for the total runtime; some of them are changeable but not in short periods of time. This type of information is an application's serial number (that might differ from the installation serial), the vendor and the manufacturer of the application, the vendor-specific model identifier, and the supported languages.

3.3.4.7. Variable Configuration Information

```
struct sRuntimeConfigVariable{
  unsigned short       logLevel;
  unsigned short       debugLevel;
  unsigned short       monitoringLevel;
  unsigned short       transactionTimeout;
  string               securityModel;
  MAMA::tSecurityLevel securityLevel;
  string               operationStatus;
  MAMA::tTime          lastChange;
};
```

Variable configuration information might change or might be changed frequently during the total runtime of the application. The first group relates to the levels of log, debug, and monitoring the application should be configured to. The transaction timeout, security model, and security model are reserved for the protocol. The operation status explains the current runtime status of the application (up, running, initializing, down, broken, waiting). The last member of the structure specifies the last time the application was changed, e.g. a new library was added or configuration data changed.

3.3.4.8. Log Information

```
struct sRuntimeLog{
  MAMA::tTime   lastRequestIn;
  MAMA::tTime   lastRequestOut;
  MAMA::tTime   lastResultIn;
  MAMA::tTime   lastResultOut;
  MAMA::tTime   rejectedRequestsIn;
  MAMA::tTime   rejectedRequestsOut;
  unsigned long requestInCount;
  unsigned long requestOutCount;
  unsigned long resultInCount;
  unsigned long resultOutCount;
  unsigned long rejectedRequestsInCount;
  unsigned long rejectedRequestsOutCount;
};
```

Log information reflects the processing of operation calls (requests) by the application. The first four members of this structure provide information on the last time the application called an operation, an operation was called on the application, the application sent results, and the application received results. The next two members store the time the application has rejected an operation call or received the rejection of its own operation calls. The last six members provide statistical information on sent and received operation calls, results, and rejects.

3.4. Application Protocol

The Application Protocol defines the mechanisms of data exchange between applications and among applications and services. The protocol has the following objectives:

1. Provide a generic mechanism for the communication between distributed objects. This mechanism needs to be independent of concrete middleware technology but should be mapped to employed middleware easily.

2. Realize the communication between MAMA objects supported by the API. The protocol defines communication behavior and the API implements it. The communication between MAMA objects is done via the exchange of either ADL typed or xADL typed parameters.
3. Support of management hierarchies. Management systems assume the existence of a management hierarchy that aligns managers, agents, and managed objects in form of a management tree. Since distributed objects are only loosely coupled, this special requirement needs to be supported by the protocol with mechanisms for hierarchical addressing of objects, scoping and filtering, and the realization of transactional operations.

The communication between application objects can be divided into three different layers. As Figure 3-6 shows, the first layer describes the relationships between the core objects of applications. These objects exchange information by means of messages or operation calls to realize the aims of the application. The second layer models the transmission of messages and/or operation calls between MAMA API objects. The MAMA protocol specifications reside in this layer. The third layer is dedicated to the employed middleware and the middleware specific protocol. Further layers are not needed since the middleware protocol already abstracts from underlying network transport protocols.

This approach can be compared to the Reference Model for Open Systems Interconnection (RM-OSI; [ITU-X200]). The layers are independent of each other. Protocol services are offered via Service Access Points (SAP) here called interfaces. The MAMA API utilizes the ADL specification of the system to offer the application core objects with an interface for message transfer and operation calls. This interface, described in ADL, is mapped to the actual implementation language of the core object. Therefore, it is middleware generic but programming language specific. The interface to specific middleware architectures is specified middleware generic in ADL and needs to be mapped to the actual middleware. The API must implement this interface for each type of middleware.

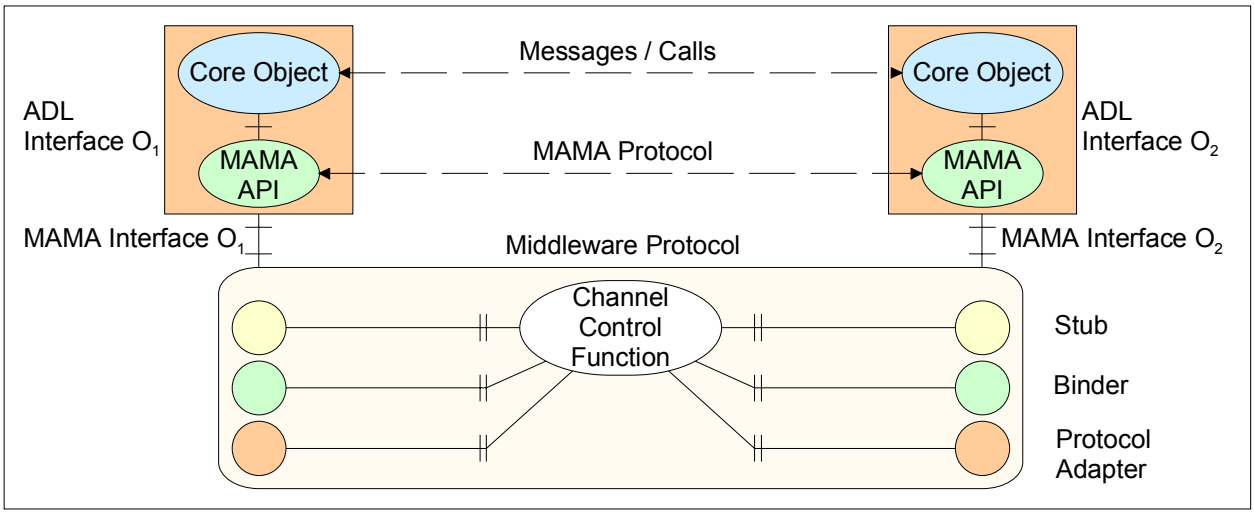

Figure 3-6: MAMA – Protocol

The MAMA protocol is a combination of the MAMA interface and interaction schemes that are defined by the actions and attributes of application objects. The MAMA interface is specified in ADL in a way that it can be easily mapped to any concrete middleware language.

The design of a middleware independent protocol implies an exact method for the mapping of information from the core objects via the MAMA API towards the actual employed middleware protocol. This mapping is characterized by the interfaces each layer offers. The core objects of applications communicate via messages or operation calls. The information exchanged is ADL typed data. This information needs to be mapped to the MAMA interface. This mapping is realized by the MAMA API, which offers programming language specific methods for message exchange and operation calls. Furthermore, the API converts the information into the middleware specific protocol, which is an environment specific modeling of the MAMA interface.

3.4.1. Protocol Specification

```
MAMA::tNameValueList swAction([In] tOperation operation,
                              [In] tSeqObjectPath addresses,
                              [In] MAMA::tNameValueList parameters,
                              [In] MAMA::tNameValueList options);
```

The MAMA interface is specified in ADL. The interface is modeled following management protocols. The interfaces of managed objects rely on a fixed set of operations that are offered by the management protocol. SNMP grants operations to retrieve information about attributes and to alter them. The Common Management Information Protocol (CMIP) adds a specialized operation called *action* that is capable of wrapping specific operations of managed objects. Thus, management systems decouple the access to managed objects and their behavior from the description of that very behavior. The latter is described by management languages (SMI, Guidelines for the Definition of Managed Objects – GDMO). Whatever the definition of a managed object includes, the operations to access this managed object are fixed to:

- get – retrieve information of a managed object and its attributes;
- set – alter information of a managed object and its attributes; and
- action – perform a further described specific operation on a managed object or its attributes.

Applying these three general operations to a middleware interface definition results in the following code:

```
void          set (listOfParameters params);
NameValueList get (listOfParameters params);
NameValueList action (string operation, listOfParameters params);
```

This interface depends not on the middleware interface definitions and it is not subject to be changed. Therefore, the signature of the interface will not change and a re-compilation of client or server is not necessary when the interface definitions of the computational objects change.

The three operations can be further compressed. The operation *action* already includes a string that identifies the requested operation. This string can also have the values *set* or *get*, so that the first two operations of the given interface are not needed.

```
NameValueList action (string operation, listOfParameters params);
```

The result is one interface operation that wraps concrete operations of application objects. This is exactly what the MAMA interface is doing. This interface operation is called *swAction* to distinguish it from normal interface operations and to show the conceptual relationship to management protocol operations especially the *action* operation of CMIP. The operation *swAction* returns a data type *tSeqNamedValue* and gets several arguments that are necessary for the correct handling of operation calls. The arguments are the called operation, a list of addresses, the parameters for the called operations, and a list of options for the parameterization of the protocol itself.

3.4.1.1. Operation

```
typedef string tOperation;
```

The required operation is presented in form of a string. This string can be interpreted by the MAMA API to select an appropriate function on the application's core object or an API specific operation. This parameter of the operation *swAction* must follow the rules of identifiers as described in section 3.2.1.3.

3.4.1.2. Addresses

```
typedef MAMA::tPath[] tSeqObjectPath;
```

For each call to an operation, the calling object must specify the objects where this very operation should be executed. In classic middleware, this address is a single object. For management systems, those addresses need to cover management hierarchies and groups of objects that should be called at once. Therefore, the address field in the protocol interface specification is modeled as an array of strings.

Each operation call can be associated to any infinite non-negative number of addresses. Each address is represented by one member of the string array *tSeqObjectPath*. With this mechanism, group of objects can be contacted by an application with only one function call. Protocol options can be applied to indicate the policy for processing this operation call for each individual object of the array.

Each string of the array describes either a single identifiable object (like in classic middleware) or the path in a management hierarchy to a specific object (as in management systems. When a single object is addresses, the processing of the operation is simply to call the operation on the object.

The situation becomes more complex when complete object paths are given as address. In this case, the path will be traversed and evaluated at each node. These nodes will change the path (by removing themselves from the path) and look for options that indicate a local execution or a further forwarding to other objects addressed in the path.

The part of the protocol that is visible to the applications core objects handle MAMA specific addresses. The actual naming conventions for MAMA objects are not specified by the protocol itself but by the MAMA naming service. The protocol demands only that object names are given in form of strings. The protocol itself processes MAMA addresses and transforms them into middleware specific addresses. This is done by the help of the naming service. The application core object only needs to deal with MAMA addresses.

3.4.1.3. Parameters, Options, and Return Values

Parameters, options, and return values are transmitted in form of Name-Value Lists (NVL). An NVL is a complex map for exchanging typed values in form of a sequence. This concept can be found in many application protocols although it is not every time called so. SNMP uses variable bindings and CMIP managed objects contain attributes composed as name-value lists. The MAMA protocol specifies further a set of flags to configure the behavior of operation calls. Those flags are combined in the option field of the protocol.

Name-Value List

```
struct sNamedValue{
  string name;
  string value;
  tDataType nvDataType;
  tAccessFlag nvAccessFlag;
};
typedef sNamedValue[] tNameValueList;
```

The name-value list for the MAMA protocol is a list that collects any number of *sNamedValue*. This again is a structure that contains all information on a single value that is necessary for marshaling/demarshaling. The first three members describe the parameter that should be transmitted:

- *name* is the name of the parameter;
- *value* is the value assigned to the name;
- *nvDataType* represents the ADL data type of the parameter; and
- *nvDataFlag* characterizes the access permissions for each value.

With these three members it is possible to marshal any declaration of a basic data type. Declarations that are based on new type definitions can be decomposed until a basic data type matches. The fourth member of the structure is used to indicate whether the parameter is read only or writable. Read only parameters are not subject to be changed by the called operation. Writable parameters are used as return values.

tDataType

```
[ValueMap("0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11",
          "12", "13", "18", "19"),
 Values("inconsistent", "char", "string", "boolean", "octet", "short",
        "ushort", "long", "ulong", "longlong", "ulonglong", "float",
        "double", "longdouble", "array", "struct")]
typedef unsigned short tDataType;
```

A *tDataType* identifies the ADL basic type of a value to reconvert transmitted values into the original data type. The values *1* to *13* express ADL basic data types. The value *0* indicates a non-ADL data type. New data types, beyond the scope of ADL basic types, can be added in order to model domain specific requirements. Those data types should start not before the value *19*. Data types that cannot be reconverted because their actual type is unknown are converted to a string.

Arrays and structures are structured data types that need to be treated in a special way. For both, the start and the end has to be marked clearly. The protocol marks both data types at the start with the name of the array or structure and with the respective value for the data type (that is *18* or *19*). The end of them is identified by an empty name and the respective value for the data type. Members of structures are characterized by their name, data type, and value. Members of arrays are only characterized by their data type and value. Following this procedure, arrays and structures can be marshaled and de-marshaled by the protocol.

tAccessFlag

```
[ValueMap("0", "1", "2", "4"),
 Values("none", "read", "write", "exec")]
typedef unsigned short tAccessFlag;
```

A *tAccessFlag* specifies flags for the access of values according to UNIX file permissions [SunOS-chmod], as described in section 3.3.2.3. Each value can have a set of flags. It is possible to set all permutations of *read*, *write*, and *executable*. The default is *0* so that the access to a value is no further specified.

This flag relates to the qualifier *In* and *Out*. The value of *read* indicates and *In* parameter, the value of *write* indicates an *Out* parameter, and finally the combined value of *read* and *write* indicates an *In* and *Out* parameter. The values none and exec are included to characterize unknown access policies and for the identification of functions that might be called similar to function pointers in C++.

Options

In general, the protocol can be configured by four flags. All flags are transmitted with an operation call in the options filed of *swAction*. Since this field is a name-value list, new options and application specific options can be added easily. The MAMA API is in charge to provide appropriate functionality.

The flag *transactionFlag* requests that an operation call is treated as a transaction. The protocol realizes a 2 Phase Commit (2PC; [Heuer97]) automatically and returns only success or non-success to the calling application object. The flag *localExecutionFlag* is included for operation calls that take place in object hierarchies. This flag demands each object that receives an operation call to execute the included action locally. The flag *recursivelyFlag* is also defined for hierarchical structures. When this flag is activated, each object receiving an operation call is commanded to forward this operation to subordinate objects in the hierarchy. The last flag, *entityType*, can be used to specify on which type of application objects an operation call should be executed. This flag can be used in hierarchies or when the application type is actually not known but used as a policy for the execution.

Security Options

Security requirements of the protocol are authentication of objects and encryption of communication data. The following security levels are supported by the protocol

1. Level 0: No security, plain communication among objects.

2. Level 1: Object authentication (for caller and callee). Protection against active interventions which may try to sabotage the system.
3. Level 2. Authentication of objects and encryption of communication data. Protection against active interventions and passive interventions trying to catch information.

The security options can be configured for each call. The configuration is realized by setting the flag security in the option filed with the integer value of the requested security level. Server objects can demand a certain level, which than has to be followed by client objects. When an incoming communication request does not comply with the adjusted security level or if the communication could not be established at the wanted security level (e.g. authentication failures, decryption failures) the appropriate security exceptions have to be raised.

3.4.1.4. Example

The following example shows how the NVL can be used to marshal ADL data types. The example includes an array with three members, a structure with two members, and some standalone data. First, the ADL declaration of the data types and an imaginary function are given:

```
typedef string[] aliases;
struct person{
  string name;
  string famelyName;
};
action setUser ([In] aliases nicknames, [In] person userName,
                [In] short yearOfBirth, [In] longlong hashId);
```

The function *setUser* can now be called from an application. The information about a user is filled in the array *aliases* and in the structure *person*. The information *yearOfBirth* and *hashId* are generated when the function is actually called. The following code shows a C++ notation.

```
aliases al;
al[0] = "vdmeer@cs.tu-berlin.de";
al[1] = "vdmeer@fokus.fhg.de";
person pr;
pr.name = "Sven";
pr.famelyName = "van der Meer";
setUser(al, pr, 1971, 83e2855d8fd96e64785e50500c492cc0);
```

The protocol transforms the parameters of the called function into a name-value list to generate the *parameters* of the ADL action. The result has the following form, also presented in C++ notation.

```
parameters[0].name = "nicknames";
parameters[0].value = "";
parameters[0].nvDataType = 18;  // for array
parameters[0].nvAccessFlag = 1; // for read
parameters[1].name = "";
parameters[1].value = "vdmeer@cs.tu-berlin.de";
parameters[1].nvDataType = 2;   // for string
parameters[1].nvAccessFlag = 1; // for read
parameters[2].name = "";
parameters[2].value = "vdmeer@fokus.fhg.de";
parameters[2].nvDataType = 2;   // for string
parameters[2].nvAccessFlag = 1; // for read
parameters[3].name = "";
parameters[3].value = "";
parameters[3].nvDataType = 18;  // for array
parameters[3].nvAccessFlag = 1; // for read
parameters[4].name = "userName";
parameters[4].value = "";
parameters[4].nvDataType = 19;  // for struct
```

```
parameters[4].nvAccessFlag = 1; // for read
parameters[5].name = "name";
parameters[5].value = "Sven";
parameters[5].nvDataType = 2;   // for string
parameters[5].nvAccessFlag = 1; // for read
parameters[6].name = "famelyName";
parameters[6] value = "van der Meer";
parameters[6].nvDataType = 2;   // for string
parameters[6].nvAccessFlag = 1; // for read
parameters[7].name = "";
parameters[7].value = "";
parameters[7].nvDataType = 19;  // for struct
parameters[7].nvAccessFlag = 1; // for read
parameters[8].name = "yearOfBirth";
parameters[8].value = "1971";
parameters[8].nvDataType = 5;  // for short
parameters[8].nvAccessFlag = 1; // for read
parameters[9].name = "hashId";
parameters[9].value = "83e2855d8fd96e64785e50500c492cc0";
parameters[9].nvDataType = 11;  // for float
parameters[9].nvAccessFlag = 1; // for read
```

This resulting sequence can be transformed to any specific middleware, since every interface definition language supports lists, arrays, or sequences of structured data.

3.4.2. Protocol Information Flows

The protocol information flows explain how the protocol supports MAMA applications. Each application must check in on the naming service in order to enable other applications to find available functionalities. Furthermore, applications might want to register at the event service to send events or to retrieve events from other objects. Both activities belong to the initialization process. The protocol offers the basic mechanisms for this initialization. Figure 3-7 shows the protocol checkpoints for the initialization.

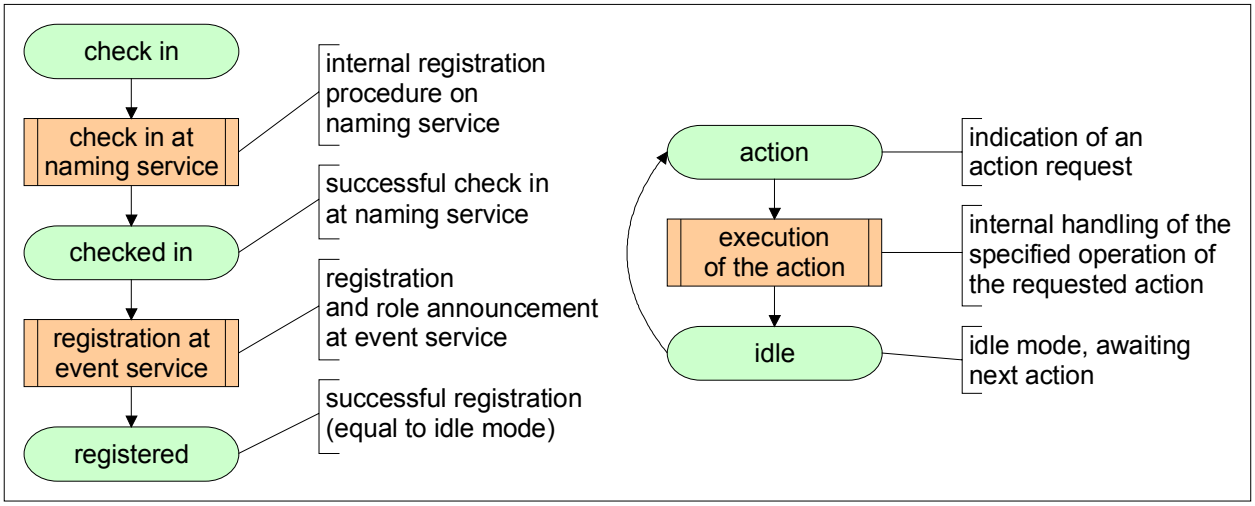

Figure 3-7: Protocol – Protocol Checkpoints [Fritzsch01]

The initialization of a MAMA application starts with the check-in at the naming service. The specific operations of the naming service are not part of the protocol. They are discussed in section 4.4. The protocol connects to the naming service, calls the appropriate operation for check in, and handles further processing until a successful check in is confirmed by the naming service along with a unique identifier for the application object. When requested by the application, the protocol proceeds with the registration at the event service. The final checkpoint of the initialization is *registered*. An application that reaches this state operates as a MAMA application.

The state *registered* is equal to the checkpoint *idle*, shown in the right side of Figure 3-7. After the initialization, the protocol loops between *action* and *idle*. *Action* indicates that an action of the core object is called (server) or that an action of another object should be called (client). The protocol performs the requested action and returns to the state *idle* awaiting the next action.

3.4.2.1. Registration on the Naming Service

The registration on the naming service is explained in section 3.6.1.5. This section describes the interface of the MAMA naming service including operations for the registration of new object instances.

The protocol connects to the naming service. In fact, the protocol searches for an available naming service on the local machine, than in the local domain. When no naming service is found, it starts a local naming service. Once connected to the naming service, the protocol takes care of the registration. The return value *not registered* is aligned with a specific error message to enable a separation of failures that might occur on the naming service.

3.4.2.2. Registration on the Event Service

A MAMA application can register on the event service. The application can decide to register as consumer, as producer, or in both roles. The latter action can be achieved by registering first as consumer and later as producer (or vice versa).

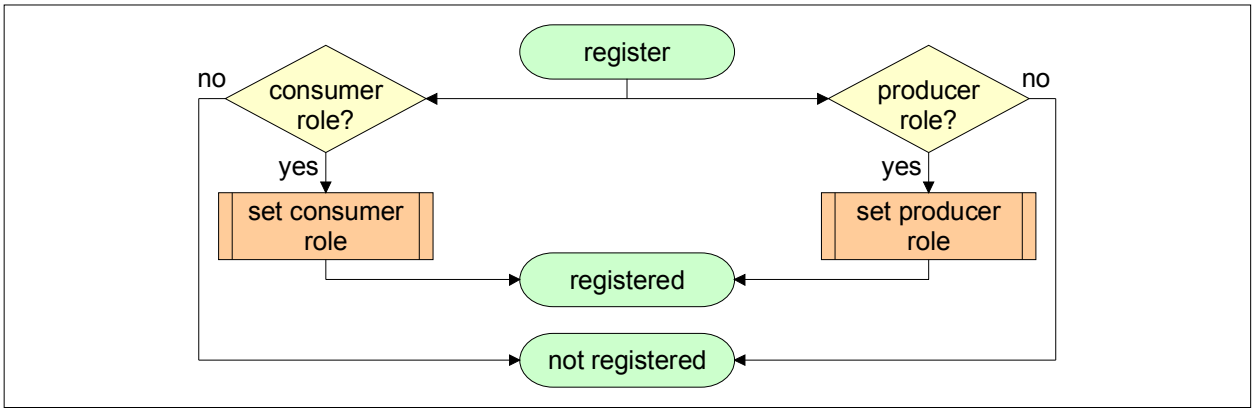

Figure 3-8: Protocol – Registration on Event Service [Fritzsch01]

The protocol connects to the event service. In fact, the protocol searches for an available event service on the local machine, than in the local domain. When no event service is found, it requests the activation of a local event service from the naming service. Once connected to the event service, the protocol takes care of the registration in the requested role. In the case that no role is specified, the protocol returns the state *not registered* otherwise *registered*. Additionally, the return value *not registered* can be aligned with a specific error message to enable a separation of failures that might occur on the event service from the simple error that no role was specified by the application.

The protocol needs to be supplied with some information in order to register correctly and to process incoming events for the application. First, the role needs to be identified. Next, the application must provide callback functions that can be invoked in case an event is received for the application. The Core Model specifies several classes of events. An application can assign a callback function for each event class to enable a different processing of information, warning, and error events. Furthermore, the application can subscribe available event channels on the event service. This subscription is supported by the protocol by calling the appropriate function on the event service.

3.4.2.3. Action Processing

For each operation call that is issued to an application, the protocol is responsible for the evaluation of the related information. The protocol searches for an appropriate operation in its local database. When the operation was not found, an exception is generated and returned to the calling object. Otherwise the protocol can proceed. The next step is to filter the options of the operation call. As introduced in the protocol

specifications, an action can be applied with two flags that indicate local execution and forwarding. An additional flag identifies types of objects the operation should be forwarded to. The protocol analyzes the flags to decide whether the operation should be executed locally, forwarded to specific types of objects, or forwarded in form of a broadcast to all objects connected to the application.

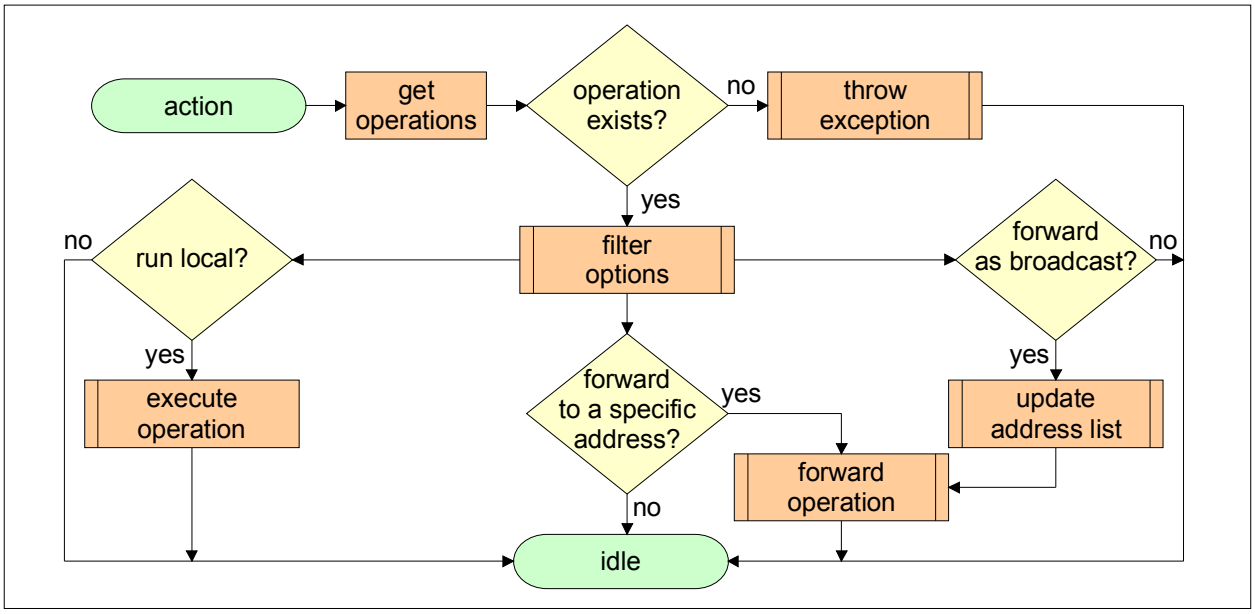

Figure 3-9: Protocol – Action Processing

The local execution and the forwarding can be combined so that an operation is executed locally and forwarded. After the evaluation of the filter parameter, the protocol invokes the proper action, transmits possible return values, and returns to the state *idle* awaiting new operation calls.

3.4.2.4. Registration of Application-specific Operations

Initially, the protocol supports only pre-defined operations from the MAMA Core Model. Application specific operations must be registered. The protocol handles a simple database with all available operations of a MAMA application to route an action request to the appropriate action.

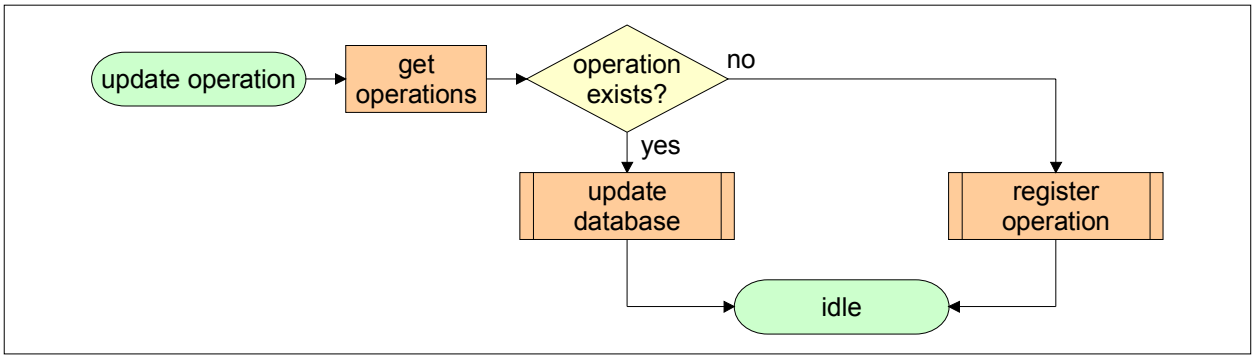

Figure 3-10: Protocol – Registration of Application-specific Operations

Operations can be registered with the protocol at any time during the runtime of the application. This enables the registration of new operations at startup, the registration of new operations that are dynamically added to the application, and the update of already registered operations when they have been changed during runtime. The protocol searches its internal database for the operation that should be registered. If found, the operation is added to the database, otherwise the database will be updated.

The integration of this behavior in the protocol offers applications to run on different roles. They can operate as node objects inside of a hierarchy or as leaf objects. It is also possible that an application be-

longs to more than one hierarchical structure of objects and that it runs in more than one role at the same time. Considering a management system, an application can act as a manager, an agent, and a managed object simultaneously.

3.4.2.5. Sequence Diagram

Figure 3-11 shows the sequence diagram of the whole lifecycle of a MAMA application. Application *A* invokes its initialization on startup. The protocol connects to the naming service to check in and to receive a unique identifier for the application that is used as address by other applications. The second step is the registration on the event service. This service returns general information on available event channels and registers the application for certain types of events.

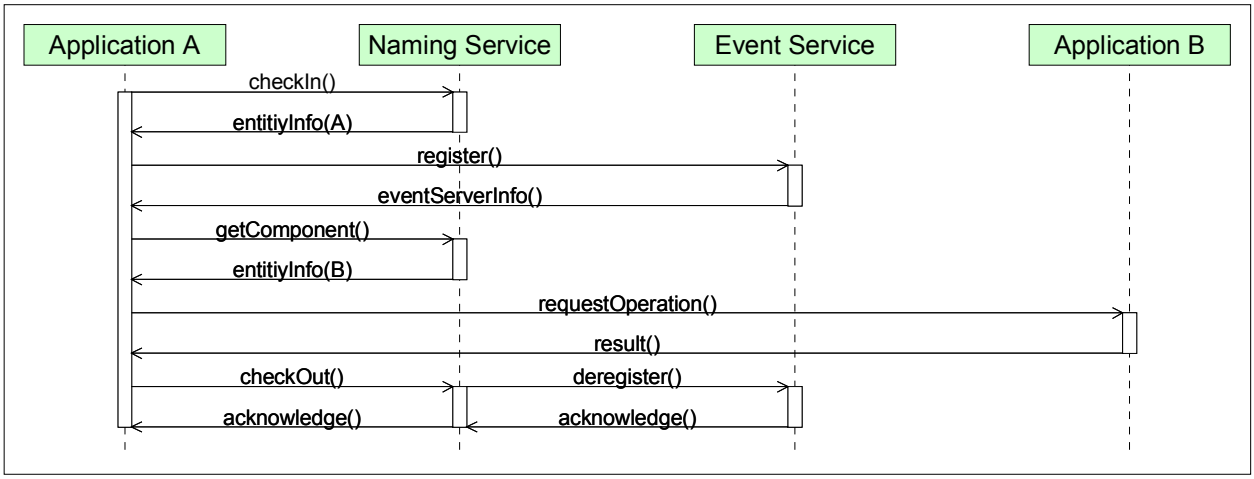

Figure 3-11: Protocol – Sequence Diagram [Fritzsch01]

Application *A* is now a registered MAMA application. In the given example, *A* requests information about an application *B* from the naming service. The returned information might include more than just the address as for example information on the application's interfaces. Now, *A* calls an operation on *B* and *B* returns the results to *A*. The protocol is responsible for the connection between *A* and *B* and for the transmission of data (that is marshaling and de-marshaling). Finally, *A* terminates. The termination can be done right after the protocol was requested to checkout of the naming service. In Figure 3-11, the naming service sends an event to the event service about the termination and the event service automatically deregisters application *A*. The protocol receives the final acknowledgement and application *A* can terminate.

3.4.3. Protocol Support for Hierarchies

The protocol supports hierarchies as used in management systems and peer to peer networks by means of addressing objects within a hierarchy. Additionally, objects can be filtered and scoped, and operations can be forwarded via a set of objects until the finally addressed object is reached.

Hierarchical structures consist of objects that offer specific functionality. In a management system, those objects are managers, agents, and managed objects. In a peer to peer network those objects might offer distributed lookup and discovery services, persistency, and resource management. Objects can invoke operations on other objects and they can receive notifications on occurred events. Both types of communication can be either synchronous or asynchronous. Policies can be applied for the forwarding of operation calls and notifications. The following subsections introduce the application of the protocol for addressing objects within hierarchies taking a management system as example. The mechanisms can be also employed by other applications, such as peer to peer networks or intelligent agents.

Management operation requests are forwarded along the hierarchical structure of the management tree to the agents or the managed objects, on which these operations should be performed. Therefore, the address of an object consists of a description of the path to this object and unique an object identification. The

path description is composed by one or more unique object identifications of agents, which are responsible for the addressed agent or managed object.

An address may regard to single or multiple agents or managed objects (e.g. all objects of an agent or even all objects of a specific sub-tree). Addresses and lists of addresses are assigned to a management operation request. The protocol already offers flags in the option parameter of the operation *swAction* that describe the execution policy of operations. For better understanding, the following abstract operation is used instead of the complete *swAction*. The field *list of entities* contains all information regarding the addresses of objects. The field *flag* includes the options introduced in section 3.4.1.3.

```
Op('list of entities', 'flag', ...)
```

A hierarchy consists of nodes and leafs. Each node and each leaf represent an object of a distributed application with a special functionality. Nodes can forward operation calls to other nodes or leafs. Nodes can also execute operations locally. Leafs receive operation calls and execute them. Both, leafs and nodes, can send notifications to other objects in the hierarchy. In a management system, nodes are called agents and leafs are called managed objects.

3.4.3.1. Addressing Node Objects

Each node in the hierarchy of a management system is provided with the information that it belongs to the group of agents. Addressing a node in the hierarchy now implies to set the flag *entityType* to *agent*. The management protocol can evaluate this flag at each node to invoke the proper operation or to discard the operation completely. The two other flags of the option field of the protocol have the following meaning:

- When the *recursivelyFlag* is set, the operation is forwarded from each node to all other subordinate nodes. No action regarding a forward is done when this flag is not set.
- When the *localExecutionFlag* is set, the operation is executed by the node itself. No action regarding a local execution is done when this flag is not set.

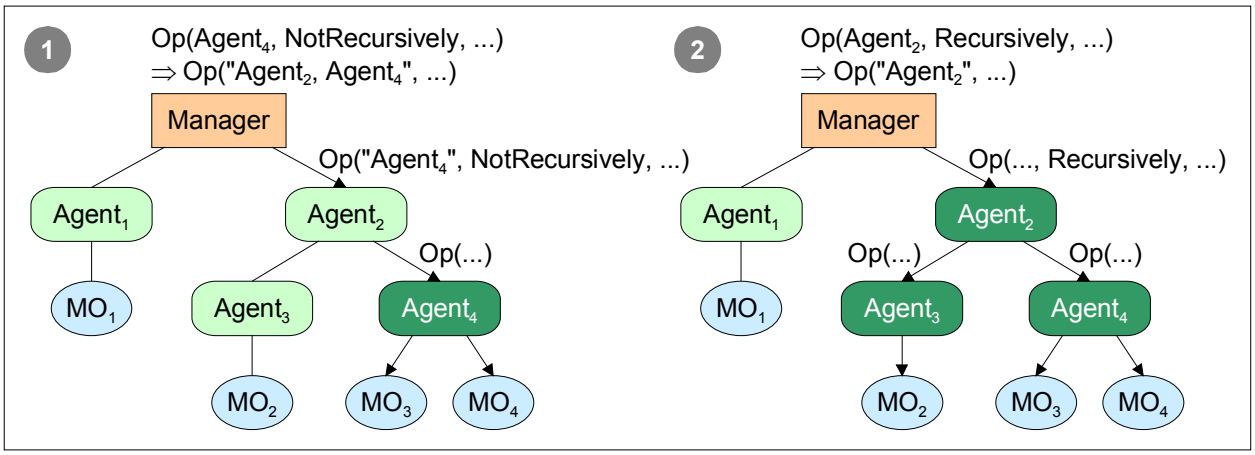

Figure 3-12: Protocol – Addressing of Nodes

The two flags do not affect each other. In the worst case, the operation is neither executed locally nor forwarded to other nodes. Furthermore, the two flags allow three variants.

1. An operation is executed locally and not forwarded to other nodes.
2. An operation is not executed locally but forwarded to other nodes.
3. An operation is executed locally and forwarded to other nodes.

Figure 3-12 shows the effects of the *recursivelyFlag*. In case 1, a single node is addressed directly. The operation is initiated by the object *Manager* and forwarded via the object $Agent_2$ to the object $Agent_4$. In case an operation is called by the object *Manager* on the object $Agent_2$ with an activated *recursivelyFlag*. The object $Agent_2$ automatically forwards this operation call to the objects $Agent_3$ and $Agent_4$. The *loca-*

lExecutionFlag is used to ensure that the operation is executed on the addressed objects (*Agent*$_4$ in case 1, *Agent*$_3$ and *Agent*$_4$ in case 2) only or also on the objects that forward the operation (*Agent*$_2$ in both cases).

The usage of these two flags can be combined with the addressing of objects supported by the management protocol. The address field of the protocols allows for multiple addresses and object paths. This means, the object *Manager* is able to invoke the same operations not only on the nodes *Agent*$_2$, *Agent*$_3$, and *Agent*$_4$ but with the same call on the object *Agent*$_1$.

3.4.3.2. Addressing Leaf Objects

Each leaf in the hierarchy of a management system is provided with the information that it belongs to the group of managed objects. Addressing a leaf in the hierarchy implies to set the flag *entityType* to *managed object*. The management protocol can evaluate this flag at each node to invoke the proper operation in the managed object itself. The option *recursivelyFlag* has no special meaning and will not be interpreted by leaf objects. The flag *localExecutionFlag* is evaluated and the operation is invoked locally when set. Otherwise the requested operation is forwarded up to the addressed leaf object.

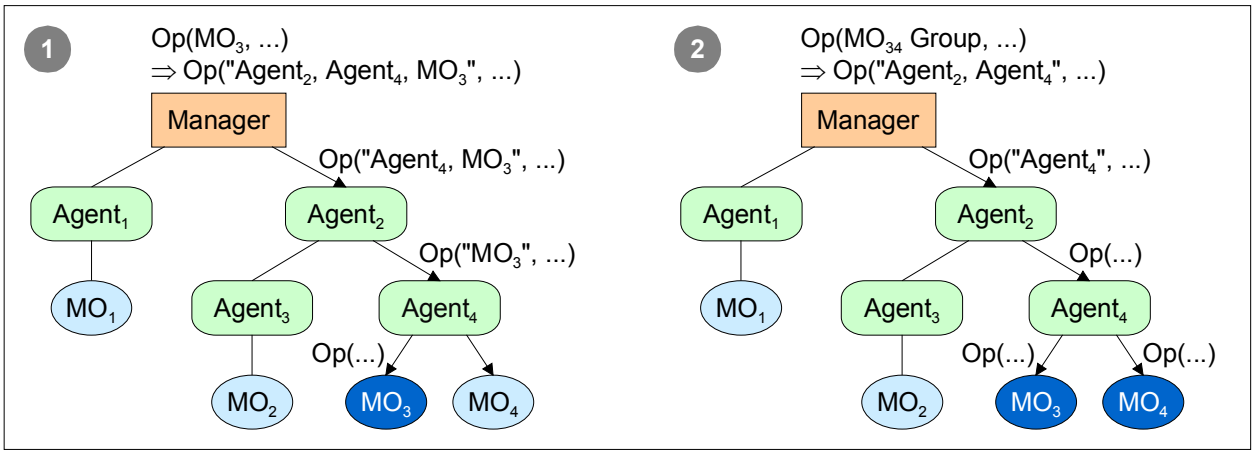

Figure 3-13: Protocol – Addressing Leafs

Figure 3-13 shows the two possibilities for addressing leaf objects within hierarchies. In case 1, the object *Manager* addresses the object *MO*$_3$ directly. All objects in between *Manager* and *MO*$_3$ forward the operation request. Case 2 depicts the addressing of two leaf objects with a group call. The object *Manager* addresses the objects *MO*$_3$ and *MO*$_4$ with one function call. This kind of addressing can be achieved by two different ways. First, the address filed can include the address of the object *Agent*$_4$ and the options field has an activated flag *recursivelyFlag* and a deactivated flag *localExecutionFlag*. The second possibility is to address both objects, *MO*$_3$ and *MO*$_4$, directly with an activated flag *localExecutionFlag*. The flag *entityType* must be set to *managed object* for this scenario.

3.4.4. Protocol Support for Transactions

The MAMA protocol supports transactional operations. The necessity of the transaction concept refers to operations changing the state of multiple objects within the system. If the states of multiple objects depend on one another, the performance of operations on these objects may be only sensible when they are performed successfully on all affected objects. If the operation could not be performed on one or more objects, the system is in an inconsistent state. This may influence the system's runtime behavior negatively and has to be avoided. The MAMA protocol employs the 2PC protocol. Additionally, transactions are combined with the support for hierarchies so that transactional operations cannot only be provided for peer to peer object communication but also for hierarchical object communication.

The protocol applies transaction processing to an operation call when the *transactionFlag* in the options filed is activated. The MAMA API is in the position to offer a configuration function that activates this flag for all operation calls or for the communication with a certain object group, or for the communication with a certain object.

When the flag *transactionFlag is activated*, the protocol generates a Transaction Identifier (TID) for the related operation call. The protocol must be supplied with information, how the requested operation can be rolled back, that is how the object can be returned to the state it had before the transactional operation was executed. This information, usually a function call on the application object, is stored together with the TID. This is the mechanism to remember an undo operation as long as the transaction is active. The application object has to take care that the altered data cannot be changed until the end of the transaction, e.g. applying locking mechanisms.

When the transactional operation is performed on multiple objects, each individual call is treated as a single transaction. Node objects must store all TIDs of subordinate objects and the rollback mechanism for each operation call. When node or leaf object do not support transactional operations (e.g. when they are not implementing transaction processing), the superior objects can simulate a transaction by interpreting return values and restore the object's pre-transactional state through specific operations.

Figure 3-14 and Figure 3-15 show the protocol handling of a successful and a non-successful transaction in four steps. The initialization of the transaction is identical for both. The object *Manager* requests an operation that should be executed on the objects MO_1, MO_2, MO_3, and MO_4 as a single transaction. It generates two TIDs, one for the operation call to the object $Agent_1$ and one for the operation call to the object $Agent_2$. Now, it requests the forwarding of the operation. Subordinate objects (in the example $Agent_3$) perform the same actions.

3.4.4.1. Successful Transaction

The first step is finalized by the successful execution of the requested operation on all leaf objects. In step 2 of a successful transaction (case two of Figure 3-14), all leaf objects have executed the requested operation successfully and return the TID to notify the object *Manager*.

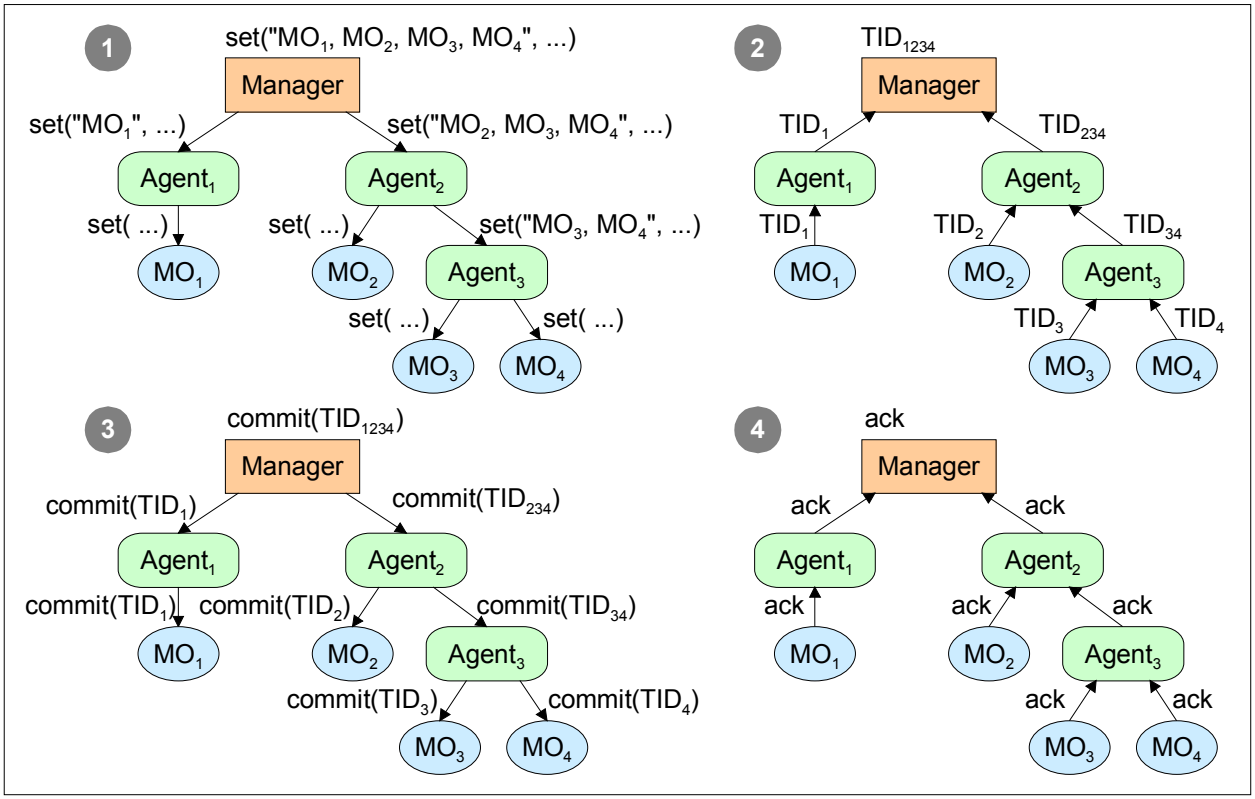

Figure 3-14: Protocol – Successful Transaction

Since each operation is treated as a single transaction, all node object and the object *Manager* follow the same procedure. When they have received all TIDs from subordinate objects, they return the TID related to the communication with their own superior object to that very object. Finally, the object *Manager* receives the two TIDs it has generated by itself from the node objects $Agent_1$ and $Agent_2$

Now, the object *Manager* invokes step 3. It sends a commit message to all objects to indicate that the transaction was completely successful and that no rollback mechanism has to be performed. This commit message is forwarded up to the leaf objects.

The final step number 4 comprises the emitting of an acknowledgement message from all involved objects. At the same time, all information related to the transactions is removed by the objects. The objects receiving a commit message release the locks of its data and the transaction related information. The system has reached its final state and is ready for further transactions.

3.4.4.2. Non-successful Transaction

In a non-successful transaction, the first step could not be performed on all objects successfully. The second step of a non-successful transaction is shown by case 2 in Figure 3-15. In the given example, the requested operation could not be executed on the leaf object MO_4. This event changes the steps 2, 3, and 4. In step 2, the leaf object MO_4 returns an abort message to the superior object. The node object $Agent_3$ receives one commit message and the abort message. It stores the TID from the leaf object MO_3 to remember that the operation was successfully executed there, and the abort message for MO_4. Now, it sends an abort message to its own superior object. At the end of the chain, the object *Manager* receives one commit message (from $Agent_1$) and one abort message (from $Agent_2$).

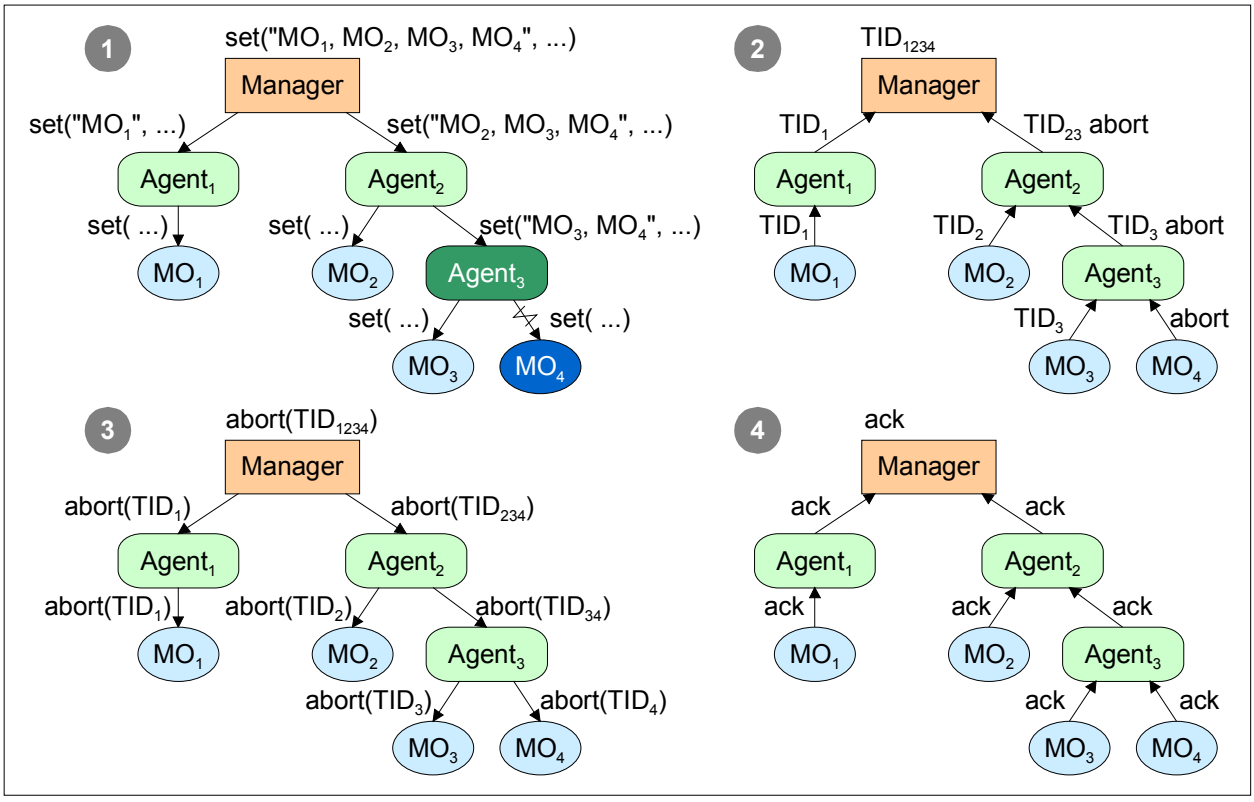

Figure 3-15: Protocol – Non-successful Transaction

In step 3, the manager sends an abort command to all objects. All objects that receive the abort command invoke the proper actions to rollback the already performed operation. Step 4 shows that all objects that successfully realized the undo actions send an acknowledge message. When the object *Manager* has received all acknowledgements, the system is in the same state as prior to the transaction.

3.4.4.3. Additional Issues

The 2PC protocol has several disadvantages. The commit, abort, and acknowledge messages need to be transmitted successfully in order to realize the transaction. In the case that these messages are not received by the addressed objects, the system might run into a deadlock awaiting messages and locking other transactions or operations. For this reason, the 3-Phase-Commit (3PC) protocol has been developed.

This protocol provides a secure handling of transactions, but increases the protocol overhead and the number of communications during a transaction. The 3PC protocol was tested and this overhead prevented the adoption of this protocol.

MAMA applies a more simple mechanism to avoid deadlocks. Each transaction is accompanied with a timeout. This parameter specifies that objects should invoke a rollback automatically when they have not received messages within a certain time slot. A transaction is no longer valid after the time out has been reached. The MAMA API offers the configuration of this time out.

3.5. Application Programming Interface

The MAMA API in an interface for application core objects to communicate with MAMA services and with other application core objects. The API has three different assignments:

1. Decouple the core object from any concrete middleware.
2. Provide a simple, unified interface to the functionality offered by MAMA.
3. Introduce basic management functionality transparent to the core object.

The API is placed between the application core object and the MAMA interfaces. Figure 3-16 shows an engineering view of an application with these three engineering objects. The first objective is achieved already by this engineering design. The API mediates between the core object and the middleware specific interface objects and therefore decouples the core object from the middleware. This objective also demands that the parts of the API that are visible to the core object include no middleware specific operations or parameters, at least not for the communication between objects. The API is responsible for the mapping of ADL typed interfaces to middleware specific interfaces described by the protocol. Furthermore, the API is in charge to realize middleware specific operations (e.g. for initialization) and standard duties (e.g. registration at naming server and event service).

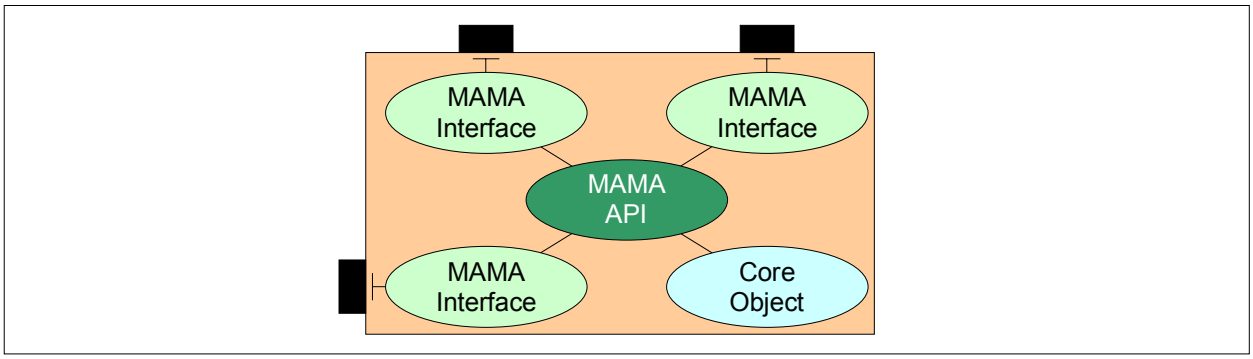

Figure 3-16: MAMA – Application Programming Interface

One part of the API is visible to the core object. The design of this part must meet the second objective. A strictly limited set of operation must be identified in order to provide a simple interface that is portable across platforms on which MAMA is realized.

One of the main tasks of the API is the evaluation of operation calls that are received by the MAMA interface. The API must analyze the parameters of the operation call and forward it to appropriate operations implemented by the core object. This mechanism of intercepting operation calls is now used for the seamless integration of management functionality. Additional interfaces, for the purpose of management, can be added to the application object. Parts of the API can implement the functionality specified in those interfaces and the evaluation of operation calls can result in a forward to those parts of the API, transparent to the core object.

The API itself is separated into libraries. The first library represents the part of the API that is visible by the application programmer. This part is explained in the following section. All other libraries are not visible outside of the API. Instead, they provide generic mechanisms for the management of name-value lists, the access to the specific middleware, the realization of the MAMA protocol, and standard function-

ality like address handling, ticket processing (send and receive), command line parsing, thread handling, logging, and tracing. The internal libraries of the API are the standard library, the middleware specific libraries, and libraries for management functionality. The library for management functionality implements the complete functionality of the object *oEntitiyMgmt* as specified in the MAMA Core Model.

The following section describes the API specification. Descriptive qualifiers are not given for the ADL specifications. Right after the API specification, two supporting libraries are discussed. These are the standard library and the middleware library. They are not part of the API. Those libraries represent recommendations for programmers that implement the MAMA API.

3.5.1. API Specification

The API specification covers the complete functionality that is offered to the core objects. The functions are grouped into initialization functions, functions for the registration on application services, and functions for the communication with other core objects.

The specifications of the API are given in ADL. Therefore, the specifications are independent of the programming language. A mapping from the ADL specification to programming languages has to be done to employ the API specification. Specific characteristics of programming languages might imply minor changes to the specification. Each class and function has been assigned with the prefix *sw* in order to distinguish the API from other APIs.

3.5.1.1. Initialization Functions

Operation initializeEntity

```
[ValueMap("0", "100", "101", "200", "300",),
 Values("standard", "corba", "corba_orbix", "jini", "upnp")]
typedef short swMiddleware;

[MinValue(1), MaxValue(9999), StepIndex(1)]
typedef unsigned long swMaxErrors;

typedef boolean swTransmiterror;
typedef string swServerName;

short initializeEntity([In] swMiddleware mwtype, [In] swMaxErrors maxerrors,
                       [In] swTransmiterror transmiterror,
                       [In] swServerName servername);
```

Issue	Description
General	Core objects that want to act as a MAMA object has to call this initialization function.
Arguments	The arguments the type of middleware the core object wants to use, the maximum number of errors the API should store, a flag to decide whether errors should be send to an event server or not, and the name of the core object requests.
Returns	This function returns *true* when the initialization was successful. Otherwise, the operations returns 1 for a not supported middleware (*mwtype*), 2 for *maxerrors* not accepted, and 3 for problems with the registration at the MAMA naming service. For 1 and two, the initialization must be requested again.
Caveats	*initEntity* registers the standard operations of each MAMA object automatically. Furthermore, this operation registers the object automatically with the MAMA naming service. In case the requested name of the object (*servername*) is already used within the naming service, the object must register itself with another name.

Table 3-10: MAMA API – initializeEntity

Operation configureMiddleware

```
typedef string swObjectPtr;
short configureMiddleware([In] swObjectPtr ptr);
```

Issue	Description
General	In each middleware, a server object must process some initialization routines. This function handles this duty automatically.
Arguments	The function expects to get a middleware specific pointer to an object in order to bind the MAMA interface to this very object. This function is added for middleware environments the employ dedicated binding mechanism.
Returns	This function returns *true* when the initialization was successful, *false* otherwise.

Table 3-11: MAMA API – configureMiddleware

Operation addNewOperation

```
short addNewOperation([In] swOpName opname,
                      [In] string pointer,
                      [In] swOpDescr opdescr);
```

Issue	Description
General	This function registers a new operation with the API. This operation is marked as implemented and available, and will be recognized by the API when incoming requests are processed.
Arguments	Arguments are the name of the operation (as declared in the ADL interface of the object), a programming language specific pointer to the implemented operation, and a short description of the operation similar to the MAMA qualifier description.
Returns	This function returns *true* when the operation was successfully inserted in the operations map, *false* otherwise.

Table 3-12: MAMA API – addNewOperation

3.5.1.2. Server Registration Functions

Operation registerEvSrv

```
short registerEvSrv([In] swEventServerFlags evflags);
```

Issue	Description
General	This function registers an application at an event service. The event service is discovered as described by the MAMA protocol.
Arguments	The application object can specify policies for sending and receiving events.
Returns	This function returns *true* when registration was successful, *false* otherwise.

Table 3-13: MAMA API – registerEvSrv

Operation deRegisterEvSrv

```
short deRegisterEvSrv();
```

Issue	Description
General	This function de-registers an application object from the event service.
Arguments	-
Returns	This function returns *true* when de-registration was successful, *false* otherwise.

Table 3-14: MAMA API – deRegisterEvSrv

Operation changeRegistrationEvSrv

```
short changeRegistrationEvSrv([In] swEventServerFlags evflags);
```

Issue	Description
General	This function changes the policies for sending and receiving events.
Arguments	The policies in form of flags.
Returns	This function returns *true* when the re-registration was successful, *false* otherwise.

Table 3-15: MAMA API – changeRegistrationEvSrv

3.5.1.3. Communication Functions

Operation performAction

```
struct swArgStruct{
  swOpName opName;
  tNameValueList addressList;
  tNameValueList parametersList;
  tNameValueList optionsList;
};

short performAction([In]  swArgStruct arglist,
                    [Out] tNameValueList retarglist);
```

Issue	Description
General	This function serves as the general access point of core objects to the protocol. It has to be called each time a core object wants to invoke an operation on other MAMA objects. *performAction* is responsible for generating the name-value lists required by the protocol and the mapping to middleware specific interfaces.
Arguments	The functions expects an argument list including the name of the called operation, an address list as described by the protocol, a list of parameters in form of a name-value list, and a list of options as described by the protocol. The other argument is used by *performAction* to send back the return values of the called operation. This argument is a name-value list.
Returns	This function returns *true* when the operation was invoked successfully, *false* otherwise. When the return value is *true*, the second argument includes the return values from the called operation. When the return value is *false*, the second argument includes the error code and description in form of a MAMA ticket.

Table 3-16: MAMA API – performAction

Operation sendEvent

```
short sendEvent([In] swEventDescr evdescr, [In] swEventNumber number);
```

Issue	Description
General	This function can be used to send tickets to the event service.
Arguments	The function is parameterized with a description of the ticket and the ticket category. When an application wants to use the facility of individual header and body information (cf. section 3.3.3.2), the event services must be accessed directly (not via the API).
Returns	Returns *true* when the ticket was sent successfully, *false* otherwise.
Caveats	Tickets can be sent at any time, even before registering with the event services. The API stores up to 50 tickets and sends them when the application connects to the event service.

Table 3-17: MAMA API – sendEvent

3.5.2. The Standard Library

The access to and the realization of the MAMA protocol is specified in the standard library of the API. This library can be visible to the application programmer. All data structures that are needed to realize the protocol are specified in classes. Each class is accompanied by one or more member functions that ease the handling of the data structures.

3.5.2.1. Class swNamedValue

The class *swNamedValue* represents an easy mechanism to access and to manage name-value pairs. All protocol information is converted into this kind of structure. This class represents the basis for all lists, since all protocol information is processed with name-value lists. Information on the structure of a name-value list, relevant flags, and options are explained in section 3.4.1.3.

Member Function	Arguments	Description
Name	-	Return the content of the field *name*.
changeName	string	Change the field *name* of the current object.
Value	-	Return the *value* in a string representation.

Table 3-18: MAMA API – Member Functions of the Class swNamedValue

3.5.2.2. Class swOptionsList

The class *swOptionsList* is the most complex class of the API. All information that is placed into maps can be handled by this class. Its main objective is to provide a basis class for name-value lists. Each entry in an option list is a key-value pair, where the value part is represented by a name-value pair. The key depends on the field *name* of an *swNamedValue*. The class itself has defined methods for persistence. This offers the possibility to save current state information into files or data streams.

An implementation of this class can profit from commercial class libraries, which offer basic functionality for list handling. However, the implementation should hide additional libraries. This prevents a MAMA application from becoming dependent on specific commercial class libraries.

Member functions that insert entries into the map do not overwrite existing entries. When an entry should be added to the map that already exists, the functions return *FALSE*. This value is also returned when iterators have reached the end of the map or the map is empty.

Member Function	Arguments	Description
changeNV	name-value pair	Change the current entry to the given name-value pair. This will change all information in this entry to the given pair.
concat	option list	Concatenate the current and the given option list to one list. The sequence of the entries may change after this operation.
contains	string	Return *true* if the argument matches a key in the map.
current	-	Return the current entry as name-value pair.
entries	-	Return the count of entries in the list.
getNext	name-value pair	Change the argument to the next entry in the map.
getNextKey	string	Change the argument to the next key in the map.
insert	name-value list	Insert the given argument into the map.
next	-	Increase the internal iterator by one.
remove	string	Remove the entry matching the argument.
reset	-	Reset the internal iterator for the current list to the first entry.
showAll	-	Stream all entries to the standard output device.

Table 3-19: MAMA API – Member Functions of the Class swOptionsList

3.5.2.3. Class swOperationMap

The class *swOperationMap* is the part of the API that deals with all ADL defined actions of an application object. It should exist only with one instance for each core object. This instance includes standard API operations, special API operations from other API libraries, and the application specific operations. API operations are automatically registered with the map, while application specific operations must be added manually during the initialization of an application object.

Member Function	Arguments	Description
contains	string	Returns *true* if argument matches an operation, *false* otherwise.
getNext	-	Returns the name of the next operation in the map. When no more operations are available, the string is empty.
getCurrent	-	Returns the current key.
getCurrentDescription	-	Returns the current operation's description, if available.
reset	-	Resets the internal iterator of the map.
insert	key, operation, description	Inserts the parameterized operation. It will overwrite existing operations with the same name.
entries	-	Returns the number of available operations as unsigned long.
showAll	-	Stream all entries to the standard output device.
list	operation	Return name and description of the matching operation.
listAll	-	Return a list of all associated operations.

Table 3-20: MAMA API – Member Functions of the Class swOperationMap

An operation map actually consists of two maps. The first map collects function pointers to the operations implemented by the core object or other API libraries. All implemented operations have the same signature that is identically to the MAMA protocol. The evaluation of parameters and operation-specific options is up to the implementation. The second map includes descriptions for each operation. An application object can use *addNewOperation* to register new operations.

3.5.2.4. Class swAddressList

The protocol specifications for addresses are realized by the class *swAddressList*. It is the API equivalent of the *prSeqObjectPath* of the protocol (cf. section 3.4.1.2). The class handles all requirements of the addressing of objects within hierarchies, including the manipulation of address lists before the actual forwarding of operations to subordinate objects.

Member Function	Arguments	Description
entries	-	Returns the count of addressed components as unsigned long.
getFirst	-	Returns the first existing path.
removeFirst	-	Returns the first element and removes it from the path.

Table 3-21: MAMA API – Member Functions of the Class swAddressList

3.5.2.5. Class swObjectPath

This class deals with a single address line. This line can identify an application object by its identifier (middleware) or by the path (management hierarchy). The major operation of this class is to decrement an object path while an operation call is forwarded via application objects that are in the role of managers or agents. The general mechanism of addressing application objects is explained in section 3.4.3.

Member Function	Arguments	Description
decrementPath	-	Returns the first element of the object path and removes it.

Table 3-22: MAMA API – Member Functions of the Class swObjectPath

3.5.2.6. Class swError

This class handles all errors that occur at runtime. Each object must specify the policy for error handling while initializing. Error messages are managed similar to events. The class *swError* stores all errors locally in a map. This map is a sorted list, whereas the sort criterion is the time of occurrence of the error and the sort order is ascending.

Member Function	Arguments	Description
setServerName	string	Set the name of the (event) server errors should be sent to.
setTransmitMode	boolean	When the argument is *true*, all errors are sent immediately.
newError	error list	Generates a new error message.
lastError	-	Returns the last occurred error.
listErrors	-	Returns a list of all errors in the local error map.
showErrors	-	Prints all errors on the standard output device.
showLastError	-	Prints the last error on the standard output device.
sendLastError	boolean	Send the last occurred error to the event server.

Table 3-23: MAMA API – Member Functions of the Class swError

3.5.3. The Middleware Library

The decoupling of the core object from the middleware is already achieved by the API specifications. However, most internal parts of the API need also to be kept middleware independent. The general handling of incoming operation calls e.g. is not bound to middleware, but to the MAMA protocol. Therefore, the API includes some libraries that are responsible for the handling of middleware specific mechanisms like creation of interface objects, configuration of servers, and registration with basic services.

3.5.3.1. Class CORBA Server

The CORBA server comprises CORBA specific functionality that is dedicated to a server object. A server must create interface objects. A server must forward operation calls to the implementations of the core object. And a server provides general functions for monitoring, start, and shutdown. Furthermore, the server represents the application itself. It can take over the tasks of command line parsing, version and usage output, and naming service handling.

Member Function	Arguments	Description
shutdown	-	Stop the core object, cleanup all resources, and terminate.
visible	-	Activate and deactivate the server's standard output device.
checkLocalExecution	argument list	Check local execution policy.
checkForwardExecution	argument list	Check forwarded execution policy.
ForwardExecution	argument list	Forward the execution regarding forward execution policy.

Table 3-24: MAMA API – Member Functions of the Class CORBA Server

3.5.3.2. Class CORBA

The CORBA library defines an object that acts independent of the actual application objects. This object comprises all CORBA specific function calls a client application object requires. These function calls are representation for complex CORBA mechanisms, like registration on naming service, object address retrieval, and the actual function call on server objects.

Member Function	Arguments	Description
Initiate		Bind the given CORBA object pointer.
Action	argument list, return list	Request an action on other objects. Check the argument list for the action, parameters, and options.

Table 3-25: MAMA API – Member Functions of the Class CORBA

3.6. Application Services

The application services represent definitions and specifications to realize requirements of the general framework (cf. section 2.3.2) and rules of the Service Plane of the Conceptual Model (cf. section 2.3.3.3). The requirements depicted by the application services are: naming and addressing and registration of objects, discovery and lookup services, message services, and visualization. The rules already identify the four services that required for MAMA by means of naming, event, repository, and visualization services. The following subsections specify the application services that are needed to provide a MAMA execution environment.

3.6.1. Directory Naming and Specification Service

The Directory Naming and Specification Service (DNSS) is a combination of the three most important services of distributed applications. A directory represents the most favored mechanism of storing structured data in form of tree hierarchies. A naming service is the foundation of every distributed application because it provides unambiguously identification of application objects. A specification service realizes the virtual database of a management system, also called a Management Information Base (MIB). The DNSS has the following objectives:

1. Define a naming convention for MAMA. This naming convention must specify a set of rules for syntax and semantics of names for MAMA objects. The rules must not be technology-specific. This means, the naming conventions of a specific middleware or management system are sufficient.
2. Provide an association between names and MAMA application object. This association must enable to locate an application object based on its name.
3. Specify a directory for structured information of a MAMA system. This includes mechanisms for storing the directory structure in databases or with other appropriate technology. The directory itself should be distributed in order to address requirements such as scalability.
4. Identify a structure for storing and processing specification information. This information is defined in ADL or xADL. This is at least the MAMA Core Model.
5. Provide a relationship between the information about object classes (specification) with the information about object instances (directory). An application should be able to retrieve information about all known instances of an object class as well as the specification of a particular object instance.
6. Ensure the persistence of the information maintained by the DNSS. When the server crashes or terminates by accident, the registrations of objects and their specifications must not be lost. A uniform format must be developed to store the data and to exchange information between clients and the DNSS. Exchange formats for both specification and directory data are needed.

The combination of the three services solves several problems in a one-stop-shopping. It offers distributed applications mechanisms to search and interoperate. An application can register its services with the DNSS in order to be found by applications that profit from those services. The DNSS operates as a mediator between supplier and demander of services. Furthermore, the DNSS takes over the responsibility of a database storing information on application objects. DNSS permits applications to populate the database by registering and manipulating the stored information through uniform interfaces.

The DNSS can be compared to Yellow Pages (YP). Applications can be searched according to the services their offer or by their addresses. The DNSS offers the information that guides the distributed applications to select the right service and get its address. Additionally, the DNSS serves as a single point of contact for search, filter, and scoping operations. An application can request specific information applying filters for particular characteristics and behavior, or defining the scope of services it searches for.

The first three objectives define the basic functionality of the DNSS. The objectives four and five focus on basic features of an implementation. The following sections discuss the objectives step-by-step, in order to present the specification of the DNSS. This discussion starts with the naming conventions, which is followed by the general DNSS model (integration of directory and specification issues). The DNSS model provides the basis to derive the directory model and the specification model.

3.6.1.1. Terminology

A name is assigned to an object by following the naming syntax called the **naming convention**. The association of a name with an object is called a **binding**. Objects are not placed directly inside of the naming service. They are referenced by a **pointer** or a (object) **reference**. A **context** is a set of name-to-object bindings. This is an object with a state that represents bindings with distinct names. A context has an associated naming convention. A **sub context** can be created when a name of one context is bound to another context with the same naming convention. A **naming system** is a connected set of contexts of the same type. A **name space** is a set of names in a naming system.

A **directory** is an application that manages a set of inter-connected directory objects and provides directory services. A **directory service** offers functionality for creation, addition, remove, and modification of

attributes that are associated with directory objects. A **directory object** represents a logical or physical object. Directory objects contain attributes that describe the represented object. An **attribute** has an attribute *identifier* and a set of attribute *values*. The identifier is a token that identifies an attribute independent of its values. Directory objects are arranged in a structure named the Directory Information Tree (DIT). The structure is flat or hierarchical a tree.

3.6.1.2. Naming Convention

The naming convention covers two issues. The first issue focuses on the identification of object instances of a distributed system. The second issue deals with the processing of additional information in form of specification data.

The different kinds of middleware supported by MAMA include different kinds of naming schemes and conventions. To support each type of middleware, the naming conventions of the DNSS must be able to cover all naming schemes. Each naming scheme has tendency towards the provision of mnemonic names. A descriptive name tells about the characteristics of the object it addresses. Based on these assumptions, the DNSS naming convention defines the following syntax rules:

- Names can be written in Unicode Standard character set.
- Every name starts with an alphanumeric character. Names are case-sensitive.
- A name can be constructed from any character as shown in Appendix B on the left side of Table B-1 and all characters shown in Table B-4. Special symbols or key symbols including white spaces are disallowed.
- A name can be of any length. However, it is recommended to use not more than 256 characters.

The concept of Distinguished Names (DN) and Relative Distinguished Names (RDN) is used for the concatenation of name parts to names. An RDN is a general name, which is used to identify an entity within a namespace. The RDN of an entity is simply called the name of the entity. The DN is the prefix of an RDN in order to identify entities in the entire DNSS. The DN is also a composite name. It can span (multiple) name spaces. The interpretation of a composite name is done from left to right. The last part is the RDN of the addressed entity. The slash character '/' is used to separate name parts of a DN.

3.6.1.3. DNSS Model

The DNSS distinguishes between context information related to an object and information regarding an object. These types of information are maintained separately. In order to provide clients with a transparent and unified access, the DNSS is specified in form of three different models. The **DNSS model** realizes the access to the DNSS and defines the functionality accessible by client. The **directory model** handles context related information of objects; that is the name of an object instance. Additionally, it provides directory services. The **specification model** manages the specifications of objects.

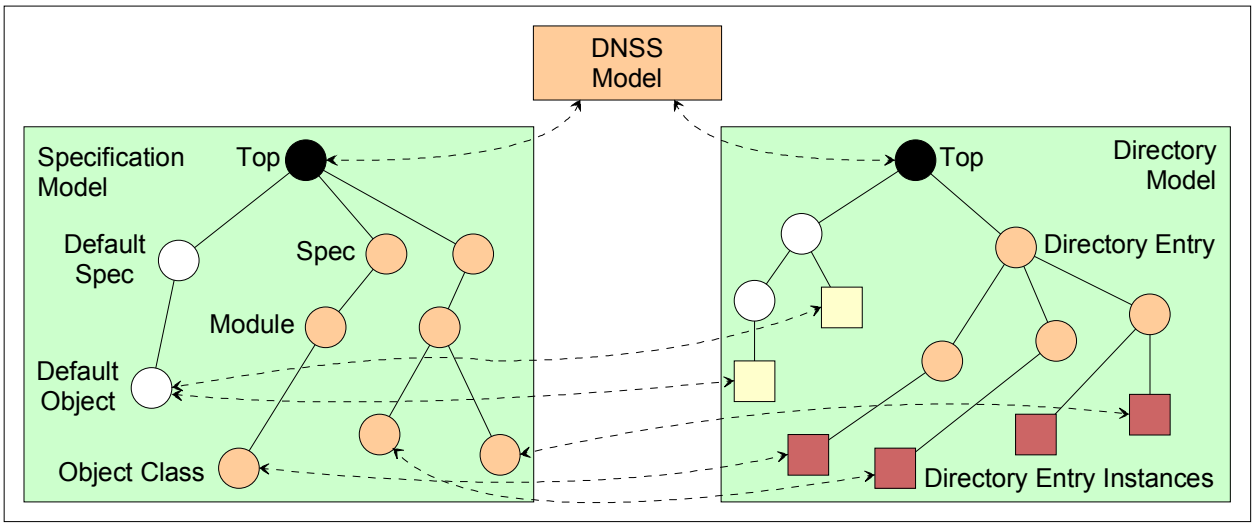

Figure 3-17: DNSS –Three Model Approach [Singh01]

The DNSS model has a bi-directional relationship to the two other models, which are only visible inside of the DNSS. Figure 3-17 shows this three model approach. The DNSS provides all functionality that is realized in the directory model and the specification model. On request of client applications, the DNSS model checks where the requested information is located and which model supports the incoming request.

This mechanism can be compared to referrals (cf. section 2.3.4.6). When the directory model needs information from the specification model, it requests this information via internal interfaces, and vice versa. Additionally, the information objects stored in the two models maintain references to each other.

3.6.1.4. Directory Model

The directory model specifies rules for the naming service part of the DNSS. The purpose of the directory model is to build naming contexts and binding instances of objects. It allows a client to create specific naming contexts according to the semantics of the client application. The model does not follow the approaches of an X.500 directory or an SNMP MIB, which bind instances of objects directly with their definitions. This approach limits the logical distribution of object instances modeling real world structures. In the DNSS, an instance of a printer does not belong to the definition of a printer, but e.g. to a certain floor of a building and to an organizational unit.

The directory model uses a DIT to process information about object instances. Predefined search operations like breadth-first, backtracking, and greedy can be employed in this structure. Figure 3-18 shows the DIT of the directory model. It groups several naming contexts under one root node.

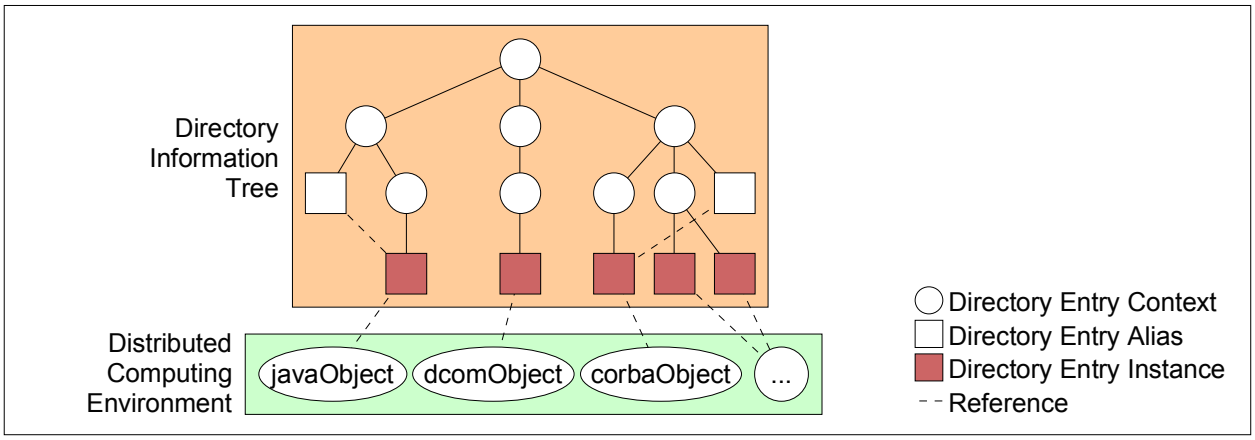

Figure 3-18: DNSS – Directory Information Tree [Singh01]

The basic building block of the DNSS directory is a **directory entry**. Each directory entry can be identified by its DN. Two consecutive directory entries maintain a bi-directional parent-child relationship. This allows traversing the DIT vertically.

A directory entry represents a node in the DIT and a name space in a naming tree. A directory entry may contain zero or more sub-directory entries as nodes and leaves. A directory entry is considered as node, even when it contains no further references to other nodes or leaves. When a directory entry is removed, the sub-entries cannot be rearranged within the DIT. They are also removed. A directory entry is created when a client application requests the registration of an instance of an object with a composite name. It can be removed explicitly by the client or implicitly a garbage collector.

A **directory entry instance** is a registered object in the DIT. It represents a leaf. Such an instance does not store the actual object; it stores only a logical image in form of collected information. The instance consists of a number of attributes:

- The *name* attribute contains the RDN of the object instance. This is the same as the name a client application has supplied with the registration of this object instance.
- The *distinguished name* stores the DN through which the directory instances can be identified.
- The *parent directory* attribute is a reference to the parent node of the directory entry instance.

- The *uuid* attribute registers the UUID of the client application with the directory entry instance. This attribute serves for security to allow changes of the directory only to authorized client applications.
- The *corresponding object reference* attribute stores a reference to the definition (specification) of the corresponding object class. This reference is a DN.
- The *instance reference* attribute contains the middleware specific address of the object instance.
- The *reference type* attribute specifies the type of an instance reference.
- The *state* attribute declares the current state of the directory entry instance. This state can be directly influenced by the client application that is responsible for the directory entry. This attribute can be used to activate and deactivate objects in a logical way.

The **directory entry alias** is a construct that allows registering object instances more than once in different naming contexts, without replicating the actual registration. An alias comprises its name, a DN, a UUID, and a reference to its parent. One additional attribute contains a reference of an instance entry from which it is the alias. An alias entry is allowed to have a one-to-one pointer to an entry instance. The value of the UUID attribute must not be the same as of the aliased entry instance. An alias entry is automatically removed when its corresponding entry instance is removed.

Directory Service Use Cases

The use cases shown in Figure 3-19 identify the functionality of the directory service. Two actors are recognized. A **service provider** registers object instances. It is the owner of a registered entry. Identification of ownership is done by a UUID attribute of the directory entry instance.

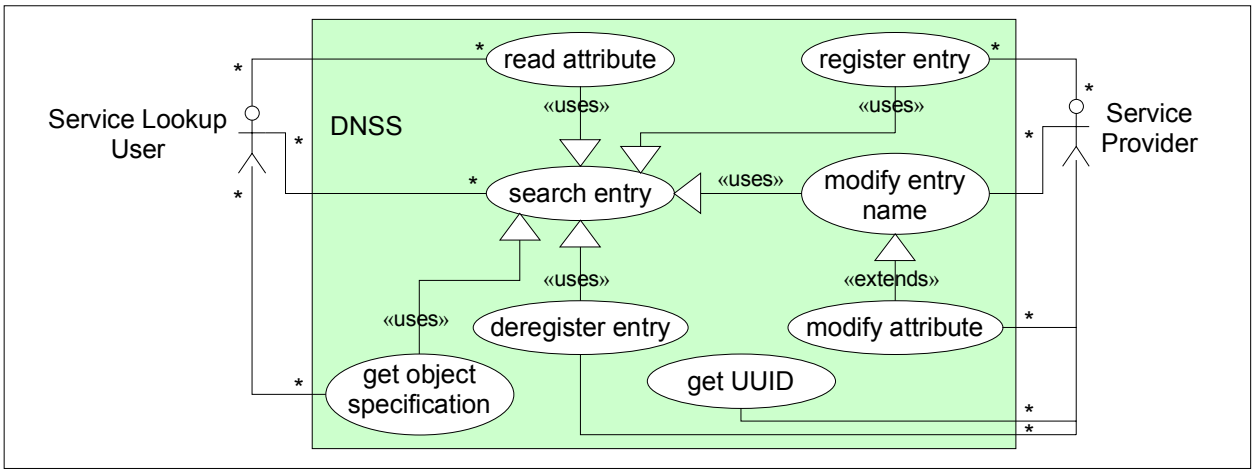

Figure 3-19: DNSS – Directory Service Use Cases [Singh01]

The **service lookup user** has the ability to lookup, search, and filter information from the directory. The resulting use cases can be classified as *directory interrogation requests* and *directory modification requests*. Interrogations are public operations of the DNSS. Access to private attributes (such as the security relevant UUID) is not granted. Interrogations are:

- *read attribute* – retrieve the values of attributes associated with a directory entry.
- *search entry* – search for directory entries according to given conditions, i.e. as a specific name.
- *get object specification* – returns the all specification information for the given element.

Modifications are only offered to actors that belong to the service provider role. They are used to change information of the directory and its entries:

- *register entry* – registers new directory entry instances or aliases in the directory.
- *deregister entry* – deregisters and removes existing entries in the directory.
- *modify entry name* / *modify attribute* – changes the state of directory entries by changing the values of attributes, e.g. set, activate, and deactivate requests.
- *get UUID* – assigns a UUID to a service provider.

All modifications must pass a security check that evaluates the UUID of the service provider with the UUIDs of directory entry instances. Valid UUIDs can only be created and are maintained by the DNSS.

Directory Service Class Design

The class design represents the formal description of the rules and semantics of the directory model. All entries in the directory are symbolized as directory entries. An abstract class *directoryEntryAbstract* is introduced as the base class for all directory entries. All other classes of directory entries inherit their characteristics from this class, as depicted by Figure 3-20.

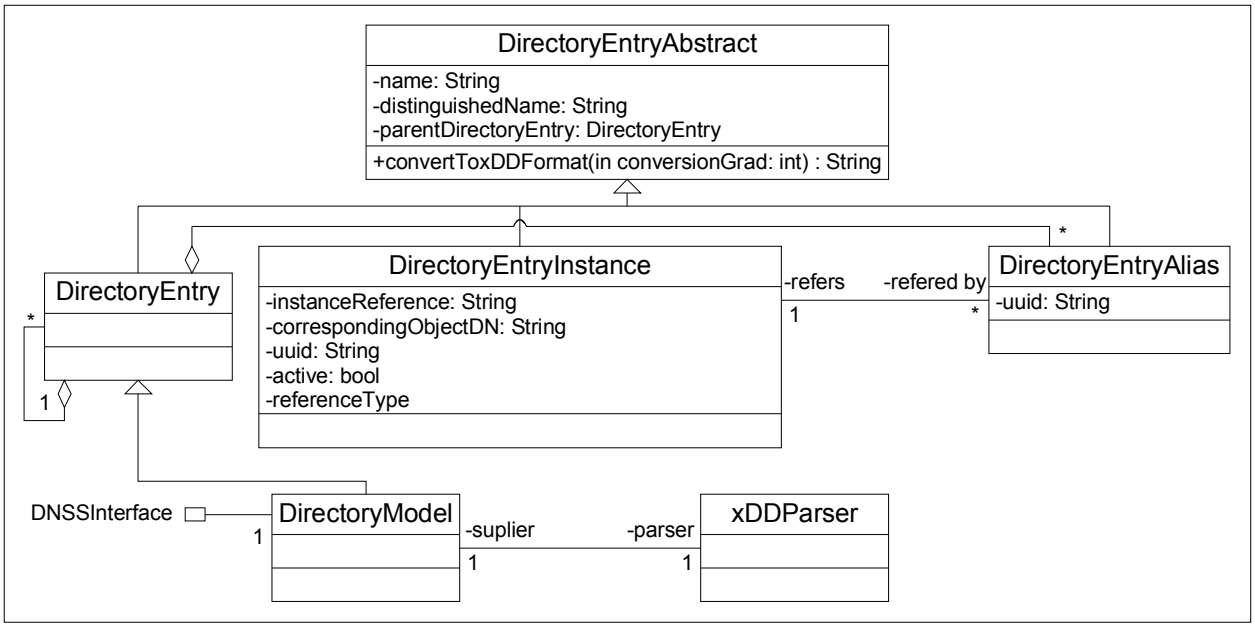

Figure 3-20: DNSS – Directory Service Class Diagram [Singh01]

The class *DirectoryModel* has the same characteristics as *DirectoryEntry*. It can contain zero or more directory entries of any type. The *DirectoryModel* is also associated to *XDDParser* to model the relationship of an external parser for the xDD format, which is discussed in the next subsection. This parser is responsible for the generation of the information in the directory model. The class diagram also specifies the *DirectoryInterface* that provides access to the directory functionality of the DNSS.

Directory Exchange Format – xDD

The eXchange Directory Definition (xDD) defines the format that is used to exchange DNSS directory data. The mapping between the directory specification (class diagram in Figure 3-20) and xDD is one-to-one. This means, the xDD format is a notation to express directories that follow the directory model.

xDD Element	Attributes	Elements
collection	-	directoryEntryInstance, directoryEntryAlias, directoryEntry
directoryEntry	name, distinguished_name	directoryEntryInstance, directoryEntryAlias, directoryEntry
directoryEntryInstance	name, distinguished_name, state, reference, reference_type, object_distinguished_name, uuid	-
directoryEntryAlias	name, distinguished_name, entry_instance_reference, uuid	-

Table 3-26: DNSS – eXchange Directory Definition

xDD is an XML based format. It is specified in form of a DTD. Table 3-26 shows each defined element with its attributes and sub elements. The complete DTD can be found in Appendix C.4.1.1. xDD information starts with the element *collection*. This element does not contain any attribute and cannot be created as object in the DIT. It functions as container for all other elements. A collection can store the entire DIT or parts of the directory tree.

The other elements contain the attributes as specified in the directory model and in the directory class diagram. The element *DirectoryEntry* is used as a container for directory information. The elements *directoryEntryInstance* and *directoryEntryAlias* do not contain sub-elements.

3.6.1.5. Directory Service Interface Specifications

The operations of the directory interface of the DNSS are derived from the use cases. Clients can retrieve and manipulate directory data with these operations. Most of the operations must specify which kind of directory data they want to be processed. Therefore, directory entries are assigned with constant values. A directory entry is associated to *100*, a directory entry instance to *101*, and a directory entry alias to *102*. Table 3-27 lists all operations of the directory interface. The complete ADL specification of this interface can be found in Appendix C.4.1.2.

The operation *getObjectSpec* represents the link to the object specification. It allows retrieving meta information of an object class from the specification model for each known object instance. Parameters are the DN of an object instance and a flag that identifies the format the information should be returned in.

Operation	Major Parameters	Description
deregister	entryDN, entryType	Remove an instance entry or alias entry from the DIT.
getAll	-	The entire DIT is returned (xDD string).
getAttributeValue	entryDN, entryType, attributeName	Return value of any attribute. Return type is a string.
getCount	parentDN, entryType	Return number of entries of a given *entryType*, which are bound to the given node.
getEntries	parentDN, entryType fromIndex, toIndex	This operation returns a list of entries (xDD) of the given *entryType*, which are assigned to the given node.
getEntry	entryDN, entryType	This operation returns an entry with the given DN.
getInstanceIOR	entryDN	Return the middleware-specific object reference of a directory entry instance.
getObjectSpec	instanceDN, formatType	Returns the specification (specification model) of a given instance (directory model) in ADL or xADL.
getUUID	-	A new UUID is created and returned as string.
modifyEntryName	oldEntryDN entryType newEntryDN	Change the position and the RDN of an entry in the DIT. Precondition is that *newEntryDN* is unique in the directory. Return type is a Boolean.
register	instanceDN reference referenceKind objectDN	This operation registers an object instance. The RDN must not be unique. When *ObjectDN* is not found in the specification model, the instance entry refers automatically to the *default* object. Return type is a Boolean.
registerAlias	aliasDN, instanceDN	Create an alias entry.
setAttributeValue	entryDN, entryType attributeName attributeValue	This operation modifies the content of an attribute. An attribute of an entry with given values can be assigned. Return type is a Boolean.

Table 3-27: DNSS – Directory Service Interface Operations

The input parameters follow a given syntax and semantic. Parameters that are handled as distinguished name are constructed out of a string that identifies the directory class they describe (entry, instance, alias) and the acronym DN. The parameters of the interface operations are interpreted as follows:

- *attributeName* is the name of an attribute of a directory entry class.
- *attributeValue* the value of an attribute which is expected by the operation. The value is always transmitted in form of a string, even when the attribute is of a different type.
- *entryDN* is a string interpreted as the DN of the entry.
- *entryType* is an integer that identifies the entry class according to the definitions of section 3.3.3.4.
- *fromIndex* and *toIndex* indicate that the operation can generate a list as result. Both parameters can be used to specify start and end points of the list. For example, the generated list is {2,8,3,8,6,3,9,0}, fromIndex is *3*, and toIndex is *6* than the returned list is {8,6,3,9}.
- *parentDN* is a string with the DN of the parent node of an entry.
- *recursive* is a Boolean flag. When set to *true*, the operation should be invoked recursively from the starting node up to the bottom of the tree.
- *reference* represents the middleware-specific reference to the object instance, e.g. a CORBA Interoperable Object Reference (IOR).
- *referenceKind* shows the type of reference which has originally been provided during the registration of the object instance.
- *uuid* is a string interpreted as the UUID of the client application.

Boolean return values indicate a successful processing on *true* and any error on *false*. For the retrieval of errors, notifications are used.

3.6.1.6. Specification Model

The specification model handles schemas. Its functionality is similar to a MIB. xADL is used in the specification model to describe specification objects. Beside the characteristics that xADL directly inherits from ADL, the attributes *uuid* and *distinguished_name* are introduced to store and process specification information. The UUID serves for security in the same way as already discussed in the directory model.

The *distinguished_name* is used to identify specification objects with in the model. The relationships between an entry in the directory model (object instance) and in the specification model are transparent to client applications. This information is assigned to each xADL element of type object in form of a list. When an object in the specification model is removed, all references of objects in the directory model are to be changed to point to a *default* specification object. In case a new object instance is registered without the provision of specification information, this object points to the *default* object, too.

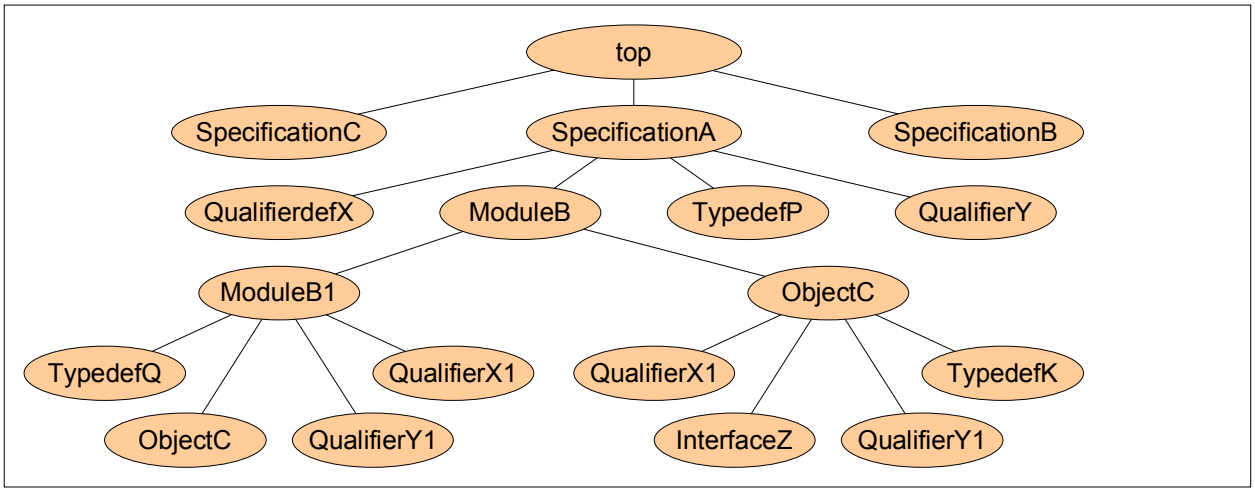

Figure 3-21: DNSS – Specification Information Tree

The specifications are stored in a tree structure. Figure 3-21 shows an example Specification Information Tree (SIT). All specification elements are represented as nodes in the SIT. Two consecutive nodes have a parent-child relationship in form of a containment relationship. Every specification element is accompanied with a pointer to its parent node. The root of an SIT represents always the element *top*. All specification nodes are arranged below *top*.

The example SIT shows three specifications. For *SpecificationA*, only modules, objects, and interfaces are depicted. Each element has its own DN and RDN. An RDN may exist more than once in the SIT, as long as the DNs of the elements are not the same. In the given example, the RDN *ObjectC* is used twice. The DN of the first occurrence of this RDN is */SpecificationA/ModuleB/ObjectC*, the DN of the second occurrence is */SpecificationA/ModuleB/ModuleB1/ObjectC*.

Specification Service Use Cases

The use cases shown in Figure 3-22 identify the functionality of the specification service. Two actors are recognized. A **service provider** adds specification objects. It is the owner of each specification element. Identification of ownership is done by the UUID attribute of the specification element.

The **service lookup user** has the ability to lookup, search, and filter information from the specification model. The resulting use cases can be classified as *specification interrogation requests* and *specification modification requests*. Interrogations are public operations of the DNSS. Access to private attributes (such as the security relevant UUID) is not granted. Interrogations are:

- *read attribute* – retrieve the values of attributes associated with a specification element.
- *search entry* – search for specification elements according to given conditions, such as a specific name or value of an attribute.
- *get object instance* – returns all information about an object instance.

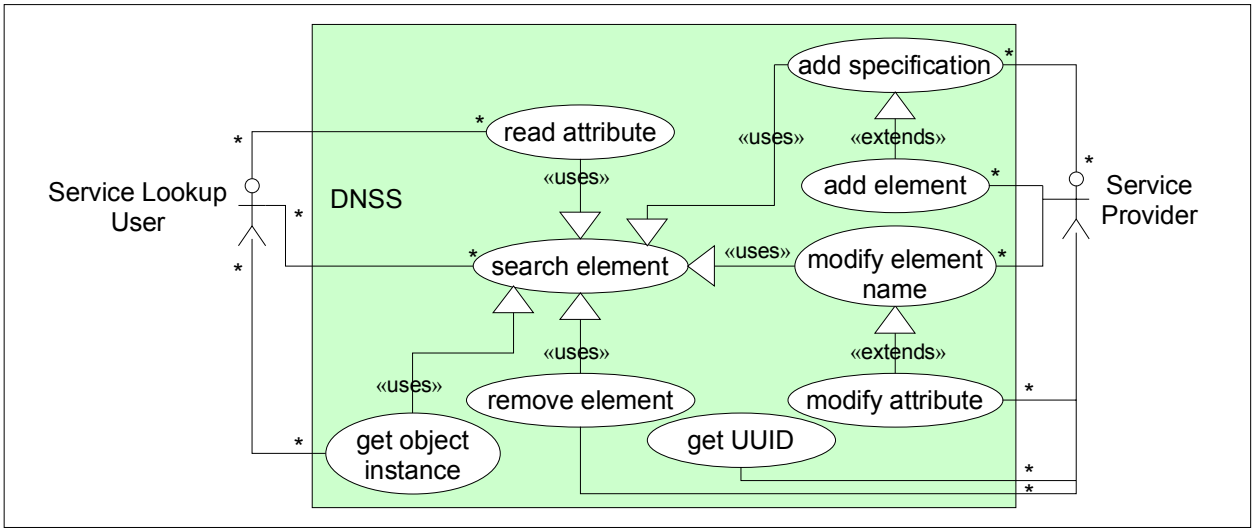

Figure 3-22: DNSS – Specification Service Use Cases [Singh01]

Modifications are only offered to actors that belong to the service provider role. They are used to change information of the SIT:

- *add specification* / *add element* – add an entire specification or a specific element to the SIT.
- *remove specification* / *remove element* – remove an entire specification or a specification element from the SIT.
- *modify element name* / *modify attribute* – change the state of specification elements by changing the values of attributes.
- *get UUID* – assigns a UUID to a service provider.

All modifications must pass a security check that evaluates the UUID of the service provider with the UUIDs of specification elements. Valid UUIDs can only be created and are maintained by the DNSS. The *get UUID* use case is introduced to allow clients to assign a UUID to all their requests.

Specification Service Class Design

The class design represents a formal description of the rules and semantics of the specification model. It is based on the MAMA Core Model (cf. section 3.3 and Figure 3-5). Some new classes have been added to develop an object-oriented design suitable for the provision of specification services. Those classes are needed to construct an SIT.

The class diagram in Figure 3-23 shows that all classes inherit their basic characteristics from the abstract base class *SpecificationElementAbstract*. Therefore, every specification element includes a name, a reference to the parent specification, and a distinguished name.

The class *SpecificationModel* is not part of the MAMA Core Model and has no association to qualifiers. This difference is modeled with the separation of the classes *SpecificationModel* and *SpecificationElement*. The class *SpecificationEntryAbstract* serves as abstract base class for the ADL element module (*SpecificationEntryModule*), object (*SpecificationEntryObject*), and interface (*SpecificationEntryInterface*). Those elements are realized as container classes in the specification model.

All typed ADL elements (attribute, action, parameter, and member) inherit their basic characteristics from the class *SpecificationTypeElementAbstract*. All those elements can be defined as arrays. They can have the characteristic signed or unsigned. Additionally, they are defined either as a base type (an ADL base type) or as a new introduced type (a prior type definition). An attribute with a plural name is a collection attribute. This means, it contains zero or more values and its attribute type can be chosen as a list or a vector.

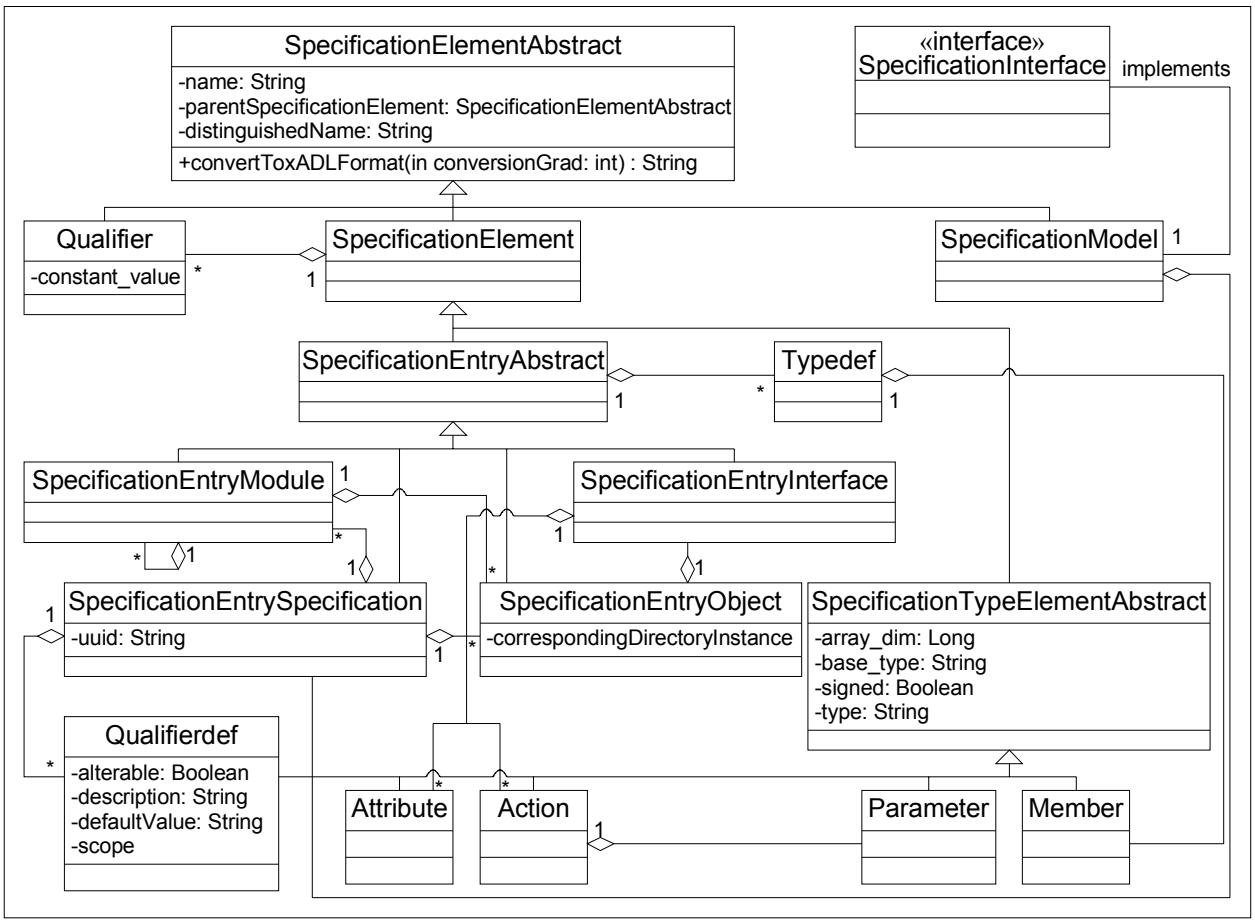

Figure 3-23: DNSS – Specification Service Class Diagram [Singh01]

The operation *convertToxADLFormat*, inherited by all classes, specifies that each class must realize a conversion to xADL. How this conversion is done is up to the implementation.

3.6.1.7. Specification Service Interface Specification

The operations of the specification interface are derived from the use cases. Clients can retrieve and manipulate information of the SIT with the operations of this interface. Table 3-28 lists all operations of this interface along with their major parameters and a short description. The complete ADL specification of this interface can be found in Appendix C.4.1.2.

The operation *getInstances* represents the link to the object instances. This operation allows retrieving information about all object instances from the directory model for each known object class. The operation expects the DN of an object class of the SIT. The returned list can be further restricted.

Operation	Major Parameters	Description
addSpecification	specName	Adds a new specification to the SIT.
addToSpecification	newElement elementType parentDN	This operation adds the given element to the given node in the SIT. The additional parameter *uuid* is used for security checks.
getAll	-	All specifications are returned as string.
getElement	elementDN elementType	An element can be searched in the SIT. The returned string can be compact or complete, in ADL or xADL.
getCount	elementDN elementType	This operation returns the number of elements of *elementType*, which are bound to *elementDN*.
getElements	parentDN elementType fromIndex, toIndex	This operation returns a list of the elements of the given type from the given node. The format of the list can be compact or complete, ADL or xADL.
getElementsByValue	attributeName attributeValue elementType fromIndex, toIndex	All elements are returned, which have same attribute value as the given attribute value. The length of the list can be further restricted. The returned string can be compact or complete, in ADL or xADL.
getInstances	objectDN, fromIndex, toIndex	Returns all registered instances (directory model) of the given object class (specification model) in xDD format.
modifyElementName	oldElementDN elementType newElementDN	The position and RDN of a specification element in the SIT is changed. Precondition is that *newElementDN* is unique in the SIT.
remove	elementDN elementType	All kinds of specification elements (including their complete specification) are removed.

Table 3-28: DNSS – Specification Service Interface Operations

The operations *getAttributeValue*, *getUUID*, and *setAttributeValue* are the same as in the directory interface specification. They are not listed in Table 3-28.

The input parameters follow a given syntax and semantic. Parameters that are handled as distinguished name are include the acronym DN. The parameters of the interface operations are interpreted as follows:

- *elementType* – specifies which ADL element is of interest. All ADL elements are associated to constant integer values as defined by *tElementType* in the MAMA Core Model (cf. section 3.3.3.4).
- *elementDN* – contains the DN of the element.
- *parentDN* – contains the DN of the parent node of a specification element.

- *recursive* – is a Boolean flag for the recursive execution. When recursive value is *true* then the operation is executed from the starting node to the bottom of the tree.
- *newElement* – is a string in xADL or ADL format that contains a specification element.
- *uuid* – contains the UUID assigned to the client.
- *formatType* – specifies the format the client expects as result in form of an integer. ADL format is assigned with constant value *200* and xADL format is assigned with constant value *201*, as defined by *tSpecLanguage* in the MAMA Core Model (cf. section 3.3.3.4).
- *compact* – refers to the alternative to return specification information in a compact form (*false*) or in the complete form (*true*). The compact form does only include specification elements and their mandatory attributes. All elements will be empty; they do not contain any sub-element. E.g., a module element will not contain any sub-modules or objects. The complex form includes all requested and available specification data.
- *attributeName* – is the name of an attribute, which belongs to any class shown in the specification class diagram in Figure 3-23.
- *attributeValue* – indicates that the operation expects the value of an attribute.
- *fromIndex* and *toIndex* indicate that the operation can generate a list as result. Both parameters can be used to specify start and end points of the list. For example, the generated list is {3,6,2,6,3,2}, fromIndex is *1*, and toIndex is *4* than the returned list is {6,2,6}.

Boolean return values indicate a successful processing on *true* and any error on *false*. For the retrieval of errors, notifications are used.

3.6.1.8. Security

The DNSS is supplied with a minimum of security. The objective is to disallow non-authorized modifications of the DIT and the SIT. Furthermore, access to private attributes is not granted. All other information of the DNSS can be accessed by clients without special permissions.

According to the DNSS design, every specification element, directory instance, and directory alias stores the UUID of its owner. Access for modifications is only granted, when the UUID supplied by the client along with the modification request is identical to the UUID of the entry or element. The attribute *uuid* can only be accessed by components of the DNSS system. The implementation of the DNSS is responsible to realize this behavior.

3.6.1.9. Distributed DNSS

The DNSS is developed as a stand-alone service that supports applications within a domain. To enable the support of inter-domain communication of applications, the DNSS concept has been enhanced with mechanisms that allow a distributed cluster of DNSS servers. The mechanisms of this concept are a combination of the replication approach of the Lightweight Directory Access Protocol (LDAP; [IETF-RFC2251]) and the multi master replication approach of the Active Directory (AD; [MS-ADArch]).

When more than one DNSS server is available, one of these servers operates in the role of a master DNSS server. The master DNSS server manages complete replications from other DNSS servers on request, not by default. DNSS servers that are going to shutdown can transfer their data to the master DNSS server. This avoids redundancies in the maintained information and extra data management.

A client communicates with one DNSS server. The communication between DNSS servers is transparent for the client. Client requests that cannot be processed by its dedicated DNSS server are forwarded the master DNSS server. The master DNSS server is responsible for either processing the request or return the reference of a DNSS server that is in the position to process the request.

Figure 3-24 shows the distributed processing of DNSS functions in form of a sequence diagram. The figure comprises a client, two DNSS servers, and the master DNSS server of an exemplary distributed system. A client request results in the following steps.

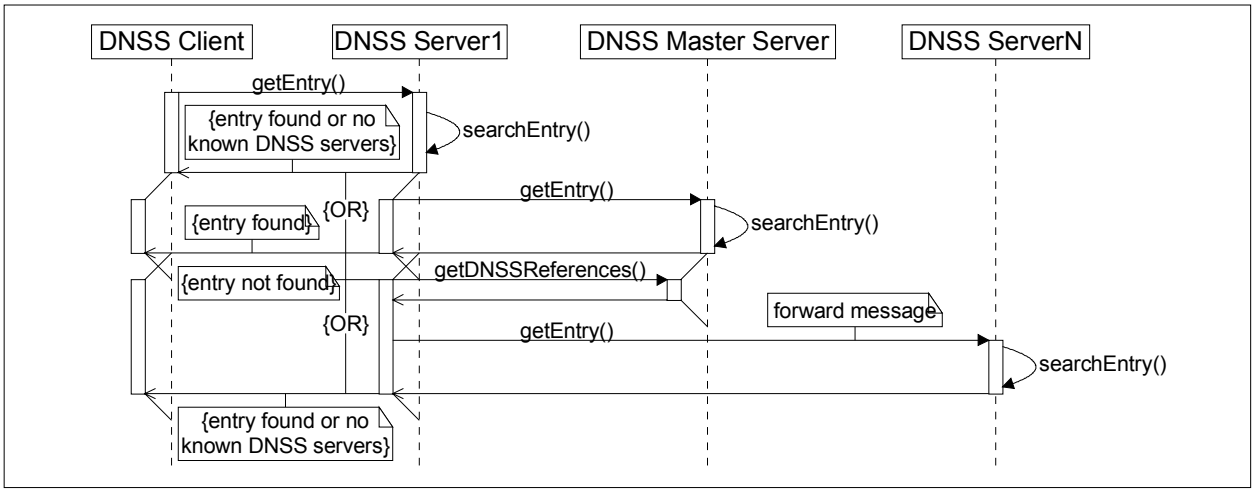

Figure 3-24: DNSS – Distributed DNSS

1. The client requests its dedicated DNSS server to search for a directory entry.
2. When the entry was found, it is returned. When the entry was not found, the request is forwarded to the master DNSS server. When no relationship exists between the DNSS server and the master DNSS server, the DNSS server notifies the client that the entry was not found.
3. The master DNSS server receives the request and searches in its directory model for the entry. When the entry was found, it returns the result to the DNSS server which returns it to the client. When the entry was not found the master DNSS server notifies the DNSS server about this event.
4. The worst case is that the searched entry was neither found at the DNSS server nor at the master DNSS server. In this case, the DNSS server requests references of other known DNSS servers from the master DNSS server. The result is a list with references. The DNSS server can now request each DNSS server of the list for the searched entry. The transaction with the client terminates immediately when the entry was found.

When a DNSS server is started, the reference of the master DNSS server must be supplied as argument. Right after startup, the DNSS server must register with the master DNSS server. When no reference is supplied, the started DNSS server can act as a master DNSS server itself.

3.6.1.10. Requirements on Clients

An application is considered as a client of the DNSS when it calls operations provided by the DNSS. A client can be categorized as an application that consumes services of the DNSS, an application that mediates between another application and the DNSS, or another DNSS system that communicates with the DNSS.

The DNSS is a MAMA application service. It fully supports the MAMA protocol. A client that wants to utilize DNSS services must match the following conditions:

- use the MAMA protocol to communicate;
- understand the XML formats xDD for directory data; and
- understand the XML format xADL or the format ADL for specification data.

When a client registers an object instance, it must supply the middleware-specific reference of the object and describe the type of reference. When this information is not supplied, no other application can connect to the registered object instance. The MAMA API is responsible for supplying this information.

Every client of the DNSS must maintain the UUID that is assigned to it. This UUID is issued by the DNSS and must be presented for modification requests. Modification requests without a UUID are not performed.

3.6.2. Visualization Service

The major task of the Visualization Service is to provide access to information about MAMA applications with a GUI. The information available for visualization can be classified as follows:

- Information about object classes –specification data in ADL or xADL format.
- Information about object instances – directory data in xDD format for naming and in ADL for the current configuration of the instances.
- Information of a predefined model – the Core Model and application-specific extensions in ADL or xADL format.
- Combination of the available information realized for example in the DNSS where specification information and directory information is related to each other.

The aim of this service is the general visualization of ADL/xADL based systems and a detailed visualization of MAMA applications. The most important ADL elements are *module* and *object*. The element *module* provides naming contexts and an *object* represents a MAMA application. All other ADL elements belong to objects and should be presented with their object. Four aspects are important for the visualization:

- Receive all information about the specification that is currently in use;
- Receive all information about object instances (names and current configuration);
- Receive all information about the Core Model; and
- Connect the three parts above to show dependencies and relationships.

The graphical representation should be based on graph visualization. This concept is used as an external representation that exploits human visual processing to reduce the cognitive load of the user tasks [Munzner00]. For the visualization of dependencies, this concept must be supported with other graphical approaches, such as content maps and WWW browsers.

The human user group for the Visualization Service consists of developers who create system parts and developers who integrate functionality into distributed applications. At a later point of time, administrators and support persons will be able to use the Visualization Service for the maintenance of a running MAMA system.

The graphical interface enables users to handle provided functionality in an easy and intuitive way. The usage of functions built into the interface has to be done on the basis of user commands. The interaction control of the system is passive (directed by the user, not by the application itself). At this time, there are no special requirements for the execution time. The following goals can be identified for Human Computer Interaction (HCI):

- navigating through the data structure (specification, instances, Core Model) and requesting the detail information;
- navigating through the data structure and requesting filtered detail information;
- connecting specification data and data instances;
- finding similar data instances; and
- finding correspondences of specification/instances data with data from the Core Model.

It is helpful to compare different design aspects in order to define the appropriate GUI. This can be done with case studies and empirical analysis' for the design of back-end and front-end components. Interesting aspects for backend components are source code reusability as well as application and device independence.

3.6.2.1. Understanding a Design

People and their relationship to workspace and environmental parameters often depend on the time they need to understand the system. A good design should reduce the skills needed to use the system. Besides the informal understanding of design issues, the error handling is an important part. Preventing users from

making errors, predicting when errors are likely to happen, and helping the users to recover from errors [John96] increases the acceptance of a system.

The developer should care for the arrangement of displays and controls, link analysis, human cognitive and sensory limits, display design, control design, fatigue and health issues, furniture and lighting design, temperature and environmental noise issues, design for stressful or hazardous environments, and design for the disabled [Hewett96]. Example systems are [Hewett96]:

- Command-oriented: PC-DOS[3] (command style interface known to millions of users)
- Graphics-oriented: Apple Macintosh (similar interface for many applications)
- Frame-based: HyperCard (Graphically-oriented frame-based system with user programming language; first mass market frame-oriented system).
- User-defined combinatorics: UNIX (strong combinatoric paired with weak human factors); Emacs (large combinatoric command set); Nintendo (learnable without a manual by school children)

Who centric? is another aspect for the design of applications. The usual approaches are user centric, document/information centric, and application centric. The distinctions between the three approaches reflect the differences for user interaction with the system. In application centric architectures, the basic unit is the file. With the introduction of graphical user interfaces and the desktop metaphor, files became direct visual objects. The objects are accessible directly by the user, storable on the desktop or in folders, and to a limited extent they can be organized by the user or application in semantically meaningful ways. However, the content of the files is still not directly accessible for the user.

With the document and information centric interface paradigm, the basic unit is no longer the file, but a document or information. The applications role is subordinated and the user can focus on direct manipulation of documents. Users can literally get their hands on their documents. Hints of this approach may be found in a few existing interfaces. For example, recent Microsoft Word supports dragging of selected text from one place in the document to another. The Macintosh system provides transparent drag-and-drop, which Netscape enables images to be dragged from web page directly onto the desktop [Roth98].

The following subsections give recommendations for the visualization of structured data. The recommendations can be used to develop GUIs the visualize MAMA systems, MAMA applications, and MAMA services. They are presented starting with the information that need to be displayed, the required user tasks, the basic user interface structure, and comments regarding the essential front-end capabilities. The Visualization Service assumes that information is available in xADL or xDD format. ADL format is not considered, since it can be easily converted into xADL.

3.6.2.2. Specification Data – Navigation and Information Display

The data that is handled here can be classified into three levels shown in Figure 3-25. The first level contains the structure of data. This level is static and predefined by the structure of xADL. The second level contains the element names. Depending on the specified element, this part is dynamic. The third level contains the huge amount of detail information. This amount data depends on the element specified.

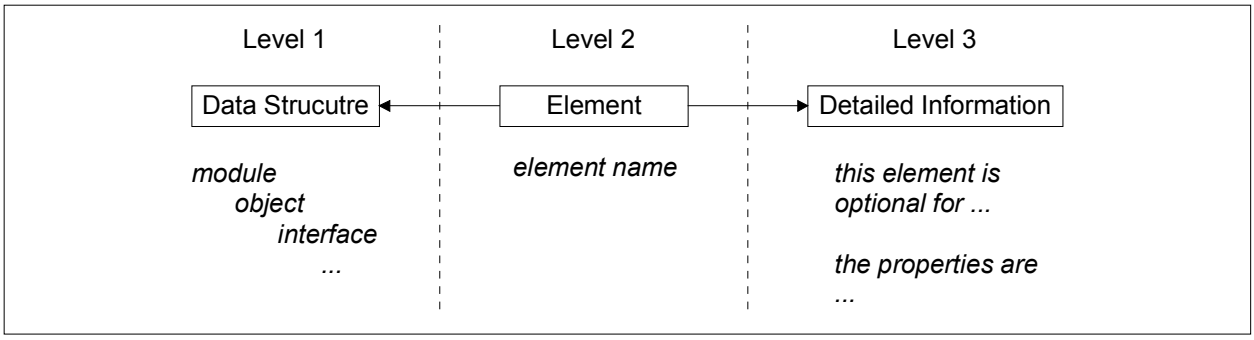

Figure 3-25: Visualization Service – Three Levels of Information

[3] Personal Computer Disc Operating System

All specification data is structured in xADL by the use of separate elements. That means, most of the data can be classified in the first and second level. Some elements, however, only characterize the parent elements and are not important for the first level. This data is located in the third level according to its descriptive character. The following, partial xADL DTD shows the important elements for the first level. Those elements are *module*, *object*, *interface*, *attribute*, *action* and *typedef*.

```
<!ELEMENT module (qualifier*, (typedef | object | module)+)>
<!ELEMENT object (qualifier*, (interface | typedef)*)>
<!ELEMENT interface (qualifier*, (attribute | action | typedef)*)>
<!ELEMENT attribute (qualifier*)>
<!ELEMENT action (qualifier*, parameter*)>
```

The elements *module* and *object* are independent of a predefined parent element. They can appear alternately as root elements. Therefore, these elements are of special interest.

User Tasks

The user can request two tasks, navigation through specification data (including detailed information) and requesting filtered detail information. Realizing the navigation through the complete structure of specification data means to display two different types of information, structure as well as actual content. Therefore, the display needs to be divided into at least two parts. This follows the assumption that it is imperative for a predictable interface to display only the information needed at a specific time without missing other parts of information.

The starting elements to access specification data are *module* and *object*. They should be used as selecting points for detail information. They are usually found in the first and second level of the data hierarchy. The source of all filtered information is modules and objects. Two tasks are necessary to receive filtered information: selecting the module or object and selecting the structural filter for this module or object.

Basic Structure of the User Interface

Figure 3-26 shows a recommendation for the user interface. It comprises three parts. The first part displays a selectable tree of the specification data. The second part includes a selectable filter for structured data. The last part focuses on the visualization of detailed information.

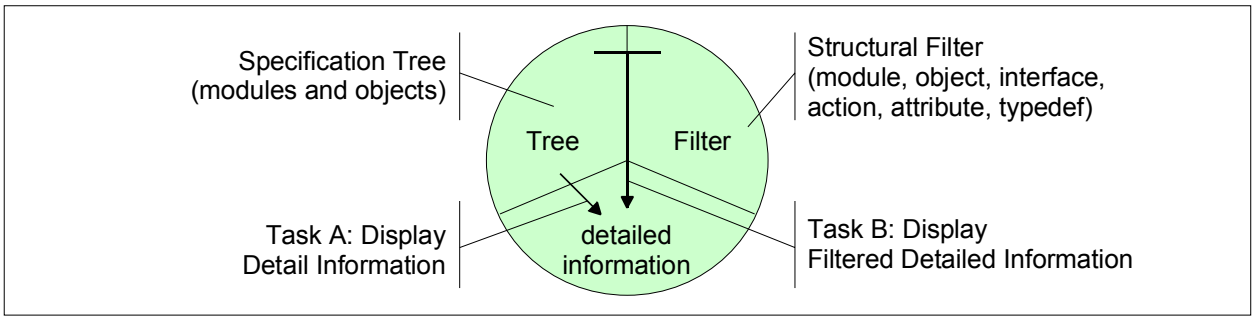

Figure 3-26: Visualization Service – Tasks for Specification Data

The parts Tree and Filter are *active*. This means, they support navigation and selection done by the user. The last part is *passive*. Its function is to display a set of data. The interaction between the three parts depends on the chosen element of the Tree part and on the Filter part.

The user interface must realize the visualization of detailed information and of filtered, detailed information. The access to information can be realized with two frames, while the third frame displays detailed information.

The recommended user interface shows close similarities to standard mail tools (Microsoft's Outlook, Netscape's Messenger, and the mail part of Opera). The Tree part matches with the rubric part, the Filter part matches with the header part, and the detail information matches with the body part. Using the pat-

tern of a standard, frame-orientated mail tool offers another advantage: Users know how to handle tools based on these structures every day, so the learning and using of this new tool will be easy.

The disadvantage of such a window-oriented visualization is the danger that the user might loose the context of information. When too much information (in form of controlling panels, icons, scroll bars, etc.) is presented, the user's cognitive capabilities are overloaded. Therefore, the recommendation includes three windows, two with selectable items and one for displaying information.

By using file system metaphors, the Tree part can be realized in form of a simple tree structure. The element *module* is similar to folders. A module can be presented as an expandable tree node, which contains other modules and objects. The element *object* is similar to a file. Access, expansion, and selection within this tree can be done using any pointing device. The tree should be realized as a vertical frame following the pattern of a standard mail tool.

The access to structural filter information can also follow the pattern of a mail tool. The frame at the top of the user interface can be small, relatively to the dimension of the complete GUI. This frame should comprise the Filter part, which can be realized as a static tree or an arrangement of simple buttons. The elements that should be considered for the filter are *module*, *object*, *interface*, *attribute*, *action*, and *typedef*.

Detailed information (third level of information) can be arranged as the main frame (bottom-right of the GUI). This frame needs to be large, since it might contain a large amount of textual information. The information itself should be formatted and if appropriate colored.

For a better adaptation on the current display device and for means of personalization, it is possible to adjust the display with an external style definition. The detail information display should be flexible and able to display other information as well. The following data is displayed:

- detail information about the selected element in the specification data;
- detail information about the selected element in the directory data;
- filtered detail information about selected directory or specification data;
- detail information about Core Model elements; and
- administrative content like help and preferences.

Information Filter

The filtering of the detail information is done by selecting one of the important elements. The resulting filtered information for the elements includes also basic structural information. This information represents e.g. the relationship of an action with the interface where it is specified. The following listing shows the range of data that should be displayed when a filter is active:

- general filter: module name, object name, and interface name;
- module filter: module name, module qualifier, module type definitions, object name, and interface name;
- object filter: object name and extends attribute, object qualifier, object type definitions, and interface name;
- interface filter: object name, interface name, interface qualifier, interface type definitions, attribute name and action name;
- attribute filter: object name, interface name, attribute name including the XML attributes, and attribute qualifier;
- action filter: object name, interface name, action name including the XML attributes, action parameter, and action qualifier; and
- type definition filter: module name, object name, interface name, all names of type definitions including the XML attributes, members, and qualifiers.

3.6.2.3. Directory Data Visualization

The directory data contains information on object instances. Access to specification information is granted with a pointer in form of a distinguished name. The relationship between a directory entry and a specification element can be many-to-one. The elements of the directory are defined in the xDD DTD. The following fragment shows the relevant elements for the Visualization Service:

```
<!ELEMENT directoryEntry (directoryEntry*, directoryEntryInstance*,...)>
<!ATTLIST directoryEntry
name CDATA #REQUIRED
...>
<!ELEMENT directoryEntryInstance EMPTY>
<!ATTLIST directoryEntryInstance
name CDATA #REQUIRED
...
object_distinguished_name CDATA #REQUIRED
...>
```

The goals of the visualization of directory information are:

- navigating through the structure of data instances and requesting the detail information;
- navigating through the data structure and requesting filtered detail information;
- connecting specification data and object instances; and
- finding similar data instances.

The cognitive skill viewpoint [Munzner00] offers a solution of the visualization issue with displaying and filtering of detail information. Identical information should be accessed in identical ways. According to the display of specification data, directory data should also be visualized in the same selectable tree part of the application.

User Tasks

Displaying the directory and specification tree data at the same time is not necessary. However a connection between the two trees has to be made. The change between both trees has to be connected closely. Nodes in the directory tree should be marked if they match with the last accessed node in the specification tree. A reload function should be provided, because of dynamics of the data. Additionally, the directory tree nodes with the same pointer (*object_distinguished_name*) should be marked when the data is reloaded.

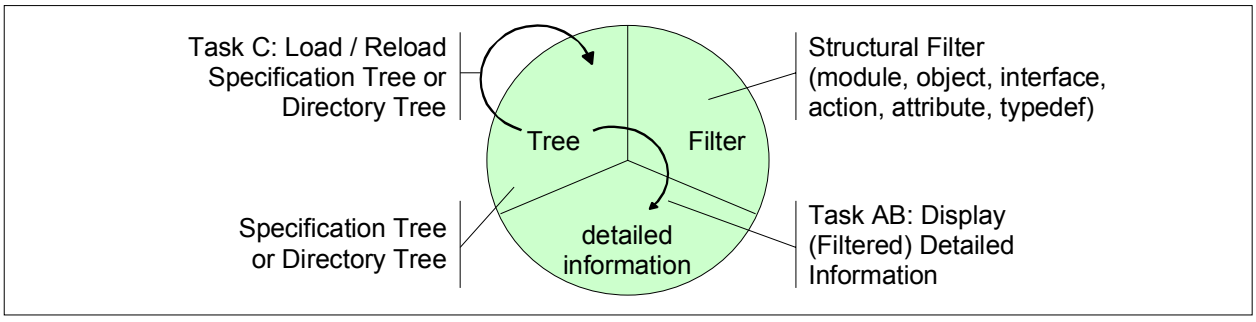

Figure 3-27: Visualization Service – Tasks for Directory Specification Data

Figure 3-27 shows the recommended parts of the application and the main user tasks. No general change of the concept is necessary. Only one user task is added in order to realize the handling of directory data. The tasks for displaying filtered or unfiltered data are the same as for the specification data. They are combined to task *AB*.

Directory Tree Front-end

The switch between the directory and specification data should be realized with two buttons. The position of these buttons is on the top of the left frame. The buttons are used for switching between the data and for reloading data. The display of both types of data in the same frame calls for different visualizations. The displayed trees need a clearly distinctive mark, so that the tree node icons must be different.

3.6.2.4. Visualization of Predefined Models

Specification data and Core Model data are both described in xADL. Important for the design process are the differences of those data descriptions.

The data structures of the specification data and the Core Model differ only regarding qualifier definitions. Those definitions can only be found in the Core Model. The relevant elements are shown in the following xADL DTD fragment:

```
<!ELEMENT specification (qualifierdef*, (typedef | object | module)+)>
<!ELEMENT qualifierdef (scope+, description)>
<!ELEMENT module (qualifier*, (typedef | object | module)+)>
<!ELEMENT object (qualifier*, (interface | typedef)*)>
<!ELEMENT interface (qualifier*, (attribute | action | typedef)*)>
<!ELEMENT attribute (qualifier*)>
<!ELEMENT action (qualifier*, parameter*)>
```

User Tasks

The tasks can be identified following the goals of the visualization of the Core Model:

- navigating through the Core Model data structure and requesting the detail information; and
- finding correspondences of specification/instances data and data from the Core Model.

Moving through the Core Model structure and displaying the detail information works in a different way than moving through the specification and directory tree. Major points of interest are the qualifier definitions and the description of every element. A separate structure of objects and modules is not in the main focus. The structural representation of Core Model data (first and second level) should be realized by using a hierarchical visualization.

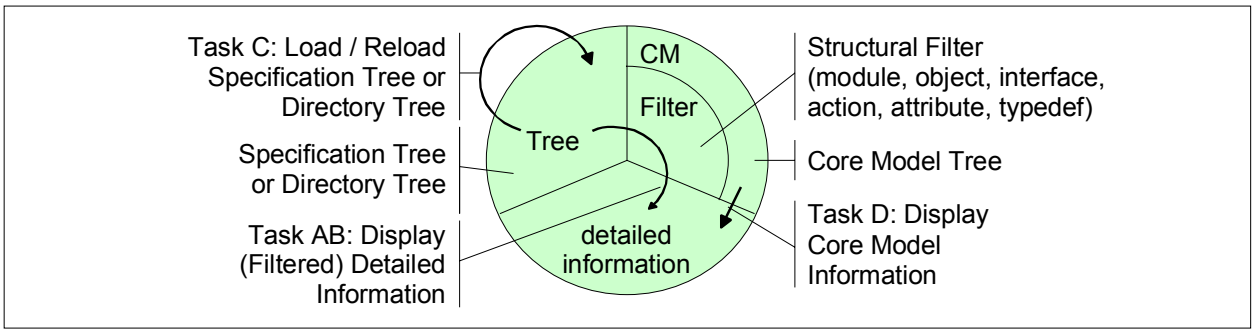

Figure 3-28: Visualization Service – Tasks for all Types of Data

The second goal is the connection of directory/specification data and Core Model data. To avoid major changes in the actual recommendation for the user interface, this connection should be done by the structure filter because its structural aspect has to be covered, too. The user needs the context information at the time of filter selection and Core Model connection.

Filter

The filter, which is a type of representing a static structure, needs to be realized in a very structural form. A simple kind of tree or any other hierarchical visualization fits this model. Additionally, the process of connecting the Core Model data has to be performed by this filter because no detailed structural informa-

tion is available in the directory or specification tree. The GUI needs to support cross connections and standard hierarchical structures.

3.6.2.5. Additional Functionality

The GUI of the visualization service should provide basic functionality like quite and help functions. Help information can be displayed in the information frame. The selection of the basic functions needs to be integrated in the general layout. It should follow common style guides.

To realize a flexible data input for the xADL and xDD data, user defined preferences can be added. They can be located in an external XML file containing the needed information. This preferences XML file can be parsed in the same way as the other files.

Execution Time

There are no particular requirements for execution time, though some aspects have to be taken into consideration. By generating strings for the detail information, all nodes have to be converted to an HTML string (or any other chosen output format). The conversion of all filter strings is usually done in one process, so the detail information is immediately available. In consequence of a fast parsing process, the XML file should be read completely at every load or reloads. The Core Model data should be read just once during the launch of the application because of its static character.

Error Handling

An error information display should be provided, at least during the startup of the GUI. The errors which could occur are *parsing errors* and *file not found errors*.

Links in Detail Information

Some detail information of the Core Model data contains hyperlinks. So, the passive behavior of the detail display has to change to an active behavior. The activation of links and the display of the linked page or information are also possible in that frame. Displaying help information for more than one page should be performed in the same way.

3.6.3. Notification Event and Log Service

Task of the Notification Event and Log Service (NELS) is to receive, process, and store notifications. Notifications are pieces of information that inform an application about occurred events. A notification is sent in form of a ticket. The supported ticket is defined in the MAMA Core Model (cf. section 3.3.3.2).

Clients of the NELS can register in different roles. A client can restrict a subscription to notifications that describe a particular event, notifications of particular object classes, or notifications of particular object instances. These three types of subscription are also called channels. This means, a client can subscribe a certain channel that is offered by the NELS in order to receive notifications. Notifications can be sent to clients employing the push method or the pop method.

The NELS provides event service and log service functionality. The complete functionality can be described as notification service. Additionally, the NELS can function as a system monitor. For this task, the NELS must be provided with a console as output window for received tickets.

3.6.3.1. Notification and Event Service

The notification and event service part of the NELS offers services that are similar to the CORBA notification service [CORBA-NS] and the CORBA event service [CORBA-ES]. However, the functionality of the NELS is limited compared to the two CORBA services. This enables a simple and lean implementation of the NELS. For more complex application areas, the NELS can be realized in form of a mediator between MAMA applications and a notification service such as the one of CORBA.

A client must register at the NELS in order to receive tickets. This registration is called subscription. The subscription can be parameterized. The parameters are collected in a structure that is specified as follows:

```
struct nelsSubscription{
  [ValueMap("1", "2", "3", "4"),
   Values("Consumer", "Producer", "Subscriber", "Publisher")]
  short role;

  [ValueMap("1", "2"),
   Values("push", "pop")]
  short method;

  string channel;
  string objectClass;
  string objectInstance;
  MAMA::ticketCategory category;
};
```

The first member of the structure identifies the role of the client. A *consumer* and a *subscriber* want to receive tickets. A *producer* and a *publisher* submit tickets. The roles consumer and subscriber are the same. The roles producer and publisher are the same. The roles subscriber and publisher have been added for applications that follow this terminology (instead of consumer and producer).

The second member indicates how the client wants to receive tickets. Push means that the NELS sends notifications automatically, usually immediately after reception. Pop means that the NELS just stores notifications and the clients request them. This method implies a polling of the client.

The last four members can be used by a client to restrict the subscription. A channel is a pipe where events of a certain type can be submitted and received. Channels must be configured at the NELS. Furthermore, the client can specify that it is interested in tickets from a certain object class (*objectClass*), object instance (*objectInstance*), or of a certain category (*category*).

The operations *subscribe* and *unSubscribe* are used by clients to register at and de-register from the NELS. A client can register multiple times in different roles, using different methods, and for different channels. The two members *role* and *method* of the structure *nelsSubscription* must be set. For the other four members applies the following algorithm. When the value of the member is not the default value (which is NULL for strings and 0 for category), the clients wants to subscribe with restrictions. When it is the default value, the client wants to receive every ticket. In addition to the subscription structure, the client must provide its name in form of a distinguished name as specified by the DNSS.

```
boolean subscribe([In] nelsSubscription subscription,
                  [In] MAMA::oDNSS::tInstanceDN name);
boolean unSubscribe([In] nelsSubscription subscription,
                    [In] MAMA::oDNSS::tInstanceDN name);
```

The operation getChannels can be used by a client to request a list with all currently available channels.

```
string[] getChannels();
```

The operation submitTicket can be used by clients to send a ticket to the NELS. This operation should only be used by clients that have created the ticket. Furthermore, the clients should be registered at the NELS as a producer or publisher. However, it is also possible to send tickets without a prior subscription. The address of the client is not needed here, because it is already included in the ticket.

```
boolean submitTicket([In] MAMA::sTicket ticket);
```

3.6.3.2. Log and Monitoring Service

The NELS logs all received tickets. The actual number of tickets that should be logged can be configured. The default number is 500. The NELS logs tickets grouped in channels. Furthermore, the NELS should offer a monitoring in form of the following operation.

```
void showTicket();
```

When this operation is called, the NELS should print all tickets that are logged since the last call of this operation to the default output device. An implementation can also use a specific window or console for printing tickets. Further monitoring activities should be realized by a monitoring object.

3.6.4. Lifecycle and Configuration Management Service

This service offers generic mechanisms for two important issues of distributed systems:

1. The control of the lifecycle of object instances of a specific object class, and
2. The control of a configuration of object instances, from possibly different object classes.

The two features are combined to one service. The modeling follows the mechanisms and principles of the engineering viewpoint of ODP and the engineering model of TINA as described in [TINA-EMC]. MAMA supports development and runtime of distributed objects with the Lifecycle and Configuration Management Service (LCMS). Supported objects must share a common specification principle.

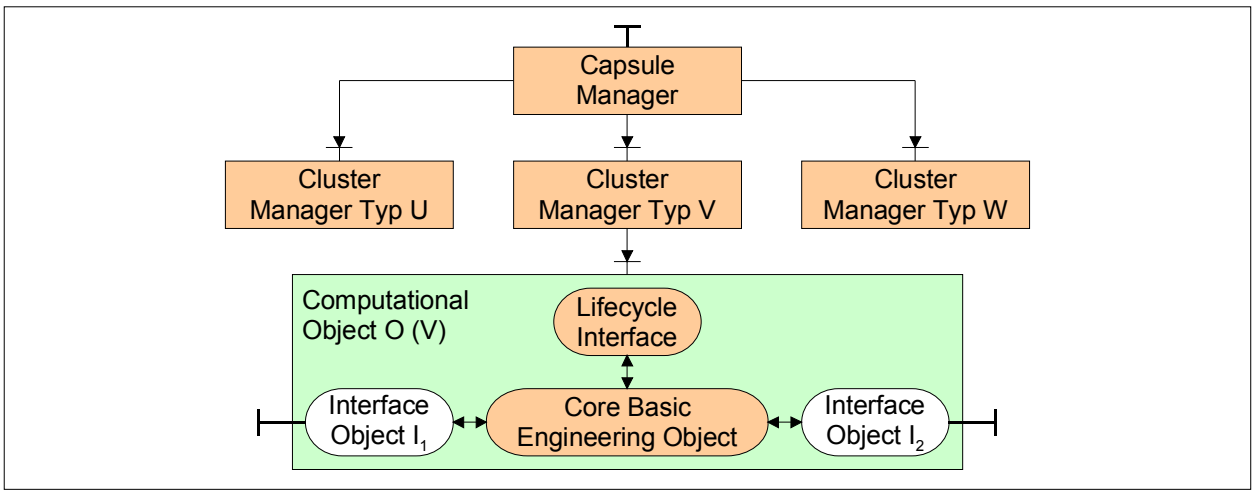

Figure 3-29: LCMS – Uniform Lifecycle Management [Eckert97]

The LCMS provides mechanisms for design and deployment. The lifecycle of objects with all its four stages – creation, initialization, execution, and termination – is supported in a uniform way. The designed configuration patterns are based on the TANGRAM project [Eckert97] and the object management of the intelligent Personal Communication Support System (iPCSS; [vdMeer96]).

Figure 3-29 shows the three components that have been identified. The capsule manager offers external interfaces of a configuration to its environment. It is also responsible for the creation, initializing, execution, and termination of a configuration of objects. Accompanying requests are delegated to type specific objects. The cluster manager is responsible for the creation, initializing, and termination of interfaces of one object class. The initialization and the termination of single objects are done by the invocation of operations on control interfaces of those objects. The object's control interface has to be provided by every computational object and can be understood as a uniform solution for the introduction of new components. The following code shows the basic type definitions for the LCMS:

```
struct sInterface{
  MAMA::oDNSS::tInstanceDN instanceDN;
  string intReference;
  MAMA::tMiddlewareReference intReferenceType;
  MAMA::oDNSS::tDN interfaceSpec;
};
typedef sInterface[] interfaceList;
```

```
struct sObject{
  MAMA::oDNSS::tInstanceDN objectDN;
  MAMA::oDNSS::tDN objectSpec;
  sInterface[] interfaces;
};
typedef sObject[] objectList;
```

The basic type definitions are introduced to enable the description of a single interface, a set of interfaces, a single object, and a set of objects. This information is further used by the operations of the object lifecycle interface, the cluster manager, and the capsule manager.

3.6.4.1. Object Lifecycle Interface

The objects lifecycle interface represents the basis of the LCMS. This interface provides three operations. Two operations are introduced to control the instantiation of an instance of an object class. This starts with the operation *create*, which causes the creation of the composite object. This creation comprises the core object, all interface objects the computational object supports, and the installation of appropriate relationships between core object and interface objects. The second operation is *init*, which supplies the object-specific initialization information to the new created instance. After the call to *init*, the object is completely instantiated and ready to operate in the distributed environment.

```
object oMamaObjectInit:MAMA::oMamaCore{
  interface iMamaObjectInit{
    boolean create();
    void init([In] MAMA::tNameValueList initParams);
    void terminate();
    void checkpoint();
  };
};
```

One operation is introduced for the termination of object instances. This operation is consequently named *terminate*. A call to this operation results in the deletion of all created interface objects, a cleanup of resources reserved and used by the object, and the final deletion of the core object.

To enable the temporary deactivation of objects, the object lifecycle interface should support an operation *checkpoint*. When this operation is called, the object should use any kind of mechanism to store its actual status in a non-volatile memory (that is making itself persistent) and terminate. The cluster manager of the object class is responsible for the provision of mechanism to restore a dedicated object instance with its state.

3.6.4.2. Cluster Manager

The cluster manager provides the service of managing the lifecycle of multiple instances of a dedicated object class. This component offers an abstract interface for this purpose. This interface is an aggregation of object factory and object naming service functionality. The object factory part is responsible for the creation, initialization, and termination of object instances. The name service part is designed to retrieve interfaces of instances of computational object.

The cluster manager relies on the ADL specifications of the object class it manages. It can be seen as guard object, delegating every call to its actions to the control interface of the addressed instance of the object class managed by it (cf. Figure 4-38).

```
object oClusterManager{
  interface iClusterManagement{
        [Permissions("0444")]
        attribute MAMA::mLCMS::objectList objects;

    MAMA::oDNSS::tInstanceDN create(MAMA::oDNSS::tInstanceDN instanceDN);
    void init([In] MAMA::oDNSS::tInstanceDN instanceDN,
              [In] MAMA::tNameValueList initParams);
```

```
    void terminate([In] MAMA::oDNSS::tInstanceDN instanceDN);
    MAMA::mLCMS::interfaceList getIntRefs(
            [In] MAMA::oDNSS::tInstanceDN instanceDN);
    MAMA::mLCMS::interfaceList getAllIntRefs();
    MAMA::oDNSS::tDN selectIntRef(
        [In] MAMA::oDNSS::tInstanceDN instanceDN,
        [In] MAMA::oDNSS::tDN interfaceType);
  };
};
```

Operation	Related to	Description
create	object factory	Create an instance of an object class. At first, the core object is created and then the needed instances of all interface objects. At second, all interface objects are connected to the core object. The capsule manager stores object references in a map.
init	object factory	Supplies created objects with initialization information. After that, the core object will be directed to initialize the computational object.
terminate	object factory	Delete the interface objects as well as the core object of the given instance. The core object will be notified about the deletion, so that allocated resources can be released.
getIntRefs	naming	Return all supported interfaces.
getAllIntRefs	naming	Return all known interface references.
selectIntRef	naming	Return the demanded interface of a specified object instance.

Table 3-29: LCMS – Cluster Management Operations

3.6.4.3. Capsule Manager

The capsule manager is a further abstraction of the cluster manager. It combines the object class specific cluster managers and offers an interface that allows the instantiation of a configuration of objects. This makes the capsule manager distinct for special environments. The interface includes a function *create* that is used to instantiate object classes on specific cluster managers. The other functions offer management functionality for discovery and lookup of interfaces. The capsule manager automatically searches for the cluster managers it should control. When one or more of them are not found, it will create them automatically. Most of the operations of the capsule manager are forwarded to the appropriate cluster manager.

```
object oCapsuleManager{
  interface iCapsuleManagement{
    boolean setType(MAMA::oDNSS::tDN elementDN);
    boolean delType(MAMA::oDNSS::tDN elementDN);
    MAMA::oDNSS::tDN getTypes();

    MAMA::mLCMS::sObject create([In] MAMA::oDNSS::tDN co_type,
                                [In] MAMA::oDNSS::tInstanceDN instanceDN);
    void init([In] MAMA::oDNSS::tDN co_type,
              [In] MAMA::oDNSS::tInstanceDN instanceDN,
              [In] MAMA::tNameValueList initParams);
    void terminate([In] MAMA::oDNSS::tInstanceDN instanceDN);
    MAMA::mLCMS::interfaceList getIntRefs(
        [In] MAMA::oDNSS::tDN co_type,
        [In] MAMA::oDNSS::tInstanceDN instanceDN);
    MAMA::mLCMS::objectList getAllIntRefs(
        [In] MAMA::oDNSS::tDN co_type,
        [In] MAMA::oDNSS::tInstanceDN instanceDN);
    MAMA::mLCMS::sInterface selectIntRef(
        [In] MAMA::oDNSS::tDN co_type,
```

```
        [In] MAMA::oDNSS::tInstanceDN instanceDN,
        [In] MAMA::oDNSS::tDN interface_type);
  };
};
```

Operation	Related to	Description
create	factory	Create and initializes instances of a specific object class.
terminate	factory	Realize the withdrawal of configurations for dedicated object instances.
getIntRefs	naming	Return information on all interfaces of the specified instance of an object.
getAllIntRefs	naming	Return information on all interfaces of all instances of an object class.
selectIntRef	naming	Return one interface reference of a queried interface type of a specific object instance.

Table 3-30: LCMS – Capsule Management Operations

3.7. Recommendations for the Design of MAMA Applications

This section focuses on issues that are important for the design of MAMA applications. Since not every designer of a distributed system is familiar with management systems, the recommendations start with a discussion of subjects related to a management hierarchy. Next, a methodology for the specification of application interfaces is explained. This technique is a combination of several analysis steps that – in combination – can lead a system designer to appropriate specifications for managers, agents, managed objects, managed resources, and other application objects. The final recommendation depicts possible variants to access non-MAMA compliant objects. This issue is of special interest when MAMA is used to integrate legacy systems or to provide migration strategies towards MAMA applications.

3.7.1. Special Issues regarding System Management

The MAMA protocol and API already support the design, installation, and configuration of management systems within MAMA. The peculiarities of system management need to be recognized in the development process to optimize control, administration, and maintenance. Figure 3-30 shows objects that act in the management roles. Here, managers (general or local managers) invoke operations on agents (general or sub agents). Agents direct the managed objects that are able to control specific resources. Communication in the other direction is realized by the exchange of notifications.

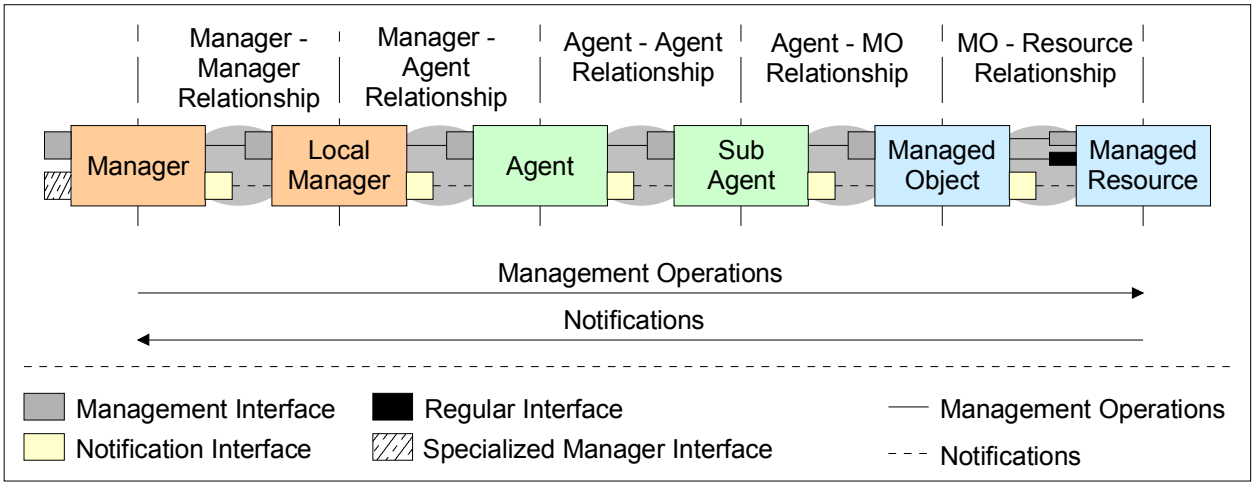

Figure 3-30: MAMA Development – Relationships between Management Roles and Interfaces

Managers, agents, and managed objects belong to the system's management. They are applied with management specific interfaces. The managed resource is either a logical resource in form of an application object or a physical resource in form of a device. Managed resources are not bound to the communication paradigm of operation/notification and need not to specify management-specific interfaces.

In a classical management system, notifications are exchanged via special notification interfaces. MAMA already offers an event service that should be used instead of the approach depicted by Figure 3-30. However, the system designer should be aware of the special relationships between all roles. These relationships model the functionality of each role. Managers are provided with information about the whole system that has to be managed. They generate complex management operations and offer them to superior management systems or human operators. Agents have only knowledge on specific parts of the system. They are able to split management operations to the managed objects they control. A managed object maps a simple management operation to appropriate operations of the managed resource. The Core Model already declares a type definition that covers all relevant management roles. Each application object can be accompanied with this information in order to characterize it as manager, agent, or managed object.

3.7.2. System Specifications

The possibilities that are offered by MAMA need to be carefully overlooked in the design and specification phase of a distributed system. Especially the combination of middleware and management functionality requires a systematic analysis of the tasks of the application(s). This analysis can be used as the basis for interface specifications and for the identification of proper object interactions. Regarding management systems, the task is to describe simple to use complex management operations offered by managers and to split them up into specific management operations of agents, managed objects, and proprietary managed resources. Hereby, the scope changes from general system management up to individual resource management.

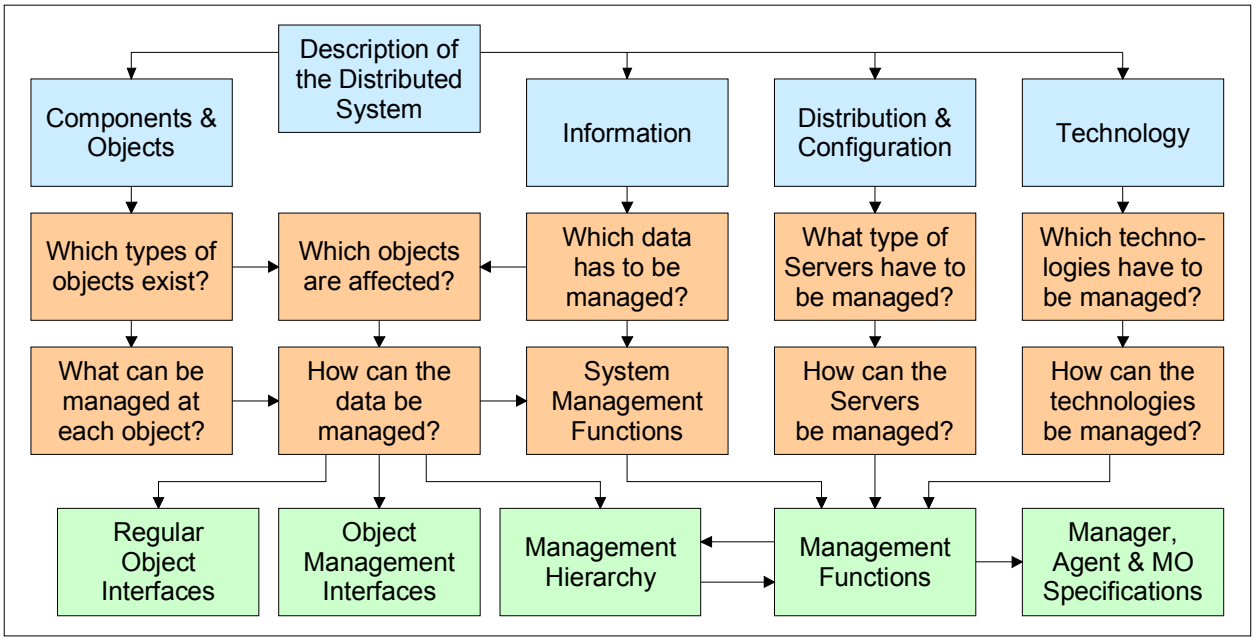

Figure 3-31: MAMA Development – Analysis

Figure 3-31 shows a recommendation for such a systematic analysis, which results in the definition of interfaces for management. This recommendation can be used to design standalone, separated management systems as well as integrated management systems. Other methodologies might be given precedence for the design of distributed applications without dedicated management functionality. The graphic shows four issues that are included in the overall system description: components and objects, information, distribution and configuration, and technology. Compared to ODP, they reflect the computational, information, engineering, and technology viewpoint. However, the system description might not be based on

ODP and so the four parts might not reflect ODP recommendations. The analysis starts with these basic information about the system, identifies what is important for its management, answers the questions on how to manage it, and specifies appropriate interfaces for managers, agents, and managed objects.

In general, the analysis can be divided in a system related part and a content related part. The system related analysis considers the structure of the distributed system, e.g. the relationship between objects, how they communicate with each other, or lifecycle issues. The content related analysis concentrates on the maintenance of the system's data that is hold and processed by the objects. The system and the content related analysis are used to clarify management semantics (control, administration, and maintenance). However, they can be also used to for usage semantic (usage, operation, and control).

3.7.2.1. Content-related Analysis

The content related analysis collects all data that is handled by the distributed application. This collection is compared to the major task of the application to construct data structures that reflect this task. The most important data can be combined to more or less complex data structures that need to be controlled and maintained as a whole. The processed data might be distributed over several objects. However, this step of the analysis does not consider the objects the data is assigned to.

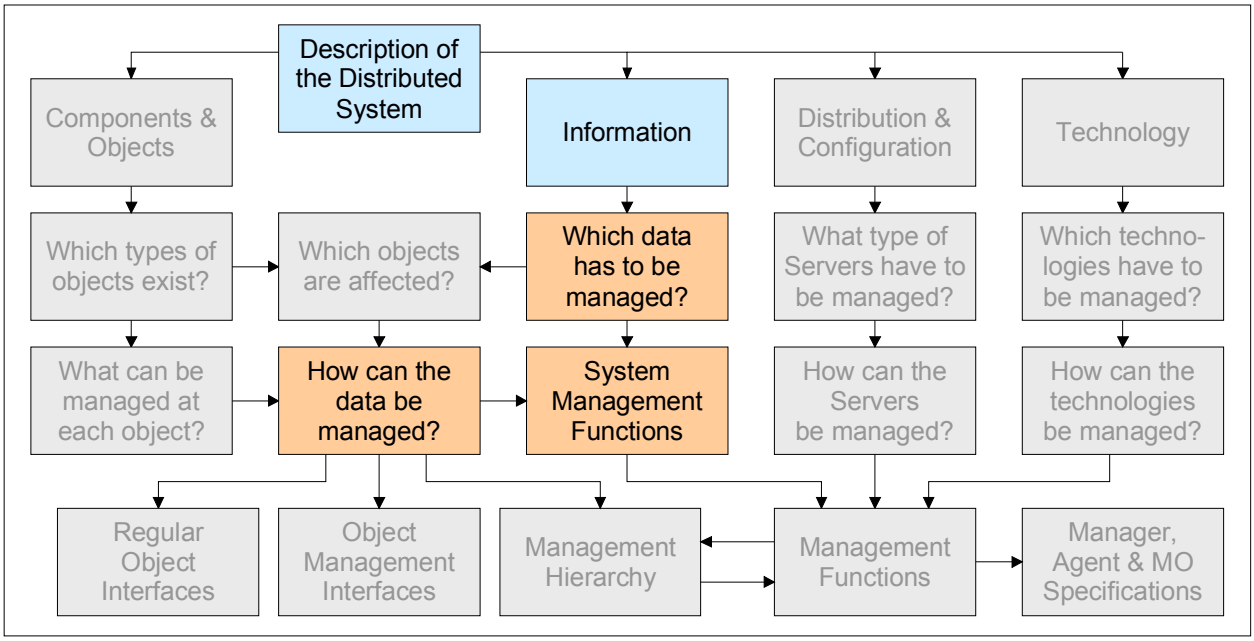

Figure 3-32: MAMA Development – Content-related Analysis

To give an example: A system maintains information about users. A single user is described by his real name, a unique identifier, an address, a password, and some communication addresses like email and telephone number. This kind of data is handled by different objects. One part of the system deals with authentication (that is identifier and password), another part of the system covers communications (here the addresses are of interest). The information about a single user need to be maintained as if they where stored at a single place. The result of the content related analysis is the definition of a complex data structure, which might be called user profile, which is managed at once regarding the introduction of new users, the change of user related information, and the deletion of users.

The first step answers the question which data has to be managed. The second step clarifies how the data can be managed. The later result is the basis to specify system wide management functions that can be further used to identify general management operations of managers and local managers as well as single management operations of agents.

3.7.2.2. System-related Analysis

The system related analysis considers the structure of the system. In the first step, the individual components and objects of the system are examined. Each object handles specific types of data and has certain relationships to other objects of the overall system. One result of the first step is to investigate the distribution of the data structures of the content related analysis and to combine this description with the actual place the data structures are processed at. This includes the communication links and dependencies among objects. Figure 3-33 shows the two questions that need to be answered by this first step: Which objects exists in the system and what type of data can (and should) be managed on those objects.

The second step focuses on the distribution aspect and on the actual configuration of the system. There are several suitable ways for the implementation and deployment of objects. In general, each instance of an object is handled by a server and those servers also offer management functionality. This includes the instantiation and termination of objects as well as the activation and deactivation of complete objects or single interfaces. The third step follows technological aspects. Here, specific technologies that should be included in the system's management are analyzed to specify appropriate management functions.

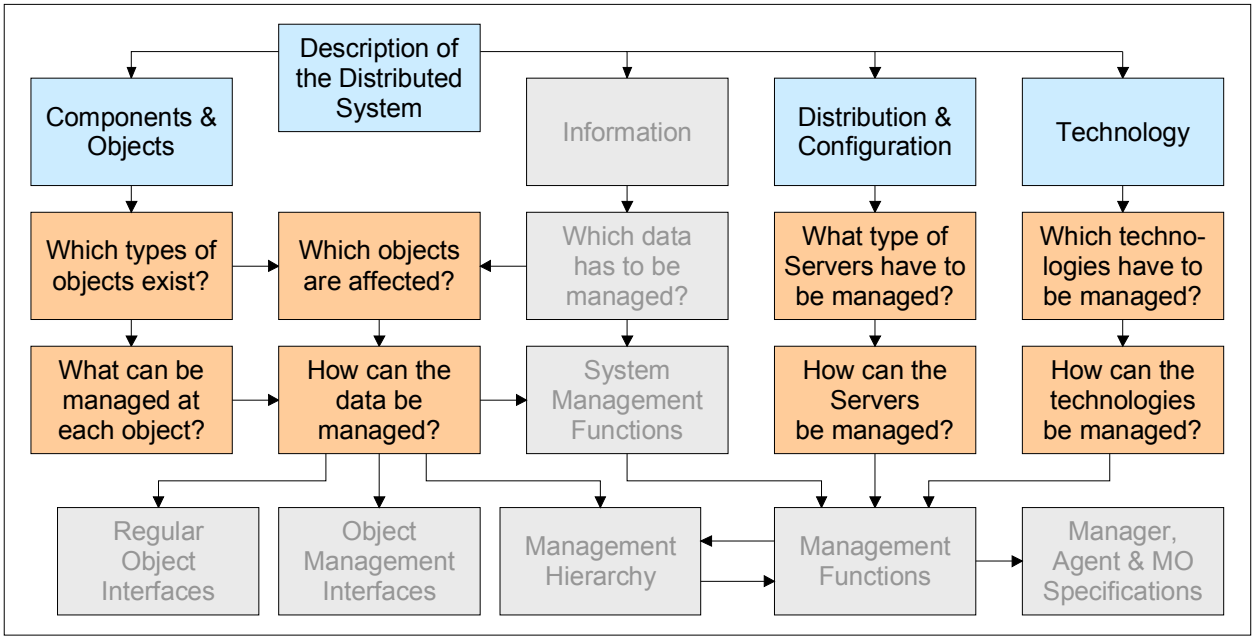

Figure 3-33: MAMA Development – System-related Analysis

Finally, the management needs of the determined objects, servers, and technologies have to be examined regarding system management. This leads to the definition of the management interface of each object type and to the definition of corresponding management functions. It is up to the considered system whether the individual managed objects are supported with a corresponding management interface, or if the necessary management functionality is shifted to the appropriate agent. For example, if an already implemented system should be provided with management facilities belatedly, the individual objects usually cannot be supported with an extra management interface. Then, the ordinary interfaces of the objects have to be sufficient to comply with the management needs. With it, the controlling agent of the appropriate objects has to translate the management operations and the configuration functions to the according operations of the ordinary interfaces. These kinds of managed objects can be managed by such extended and specialized agents only.

3.7.2.3. Specifications

The outcome of all analysis steps is the specification of a management system (cf. Figure 3-34). This specification comprises regular object interfaces that can be used for management purposes, the definition of object management interfaces that can be assigned to existing objects, and the definition of managed objects that supervise managed resources. Furthermore, the specification describes the management hierarchy. This hierarchy takes into account business processes and other constrains that enable a secure and

effective operation of the system. The hierarchy consists at least of one manager and one agent. For more complex systems, the hierarchy can be defined in a more fine-graded way with local managers and sub-agents to realize geographical or organizational structures.

The system's management functions are processed and forwarded within the management hierarchy. Managers provide simple management operations that cover complex management tasks. Those operations need to be forwarded to appropriate local managers and agents while the manager still keeps control of their execution. With the management functions, single interfaces of all roles in the management hierarchy can be specified.

The specifications are used to assemble repositories of a MAMA system, similar to MIBs. Management standards already identify basic criteria for the design of a MIB that can be also applied here. The Internet Engineering Task Force (IETF) standard [IETF-RFC1493] summarizes such criteria as follows.

1. Start with a small set of essential objects and add only as further objects are needed.
2. Require objects be essential for either fault or configuration management.
3. Consider evidence of current use and/or utility.
4. Limit the total number of objects.
5. Exclude objects which are simply derivable from other objects in this or other MIBs.
6. Avoid causing critical sections to be heavily instrumented.

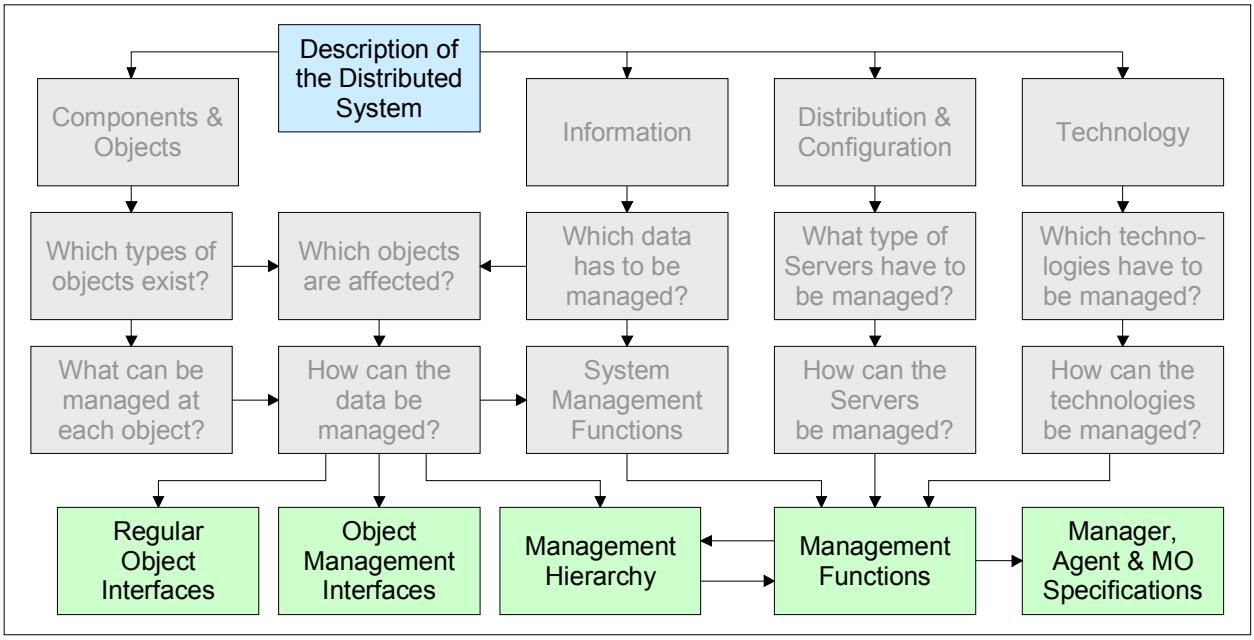

Figure 3-34: MAMA Development – Specifications

These criteria have been extended because experiences have shown that the actual design of MIB modules and repositories is a mission critical step in the specification of a management system [Draft-Config]:

- Before MIB Module design, identify goals and objectives for the MIB module. How much of the underlying system will be exposed depends on goals set.
- Minimizing the total number of objects is not explicit goal, but usability is. Be sure to consider deployment and usability requirements.
- During configuration, consider supporting explicit error state, capability, and capacity objects.
- When evaluating rule number five above, consider the impact on a management application. If an object can help reduce an application's complexity, consider defining objects that can be derived.

[Draft-Config] gives further examples for the best common practice defining MIB modules. The draft document identifies a variety of problems that occur during this process. To give a comprehensive example, many specifications (for MIB modules as well as for other purposes) assume that an object can com-

plete a *set* operation as quickly as it is requested and alters the current state to the new one. The following ADL example models a wheel that can spin clockwise and counter clockwise. The example specifies a variable that publishes the current state of the wheel.

```
object oWheel{
  interface iWheelRotationState{
    [Description("The current state of a wheel."),
     ValueMap("0", "1", "2", "3")
     Values("unknown", "idle", "spinClockwise", "spinCounterClockwise")]
    attribute iWheelRotationState;
  };
};
```

The system designer might not have considered the question how long it takes the wheel to change the actual rotation state. The variable with the state needs to be enhanced with a transition state that indicates that the wheel is currently changing the rotation state, which can involve a number of actions: stopping the wheel, changing the configuration of the motor, starting to rotate in the new direction. [Draft-Config] shows explains other issues that need to be taken care off defining a MIB module.

Specification of Managed Objects

Specifications of managed objects involve all the management needs related to managed resources. This regards to the data of the objects as well as to the position of the objects within the system. Furthermore, a description of the management interface for the managed object is part of the specification. Fundamentally, the management interface should comply with the general management concept. The interface has to support the set-/get-operations with appropriate configuration functions. If the management interface does not comply with the general concept or does not exist at all, the specification has to be considered within an agent specification. This kind of agent is then responsible for the management of this kind of managed object.

Managed objects can be divided into dynamic and static managed objects. Static objects can be accessed by the management system, but their lifecycle is controlled by other authorities. Dynamic objects are completely controlled by the management system. The MAMA API identifies this fact within the specification of entity types (cf. section 3.3.3.4).

Specification of Agents

Agents are able to transform management operations they offer to superior management objects into a sequence of operations that need to be executed on one or more managed objects/resources. An agent controls a unit of work in form of a transaction. Furthermore, agents are responsible for handling notifications respecting the system's policy (e.g. described by forwarding discriminators). Agents should at least inherit functionality from the management part of the MAMA API and possibly from the cluster manager of the LCMS when they control dynamic managed objects.

Specification of the Manager(s)

Specification of the Manager(s) considers on the one hand the provided functionality and on the other hand the realization of this functionality. This functionality embraces all management operations within the system. This includes the performance of management operations as well the representation of system's information. The appropriate operations should be put together in a system's management interface provided by the manager. Then, this interface may be used by user applications or by higher management systems.

Usually, these management actions and their data are more complex and therefore have to be split into several single management operations. This translation is an important result of the analyzing process. The specification of the manager has to involve how the individual management operations are performed on the system, that means which managed objects and which configuration functions are affected. Furthermore, possible dependencies among objects have to be considered performing a management operation. It has to be determined if these operations should be performed as transactional operations.

3.7.3. Access to non-MAMA Objects

MAMA offers functionality for all requirements of a distributed application. However, it might be necessary to access objects that are not compliant with MAMA. Those objects do not implement the MAMA interface and cannot be accessed via the MAMA protocol. In order to allow communication with such objects, the variants of access to non-MAMA objects are discussed in this section. Three different mechanisms can be used to integrated objects that are not based on MAMA.

1. The object offers only regular, operational interfaces. In this case, a MAMA object is in charge to provide an ADL-typed interface towards the MAMA system and to use those operational interfaces. This special object functions than as a gateway between the MAMA system and the legacy system.
2. The object offers regular, operational interfaces and a MAMA-like interface. This case is depicted by Figure 3-35. Now, the special gateway object can still use the operational interfaces. However, important functionality can be put into the MAMA-like interface to simplify the translation of specifications and operation calls.
3. The object offers only a MAMA-like interface. In this case, the object can be considered as an almost MAMA compliant object. The system designer should investigate whether the object can be converted to a completely MAMA compliant object or the MAMA system still needs a gateway object for access.

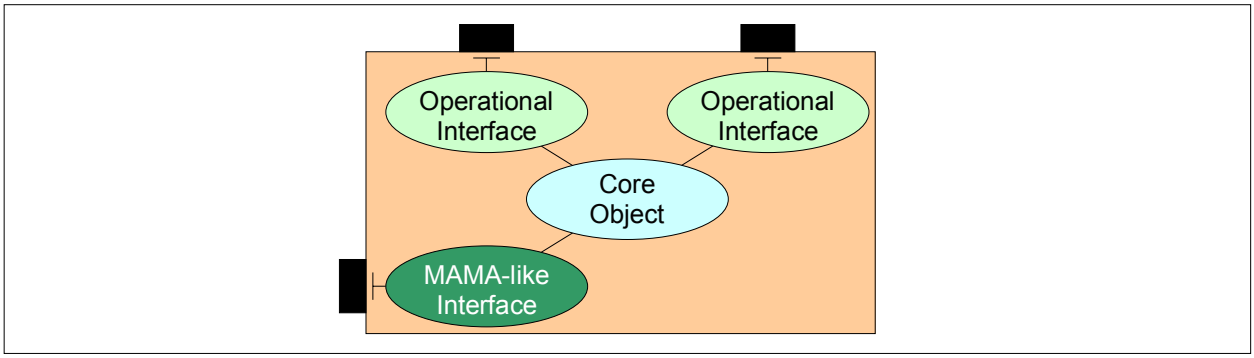

Figure 3-35: MAMA Development – Object with MAMA-like Interfaces

Management of distributed objects can be realized by calling configuration functions of the object's regular interfaces. However, it is sensible to combine all configuration functions that should be offered as management operations into one or multiple extra interfaces – the management or MAMA-like interface(s). Figure 3-35 shows the engineering view of a distributed object with one additional management interface. The definition of such an additional interface does not influence the existing operational interfaces. It decouples usage from management (what makes it easy to introduce different policies and security strategies for usage and management), and enables a flexible management system independent from the actual task of the distributed system. The usage of a management interface is inevitable when the regular interfaces do not provide the necessary configuration functions. The following code shows a simple interface that is almost conforming to the MAMA interface:

```
interface iManagedObject {
            set ([In] string ConfigurationFunction, [In] ValueList values);
  ValuesList get ([In] string ConfigurationFunction);
};
```

This specification can be easily mapped to middleware IDL. Additionally, the object applied with this interface has no significant overhead in dealing with the parameters. The parameter *ConfigurationFunction* needs to be evaluated to select an appropriate implemented operation of the core object to forward it to. Next, the parameter *values* must be examined to extract the values given with the configuration function. All other features of the protocol, such as hierarchical addressing and transaction handling, are not subject of integration and can be left out of scope.

An important issue is the ability to send and to receive tickets. A non-MAMA compliant object might still be in the position to send tickets and also want to receive them. For this case, the interface needs to be enhanced (or a second interface needs to be added). This interface should be capable of handling simple tickets that the NELS can process. The following code shows an extra notification interface with such a simple structure for tickets in OMG IDL notation.

```
interface iNotification {
  enum NotificationType {
    information,  // for informational notifications
    error         // for notifications of errors
 };

  struct Notification {
    integer          priority; // notification priority
    string           sender;   // sending object's reference
    NotificationType type;
    string           date;
    string           time;
    string           message;  // textual event description
  };

  void notify(in Notification notification);
};
```

This interface solves the tasks of receiving tickets and defines a data structure for tickets. In order to be able to send tickets, the non-MAMA compliant object must be enabled to access a MAMA event server. This process is environment specific and differs from middleware to middleware (and possibly from programming language to programming language). Therefore, no further recommendations are given within this document.

Chapter 4
Realization

This chapter features the realization of the Middleware and Application Management Architecture (MAMA). Each solution of chapter three is represented by appropriate implementations. A compiler for the Application Definition Language (ADL) is a tool for the automated processing of ADL specifications. The Core Model and the protocol are realized by specific libraries that, in combination, form the processing environment. This environment can be accessed via the MAMA Application Programming Interface (API). One implementation of the API is described in detail to show how the specifications of chapter three can be transformed to programming language and operating system specific libraries. Finally, the implementations of application services are explained in detail.

4.1. ADL Compiler

The ADL compiler has been developed to enable an automated parsing, evaluation, and transformation of the formats ADL and eXchange ADL (xADL). The compiler accepts only complete ADL and xADL specifications for processing. The ADL compiler automatically detects the format of the given input file.

Every ADL file has to start with a preprocessor directive, a comment, or the declaration of a qualifier, module, object, or type definition. xADL files need to start with the XML[1] tag *<?xml*. Files that start with other tokens are rejected. The compiler consists of two parts; one part for each language it supports as shown in Figure 4-1. After the file check, the processing is handed over to the part of the compiler that is responsible for the detected language.

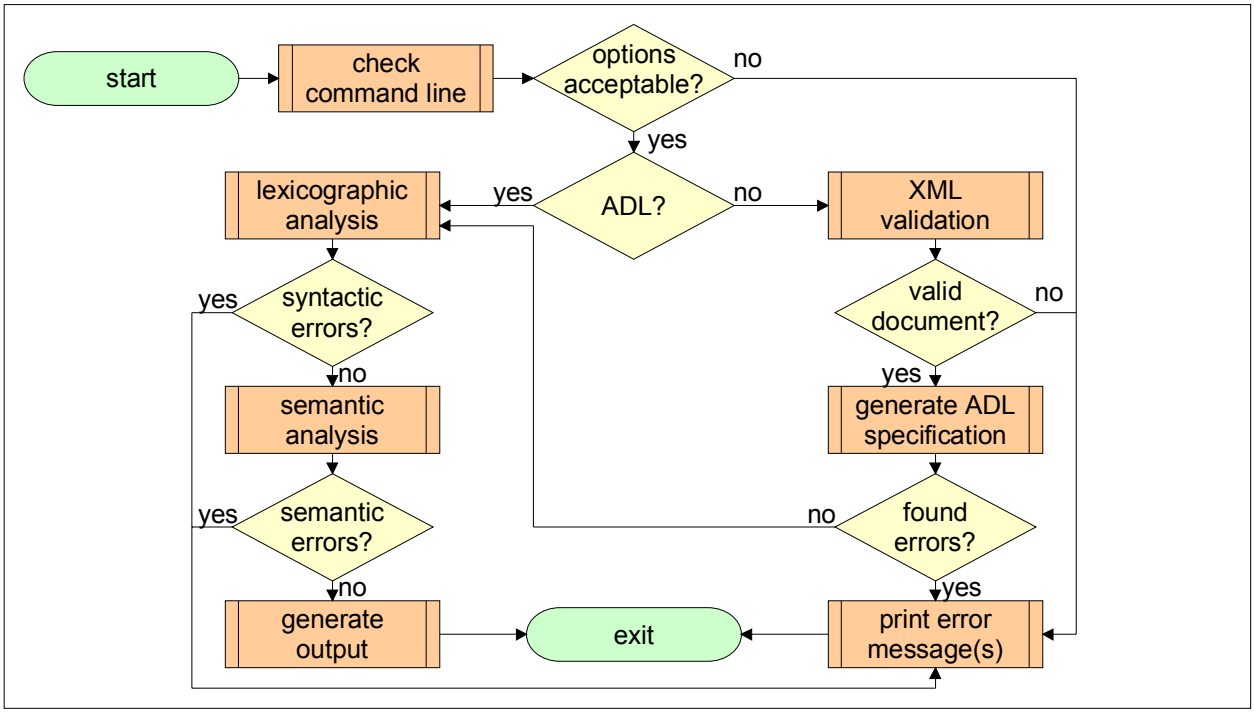

Figure 4-1: ADL Compiler – Work Flow

[1] eXtensible Markup Language

An xADL file is processed in the following way. First, the compiler searches for a Document Type Definition (DTD) file that is either contained in the xADL specification (in line 2, 3, or 4 of the xADL file) or must be provided via a command line option. Now, the compiler checks if the xADL specification is valid for the given DTD. After that, it generates temporary ADL. This ADL is further processed by the ADL part of the compiler. Finally, the compiler removes all temporary created files. When requested via command line options, the ADL specification is moved from the temporary place to the location of the xADL specification.

The processing of an ADL file starts with the invocation of a preprocessor, when requested. The output of the preprocessor is than a file that contains the complete ADL specification. This file is handed over to the lexer, which is responsible for lexicographic analysis. After that, the parser is started that applies the ADL grammar to the input character stream. Finally, the compiler generates xADL output when requested via command line options. The XML tags in this file are automatically formatted for better readability.

The compiler does not check the types of qualifiers against the qualifier declarations. However, it (in detail the part that parses an ADL specification) does the following examinations:

- Validate the qualifier declarations corresponding to the ADL rules (cf. section 3.2.2.7).
- Validate the definitions of identifiers, corresponding to the ADL naming conventions (cf. section 3.2.1.3).
- Check if all required/mandatory qualifiers of the Core Model are applied to the ADL elements module, object, interface, actions, attribute, parameter, type definition (typedef), and member.
- Check if qualifiers that are applied to an ADL element have been defined in the Core Model to be applicable to this very type of ADL element (cf. section 3.3.2).
- Includes new type definition in the global map of types and checks later usage.
- Validates object inheritance regarding scoping and naming rules (cf. section 3.2.4).

Additionally, the ADL compiler includes facilities for the generation of statistical information. This information can be printed out to the standard output in a simple or complete form. This output can include complete naming scopes or the identifiers only.

4.1.1. ANother Tool for Language Recognition

The ADL parser has been implemented using the ANother Tool for Language Recognition (ANTLR) parser generator. ANTLR constructs human-readable recursive-descent parsers in C++ and Java from pred-LL(k) grammars. This tool can generate parsers for context-sensitive languages and non-LL(k) context-free languages. [Parr95]

The yacc[2] LR-based parser generator was one of the first tools that unified the specification of lexer and parser. ANTLR goes a bit further recognizing a stream of input that can be characters, tokens, or tree nodes. The three different types of parsers (lexers, parsers, and tree parsers) are specified using the same syntax and the same code generator. [ANTLR-Man]

The ADL parser uses the lexer and parser facilities of ANTLR. The lexer (also called a scanner) breaks up the input stream of characters into vocabulary symbols for a parser. The parser applies a grammatical structure to that symbol stream.

In addition, ANTLR offers many features that enabled a fast realization and implementation of the ADL parser. ANTLR accepts grammar constructs in the Extended Backus-Naur Form (EBNF), which is the notation used to define ADL. ANTLR has automatic and manual facilities for error handling and recovery that allowed ADL specific error messages. ANTLR allows grammar rules to have parameters and return values. This feature has been used for the evaluation of ADL element identifiers, for the generation of static information on a given ADL specification, and for revision of the ADL naming conventions.

[2] yet another compiler compiler

4.1.2. Implementation

The implementation comprises C++ files (file extension *.cpp*), C++ header files (file extension *.h*), and ANTLR grammar files (file extension *.g*). The ANTLR grammar files are further separated in files that contain lexer and parser specific parts of the grammar.

Files	Content
main.cpp	This file contains the main function of the compiler, including facilities for general file access, command line parsing, language detection (ADL or xADL), handling of temporary files, and the help text as default output.
adl.cpp	This file represents the C++ access to the ADL part of the compiler. Here, the ADL lexer and the ADL parser are instantiated, appropriate functions for the processing of the given ADL file called, and the final output is prepared.
xadl.cpp	This file represents the C++ access to the xADL part of the compiler. Here, the xADL lexer and the xADL parser are instantiated, appropriate functions for the processing of the given xADL file called, and the final output is prepared.
dtd.cpp	The XML parsing facility is based on the Apache package Xerces. The file *dtd.cpp* is an adoption of the original sample file *SAXPrint.cpp* from Xerces. The ADL compiler is independent of any external tool for XML parsing, because the Xerces libraries all necessary functionality could be built-in right into the compiler itself.
adllexer.g	This file contains the grammar for lexicographic analysis of ADL specifications.
adlparser.g	This file includes private member functions for the ADL parser and the actual ANTLR grammar for parsing ADL specifications (cf. Appendix B.3.9). Furthermore, it contains functions for the generation of a character stream (as input for the lexer) and for error handling (to catch ANTLR exceptions and print appropriate error messages).
xadllexer.g	This file is similar to *adllexer.g*, except that it defines the lexer grammar for xADL specifications (cf. Appendix B.4.3)
xadlparser.g	This file contains the ANTLR grammar for xADL specifications. This grammar is used to transform an xADL specification into an ADL specification that can be further processed by facilities declared in *adlparser.g*.

Table 4-1: ADL Compiler – Files of the Implementation

The files with the suffix *.g* are processed by ANTLR, which generates appropriate C++ and header files for lexer and parser. Those files contain C++ declarations for the actual definitions made in the ANTLR grammar files.

The developed ADL compiler does not include facilities for preprocessing. Preprocessing tools exist in every development environment, so that available parsers can be reused. The current implementation is designed to use the preprocessor *cpp* from the Microsoft development environment. This tool generates a file with the suffix *.i* in the temporary directory of the operating system. The ADL parser invokes the preprocessor with the options */nologo* and */Tp*, waits for its termination, and searches for the generated file. This file is than further processed by the ADL compiler.

4.1.3. Command Line Options

The compiler can be parameterized in several ways via command line options. Mandatory for the processing of a specification is to specify the file that should be processed. The parser will not produce any output unless instructed to do so. This output can be either in ADL or in xADL format. The parser generates statistical information that can be printed after the successful processing of a specification. This informa-

tion includes per default identifiers with their complete scoped names. Furthermore, the parser provides complete statistic information on request.

Option	Description
[-c | --complete-stat]	Print complete statistic information. This includes separated counts of specified ADL elements and separated listings of identifiers for each ADL element type.
[-a | --adl]	With this flag and a given specification in xADL, the parser generates ADL after a successful processing of the file. The resulting filename is the input filename with the extension *.adl*.
[-d | --dtd] <file>	This specifies the DTD in form of a file that is needed for XML parsing.
[-f | --file] <file>	This flag specifies the file the parser should process. The file can be either an ADL or an xADL file.
[-h | --help | -? | /?]	Display help information and exit.
[-n | --no-scope]	Print statistics without the naming scope of identifiers. In this case, only the identifiers are listed.
[-p | --preprocessor]	Use a preprocessor before actually processing ADL files. The parser has no built-in facility for preprocessing, so this feature must be provided by an external tool. This tool is started by the compiler. The default configuration for the tool is the Microsoft C++ compiler.
[-s | --stat]	Print simple statistics for the successfully processed specification.
[-v | --version]	Display the program's version and exit.
[-x | --xml]	With this flag and a given specification in ADL, the parser generates xADL after a successful processing of the file. The resulting filename is the input filename with the extension *.xml*.

Table 4-2: ADL Compiler – Command Line Options

4.2. Protocol

The generic ADL specification of the MAMA protocol needs to be mapped to the actually supported middleware. This document defines the mapping for the Interface Definition Language (IDL) as defined by the Object Management Group (OMG). Mappings to other interface definition languages can be done in a similar way.

ADL and OMG IDL have a number of communalities that ease the mapping of ADL specifications to OMG IDL. The following code shows the MAMA interface in OMG IDL:

```
module MAMA{
  interface iManagement{
    IDLSeqNamedValue swAction(in IDLOperation operation,
                              in IDLSeqObjectPath addresses,
                              in IDLSeqNamedValue parameters,
                              in IDLSeqNamedValue options)
};
```

The module *MAMA* serves as a name space for the protocol specifications. The IDL compiler translates this name space into the employed programming language. No name clashing can occur, as long as no other module *MAMA* exists.

The suffixes of the names are changed from *t* to *IDL* to indicate for which environment the specification is made. The attributes of the operation *swAction* can be mapped with little changes regarding OMG IDL as described in Table 4-3.

ADL	OMG IDL	Description
swAction	swAction	The protocol operation that handles all operation calls. The name of this operation is kept the same as introduced in the MAMA protocol.
tOperation	IDLOperation	No changes made.
tSeqObjectPath	IDLSeqObjectPath	The type of the list is moved from an array of strings (ADL) towards a sequence of strings (IDL).
tNameValueList	IDLSeqNamedValue	The type of the list is changed from an array of strings (ADL) towards a sequence of strings (IDL).

Table 4-3: MAMA Protocol – ADL to OMG IDL Mapping

The interface defines three structures. The structures represent a direct mapping from the ADL structures that are defined for the protocol in section 3.4.1.

4.2.1. IDLSeqNamedValue

```
struct IDLNamedValue{
  string       name;
  string       value;
  IDLDataType  vdatatype;
  IDLDataFlag  vdataflag;
};
typedef sequence<IDLNamedValue> IDLSeqNamedValue;
```

This sequence is used to transmit Name-Value Lists (NVL). The elements *name* and *value* are similar to the ADL specification of the protocol.

The two flags *vdatatype* and *vdataflag* are specified with the MAMA qualifiers *Values* and *ValueMap*. Those qualifiers provide the declaration of enumerated items that can be applied to an attribute or parameter. In OMG IDL, this specification can be made as enumerate or using constant values. Enumerates are a list of non-negative integers.

A critical drawback of enumerates is that implementations can be written relying on the integers. When the specification changes and integers are rearranged, the implementations need to be adapted. The usage of constant value solves this problem. Here, an integer value is assigned to a specific identifier. Implementations use the identifier, and not the actual value. Drawback is a slightly more complex specification.

The member *vdatatype* represents the ADL member *nvDataType*. It is used by the MAMA API to identify the type of a value in the NVL. This information is necessary for marshalling and de-marshalling.

```
typedef unsigned short IDLDataType;
const IDLDataType IDL_DT_inconsistent = 0;
const IDLDataType IDL_DT_char         = 1;
const IDLDataType IDL_DT_string       = 2;
const IDLDataType IDL_DT_boolean      = 3;
const IDLDataType IDL_DT_octet        = 4;
const IDLDataType IDL_DT_short        = 5;
const IDLDataType IDL_DT_ushort       = 6;
const IDLDataType IDL_DT_long         = 7;
const IDLDataType IDL_DT_ulong        = 8;
const IDLDataType IDL_DT_longlong     = 9;
```

```
const IDLDataType IDL_DT_ulonglong    = 10;
const IDLDataType IDL_DT_float        = 11;
const IDLDataType IDL_DT_double       = 12;
const IDLDataType IDL_DT_longdouble   = 13;
const IDLDataType IDL_DT_array        = 18;
const IDLDataType IDL_DT_struct       = 19;
```

The member *vdataflag* represents the ADL member *tAccessFlag*. This flag defines the access to a value in the NVL. It reflects the MAMA qualifiers *In* and *Out*. Parameters that are declared as *In* are read-only (*IDL_DF_read*). Parameters that are declared as *Out* are write-only (*IDL_DF_write*). The integer values can be combined in order to allow read and write of a parameter. The rules for this combination similar to UNIX file system attributes as explained for the qualifier *Permissions* in section 3.3.2.3.

```
typedef unsigned short IDLDataFlag;
const IDLDataFlag IDL_DF_none  = 0;
const IDLDataFlag IDL_DF_read  = 1;
const IDLDataFlag IDL_DF_write = 2;
const IDLDataFlag IDL_DF_exec  = 4;
```

4.2.2. IDLSeqObjectPath

This sequence contains the addresses of objects in form of object paths. The ADL specification is given in section 3.4.1.2. The path is traversed and evaluated by the MAMA API according to the addressing mechanisms presented in section 3.4.3.

```
typedef string IDLObjectPath;
typedef sequence<IDLObjectPath> IDLSeqObjectPath;
```

4.3. Application Programming Interface

The API has been realized for the programming languages C++ and Java. The following sections describe the C++ port of the API. All specifications are given in C++ syntax. It was developed on Windows NT (Service Pack 6a) and on Windows 2000 (Service Pack2). The C++ port employs the following development environment:

- RogueWave's class libraries Tools.h++, Tools.h++ Professional, and Threads.h++;
- IONA's Orbix 3.0.1 (Patch 20); and
- Microsoft Visual Studio 6.0, including Visual Source Safe (Service Release 4).

The RougeWave class library offers the API a huge number of basic functionality such as list handling, comfortable string processing, and thread handling. Orbix represents the Object Request Broker (ORB) that is used for the communication between applications. Visual Studio includes functionality for debugging, source repositories, and project management.

The current version of the C++ API uses static linked libraries. Future releases will use Dynamic Linked Libraries (DLL) in order to reduce occupied resources and to increase the flexibility of the API.

4.3.1. Classes reused from the UMS

The C++ API reuses a number of classes that have been developed for the Unified Messaging System (UMS; [vdMeer00b]). The original specification of those classes can be found in [Dutkowski01]. The classes are grouped in a library called *MAMACUtilLib*. The classes are *CommandLineParser*, *ObjectManager*, *ThreadFilter*, *TSingleton*, *UnifiedIdentifier*, *Event*, and *VersionInfo*.

4.3.1.1. ThreadFilter

The *ThreadFilter* class is an example of a proposed threading mechanism implementation using RogueWave's Thread.h++ library and Orbix's proprietary Filter extensions. This class is based on the *UMSThreadFilter*. A few adjustments have been made. This concerns especially operations for controlling capsulated thread handling. Actually, the mechanism is built on the collaboration between the filter class and a policy class. For each possible policy, a specialized class is derived from the policy base class. The *UMSServer* base class for applications handles almost everything automatically that is necessary to use these classes, except selecting a policy. [Dutkowski01]

4.3.1.2. Event

The *SWEvent* class encapsulates the ticket structure. It is implemented comparable to a stream object from the standard C++ library. Information can be streamed using the operator (<<*operator*) together with manipulators. This information is buffered until the operation *submit* is called, similar to flush a stream buffer.

4.3.1.3. UnifiedIdentifier

Universally Unique Identifiers (UUID) are used within the API for identifying objects. Identifiers are encapsulated by this class. *UnifiedIdentifiers* are based on the UUID generator implemented by Microsoft.

4.3.2. SWAPI

Synopsis

```
#include <API/SWAPI.h>
SWAPI api;
```

Description

The class *SWAPI* represents the main class for the application programmer. This class realizes the configuration of the API, the registration of the application at the naming service and at the event service, configuration of the employed middleware, and provision of an operation for the invocation of actions.

Usage

Each component must instantiate at least one *SWAPI* object.

Members

```
int initEntity(const SWMiddleWare& mwtype,
               const unsigned long maxerrors = SW_MAXERRORS,
               const bool transmiterror = false,
               const RWCString& ServerName = (RWCString)"" );
```

This operation configures the API. It must be called before any other API operation. An application can set the type of middleware that should be used for communication (*SWMiddleWare*). The parameter *maxerrors* defines the maximum number of errors the API should store when no connection to an event service exists. The parameter *transmiterror* instructs the API to send errors immediately to the event server (true) or not at all (false). The parameter *ServerName* can be used to supply a name for the application that is used for the registration at the naming service.

```
int configMiddleWare(const Management_ptr TIE);
```

This operation binds an existing TIE object to the *SWAPI* object.

```
int registerEvSrv(const unsigned long evFlags);
```

The application must register itself at the event service. This registration follows the rules of the MAMA event service as explained in section 3.6.3.

```
int deregisterEvSrv();
```

Deregistration from the event service.

```
int changeRegistrationEvSrv(const unsigned long evFlags);
```

A change in the registration to the event server is realized by a de-registration and a following new registration. The API takes over the related calls to the event service.

```
struct SWArgStruct{
  RWCString        OperationName;
  SWAddressList    AddressList;
  SWParametersList ParametersList;
  SWOptionsList    OptionsList;
};
int performAction(SWArgStruct& ArgStruct,
                  SWReturnList& ReturnList);
```

This operation performs an action call on another MAMA application. It completely implements the MAMA protocol, including the rules for marshalling and de-marshalling of the operation parameters.

```
int sendEvent(const RWCString& desc, unsigned long number);
```

Send the given event to the event service.

```
void addNewOperation(const RWCString& operation, const PSWOPERATION pop,
                     const RWCString& descr);
```

The application needs to announce each supported operation to the API in order to make them available. An operation can be clearly identified with its name and a pointer to this very operation.

4.3.3. Standard Library

The standard library (*MAMAStandardLib*) comprises all classes that support the API. Those classes are used for the provision of internal maps inside of the API and for the conversion of C++ data into OMG IDL.

4.3.3.1. SWNamedValue

The class *SWNamedValue* is the implementation of a name-value pair.

Synopsis

```
#include <SWStandard/SWNamedValue.h>
SWNamedValue nv;
```

Description

This classes mainly features the mapping the OMG IDL data type *IDLNamedValue* to C++ and vice versa. A name-value pair is a structure with the key name and the value. The flags are for setting the original type of the value to retranslate the presentation of string correctly.

Usage

The developer does not need to address this class directly. The API creates an instance of this class for every conversion from OMG IDL to C++ and vice versa. The class defines methods for persistence to save the current state into files or write it into streams.

Members

```
RWCString Name() const;
```

Return the name.

```
void changeName(const RWCString& name);
```

Change the name.

```
RWCString Value() const;
```

Return the value.

```
SWDataType _dt() const { return m_ValueType; };
```

This operation determines the type of the value field. Returned is an integer according to the definitions of the MAMA protocol.

```
unsigned int _df() const { return m_ValueFlag; };
```

The operation determines the access permissions according to the specification of the MAMA protocol. The default status after the class is instantiated is *0* (*inconsistent*).

```
operator const char*() const;
```

This is a cast operator to a string (*char**) for the value field.

```
operator const unsigned long() const;
```

This is a cast operator to a string (*unsigned long*) for the value field.

```
operator const IDLNamedValue() const;
```

This is a cast operator to the IDL data type.

```
bool operator==(SWNamedValue& swnv) const;
```

This is an operator that enables two name-value pairs to be compared. It returns *true* when all members are identical, *false* otherwise.

4.3.3.2. SWOptionsList

This class is the most complex class in the API. It represents the implementation of an NVL. Most of the data structures handled by the API are given as NVL. The class *SWOptionsList* adds a key element to a name-value pair in order to simplify the maintenance of NVLs. The class is able to handle OMG IDL typed lists as well as the basic name-value list defined by the MAMA protocol.

Synopsis

```
#include <SWStandard/SWOptionsList.h>
SWOptionsList opl;
```

```
typedef RWTValMap<RWCString, SWNamedValue, std::less<RWCString> >
        SWNAMEDVALUEMAP;
```

Description

The main task of this class is to represent the *IDLOptionsList* in C++. This map has a key and value field. The value is an instance of the class *SWNamedValue*. The key depends on the name field of the *SWNamedValue*.

Usage

This class is used to create a structure for combining large amounts of name-value lists in one map. It is the most important structure for exchanging data. The class comprises methods for persistence. These methods allow writing the actual state of an instance of this class to a file or a stream.

Members

```
void changeNV(SWNamedValue& nv);
```

Changes the current entry to the given parameter *nv*. This will change all information in this entry to the information contained in the parameter *nv*.

```
bool concat(SWOptionsList& oplist);
```

Concatenate the current and the given map (*oplist*) to one map. The sequence of the entries may have changed after this operation.

```
bool contains(const RWCString& str);
```

Return true when the parameter *str* matches a key in the map.

```
SWNamedValue current();
```

Return the current entry as *SWNamedValue*.

```
IDLNamedValue currentIDL();
```

Return the current entry in form of an *IDLNamedValue*.

```
unsigned long entries() { return m_oplist.entries(); };
```

Return the number of entries that are present in the map.

```
bool getNext(IDLNamedValue& IDLnv);
```

Change the parameter *IDLnv* to the next entry in the map. Return *false* when the map is empty or the current key is the last available.

```
bool getNext(SWNamedValue& nv);
```

Change the parameter *nv* to the next entry in the map. Return *false* when the map is empty or the current key is the last available.

```
bool getNextKey(RWCString& str);
```

Change the parameter *str* to the next key in the map. Return *false* when the iterator is currently at the end of the map or the map is empty.

```
bool insert(const IDLNamedValue& nv);
```

Insert the given parameter *nv* to the map. Return *false* when the key is already present.

```
bool insert(const SWNamedValue& nv);
```

Insert the given parameter *nv* to the map. Return *false* when the key is already present.

```
bool next();
```

Increase the internal iterator by one. Returns *false* when the current value is the last in the map or the map is empty.

```
bool remove(const RWCString& str);
```

Remove the *SWNamedValue* matching *str*. Returns *false* when no matching key was found or the map is empty.

```
void reset();
```

Reset the internal iterator for the current map and set it to the first entry.

```
void showAll();
```

This operation simply streams all entries with key and value to the standard output device *stdout*.

```
operator const IDLSeqNamedValue();
```

Cast the current map to *IDLSeqNamedValue*.

```
SWNamedValue operator[](RWCString str);
```

An operator to return the matching key *str* as *SWNamedValue*.

4.3.3.3. SWOperationMap

The API has to process all incoming requests of clients. Each request must specify which action is called. The API is in the position to forward the request to the actual implementation of the action. The general mechanism is explained in section 3.4.2.3.

The class *SWOperationMap* represents the global database of all operations that a client has announced to the API as available implementations of actions. This class should be instantiated only once per application. The C++ port of the API handles the operations in form of a list of pointers. This class realizes the complete maintenance of this list.

Synopsis

```
#include <SWStandard/SWOperationMap.h>
SWOperationMap operationmap;

// type definition of the function pointer
typedef SWReturnList* (SWOperationMap::*PSWOPERATION)(const
                       SWArgStruct& argstruct);

// type definition for the operation's map
typedef RWTValMap< RWCString, PSWOPERATION, std::less<RWCString>
                 > SWOPERATIONMAP;
```

Description

This is the most powerful map of each MAMA application. It is the representation for registered operations accessible by other applications. Standard operations for each application are automatically inserted in this list.

This class has two maps, one for the function pointers and one for their description. The information is divided because C++ offers no mechanism to define structures that include function pointers as members.

Usage

The class *SWAPI* is responsible for instantiate this map. The developer uses the operation *addNewOperation* to register new operations.

Members

```
PSWOPERATION operator[](const RWCString& strOP);
```

Return a pointer to the operation with the name depicted by the parameter *strOP*.

```
bool contains(const RWCString& strOP) const;
```

Return *true* when the parameter *strOP* matches an operation, *false* otherwise.

```
RWCString getNext();
```

Return the name of the next operation in the map. The string is empty when no more operations are available.

```
RWCString getCurrent() const;
```

Return the current key.

```
RWCString getCurrentDescription() const;
```

Return the description of the currently selected operation, if available.

```
void reset();
```

Reset the internal iterator.

```
void insert(const RWCString& key, const PSWOPERATION& pop,
            const RWCString& descr);
```

Insert the parameterized operation. An existing operation with the same name will be overwritten.

```
unsigned long entries() const;
```

Return the number of available operations.

```
void showAll();
```

This function simply streams all entries with key and value to the standard output device *stdout*.

```
SWNamedValue list(const RWCString& operation);
```

Return the name and the description of the operation depicted by the parameter *operation*.

```
SWReturnList* listAll();
```

Return a list of all associated operations.

4.3.3.4. SWAddressList

This class handles addresses of MAMA applications according to the definitions of the MAMA protocol (cf. section 3.4.1.2).

Synopsis

```
#include <SWStandard/SWAddressList.h>
SWAddressList addrList;
typedef RWTValSlist< SWObjectPath > SWADDRESSLIST;
```

Description

This class is responsible for the address handling. It represents the implementation of *IDLSeqObjectPath*. Each application which is addressed is represented by one *SWObjectPath*.

Usage

This class is used by the API according to the definitions of the MAMA protocol. The term *self* in the following descriptions of member functions relates to the application itself.

Members

```
unsigned long entries(void);
```

Return the current count of the list.

```
SWObjectPath getFirst();
```

Return the first existing path of *self*.

```
SWObjectPath removeFirst();
```

Return the first element of *self* and remove it.

```
void operator=( const IDLSeqObjectPath& );
```

Add the given *IDLSeqObjectPath* to the *self*.

```
operator const IDLSeqObjectPath();
```

Cast operator to convert *self* to *IDLSeqObjectPath*. This is the common cast operator to map a given path to IDL.

4.3.3.5. SWObjectPath

This class handles the address of a single MAMA application.

Synopsis

```
#include <SWStandard/SWObjectPath.h>
SWObjectPath opPath;
```

Description

An object path is the address to a MAMA application. This includes the support for addressing objects in a hierarchy (cf. section 3.4.3). Here, the path is a concatenation of each address an action has to be forwarded to through the hierarchical tree.

Usage

This class is used by the API to realize management hierarchies. The term *self* in the following descriptions of member functions relates to the application itself.

Members

```
RWCString decrementPAth(void):
```

Return the first element in *self* and remove it.

```
operator const RWCString() const;
```

Cast operator for *RWCString*.

```
operator const char*() const;
```

Cast operator for *char**.

4.3.3.6. SWError

This class deals with runtime errors. Applications can specify the policy for the error handling on startup. Errors are handled like events.

Synopsis

```
#include <SWStandard/SWError.h>
SWError swError;
typedef RWTValSortedDlist<SWNamedValue, std::less<SWNamedValue> >
        SWERRORMAP;
```

Description

This class is responsible for handling errors occurring at runtime. The map *SWERRORMAP* stores all errors locally. The maximum number of stored errors can be configured.

Usage

Each error is represented as an *SWNamedValue*. The map, which contains all errors, is a sorted list.

Members

```
void setServerName(const RWCString& ServerName);
```

Set the name of the server errors should be sent to. Usually, this is the event service.

```
bool setTransmitMode(const bool transmit);
```

When *true*, all errors will be sent immediately. When *false*, errors will be stored in the map.

```
void newError(const SWNamedValue& swnv);
```

Generate a new item.

```
SWNamedValue lastError() const;
```

Return the last error in the map.

```
SWReturnList listErrors();
```

Return a list of all stored errors.

```
void showErrors();
```

Send all errors from the map to the standard output device *stdout*.

```
void showLastError() const;
```

Send the last error in the map to the standard output device *stdout*.

```
bool sendLastError(const bool transmit = false) const;
```

Send the last error in the map to a specified server. No action is performed when *transmit* is set to *false*.

4.3.4. Middleware Specific Library

The C++ port of the API provides a skeleton that already includes all operations needed for the communication following the specifications of the Common Object Request Broker Architecture (CORBA). This skeleton is based on developments done for the UMS. The original specifications can be found in [Dutkowski01].

The library includes two classes. The class *SWCORBAServer* is responsible for handling the entire initialization of a CORBA application. The class *SWCORBALib* implements the OMG IDL version of the MAMA interface as described in section 4.2.

4.3.4.1. SWCORBAServer

This class represents the core of the application. It is implemented as a singleton object. Additionally, it implements the *main* function, which serves as entry point for the application. The main function controls the application flow and calls specific operations of the instance of the class *SWCORBAServer*. The class includes functionality for the evaluation of command line options, versioning, and usage information.

Synopsis

```
#include <SWCORBA/SWCORBAServer.h>
SWCORBAServer swCORBAServer;
```

Description

This class is the implementation for all functions and mechanism needed to implement a CORBA application. The *SWCORBAServer* class is included in the static library called *CORBALib*.

Usage

The functionality of this class is completely hidden to the application programmer. All methods performed are redirects from the class *SWCORBALib*. The three member functions, which deal with the execution policies, follow the specification of the MAMA protocol (cf. section 3.4.2.3).

Members

```
void shutdown();
```

Stop the core object, cleanup all subscribed and used resources, and terminate.

```
void visible();
```

When the server has a window for standard output, a call to this function activates and de-activates this window.

```
bool checkLocalExecution(SWArgStruct& actionargs);
```

This method is responsible for validating whether this action is required to run at this application or not.

```
bool checkForwardExecution(SWArgStruct& actionargs);
```

This method checks whether the action should be forwarded to all associated applications.

```
SWReturnList ForwardExecution(SWArgStruct& actionargs);
```

This method forwards the execution of the requested action to all associated applications. It takes care to change the addresses and response back to the callee.

4.3.4.2. SWCORBALib

This class implements the MAMA interface (cf. section 3.4.1 for ADL and section for 4.2 IDL specifications). The implementation supports two directions. First, it receives requests from the CORBA interface and invokes the proper implemented operation. Second, it can be used by the application object to invoke operations on other objects.

When an application calls an action of another application, this class resolves the addressed application and requests the called actions via the IDL interface.

Synopsis

```
#include <SWCORBA/SWCORBALib.h>
SWCORBALib swCORBALib;
```

Description

This class redirects all incoming requests and initialization processes to the implemented operations.

Usage

When an action on another application should be called, this class can be used to realize the appropriate CORBA processing.

Members

```
int Initiate(const Management_ptr TIE);
```

Initializes the class instance. In fact, it binds the given TIE pointer to the IDL interface.

```
int Action(SWArgStruct& ArgStruct, SWReturnList& ReturnList);
```

This method is responsible for requesting an action on other applications. It checks the parameter *ArgStruct* for valid data about the called action, the address of the application, and parameters of the called action. The parameter *ReturnList* contains all return values for the given action.

4.3.5. Building an Application with the API

The above described C++ port of the API supports the implementation of MAMA applications. Each application needs to declare one class that inherits from the middleware stub. The follow code shows the skeleton for a MAMA application, including the definitions for the CORBA server. The example class is implemented as a singleton object:

```
#include <CORBA/SWCORBAServer.h>
#include <Util/TSingleton.h>

class ExampleApplication : public SWCORBAServer
{
  public:
    ExampleApplication();   // default constructor
    ~ExampleApplication();  // default destructor

    virtual int OnInitServer( int argc, char *argv[], char *envp[]);
    virtual void OnExitServer();
};

DEF_TIE_SW_Management(ExampleApplication);
typedef TSingleton<ExampleApplication> theExampleApplication;
```

Each application must declare and implement the two operations *OnInitServer* and *OnExitServer*. These operations are called by the class *SWCORBAServer* to realize functionality similar to the standard constructor and destructor of a C++ object. Within these operations, each component can initiate application specific structures, load and store files to accomplish persistence of complex structures, etc.

4.3.5.1. Uniform Signature of Operations

An operation represents the implementation of an action that is specified in ADL. Since operations are called via the MAMA IDL interface, the signature of the operations is identical. The API profits from this fact. It implements a generic function that can handle all actions. In order to support the API, operations need to be declared in a uniform way:

```
typedef SWReturnList* OperationName(SWOperationMap::*PSWOPERATION)(const
                                    SWArgStruct& argstruct);
```

This declaration generates an operation of type *SWOperationMap::*SWOPERATION* with *argstruct* as argument. The return value is a pointer to *SWReturnList*. Each application must inherit from the class *SWOperationMap* and follow this declaration. The argument of the operation is realized as a structure as follows:

```
typedef SWOptionsList SWParametersList, SWReturnList;
struct SWArgStruct{
  RWCString OperationName;
  SWAddressList AddressList;
  SWParametersList ParametersList;
  SWOptionsList OptionsList;
};
```

The member *OperationName* is a string that must be unique within the operation map. The member *AddressList* is a list of all applications. The member *ParameterList* is an NVL that comprises all parameters of the operation as specified in ADL for the regarding action. Finally, the member *OptionsList* can be used to transmit specific options for the execution.

To simplify the usage of *SWArgStruct*, all map definitions are of the type *SWOptionsList*. This provides a similar behavior and usage for all elements. Usually, an operation should only look in the parameter list for given arguments. This list can be further processed and returned as *SWReturnList*.

4.3.5.2. Declaration of new Operations

When an application wants to provide its own operations to other applications, it simply creates a class that contains all declaration of these operations. The following example shows such a class:

```
#include <Std/SWOptionsList.h>
#include <Std/SWOperationMap.h>
#include <Util/TSingleton.h>

class SWTestOperations : public SWOperationMap{
  public:
    SWReturnList* checkIn(const SWArgStruct& actionargs);
    SWReturnList* checkOut(const SWArgStruct& actionargs);
    SWReturnList* listOperation(const SWArgStruct& actionargs);
    SWReturnList* updateOperation(const SWArgStruct& actionargs);
    SWReturnList* getServer(const SWArgStruct& actionargs);
    SWReturnList* getServerList(const SWArgStruct& actionargs);
};
typedef TSingleton<SWTestOperations> the SWTestOperations;
```

As the example shows, the object *SWTestOperations* is implemented as singleton, too. The object uses *SWOperationsMap* as base class in order to delegate the maintenance of the list of all operations to the API. All operations have the same type (*PSWOPERATION*), which is used as the generic function pointer to the operations.

4.3.5.3. Register Operations with the API

The access to self-defined operations is only possible through the API and the object *SWOperationMap*. Operations that should be registered at the APIs handling must be present in the operation map. The following code sample shows the process of a started component to add new operations.

```
swapi->addNewOperation("checkIn",
(PSWOPERATION)theSDSOperations::Instance()->getServer, "login@DNSS");

swapi->addNewOperation("checkOut",
(PSWOPERATION)theSDSOperations::Instance()->checkIn, "logout from DNSS");
```

In the example, the two operations *checkIn* and *checkOut* are registered with the API via the API operation *addNewOperation*. The API checks its internal map and stores the function pointers in this map when no operation with the same name is already registered. The last parameter is used for the description map.

At least, only three steps are necessary to register a new operation. First, a class needs to be defined which inherits from the class *SWOperationMap*. Next, the operation is declared within this class. All operations are of the type *PSWOPERATION* and must follow the uniform signature. Finally, the API operation *addNewOperation* must be called.

4.4. Directory Naming and Specification Service

The Directory Naming and Specification Service (DNSS) has been implemented following the definitions described in section 3.6.1. The implementation design is based on the object-oriented paradigm. The DNSS server is constructed by a set of components. Each component has a significant task. However, the design of the implementation is open for the integration of new components in order to enhance its functionality or to improve its behavior. The design and the components are described in the following sections. Specifications are presented in Java.

4.4.1. Implementation Design

The *DNSSModel* is the core component of the DNSS system. It provides a uniform interface to access all services. Furthermore, this model uses the implementation of the MAMA interface (MAMA protocol and API) to enable access from MAMA application objects. The *DNSSModel* manages all sub-systems of the DNSS.

The *DirectoryModel* and the *SpecificationModel* are responsible for processing service requests. The DNSS allows transparent access to directory data and to specification data. The implementation of the DNSS uses two different information trees to store this data. The Data Information Tree (DIT) is handled by the *DirectoryModel*. It contains information in eXchange Directory Data (xDD) format. The Specification Information Tree (SIT) is maintained by the *SpecificationModel*. It contains information in xADL format.

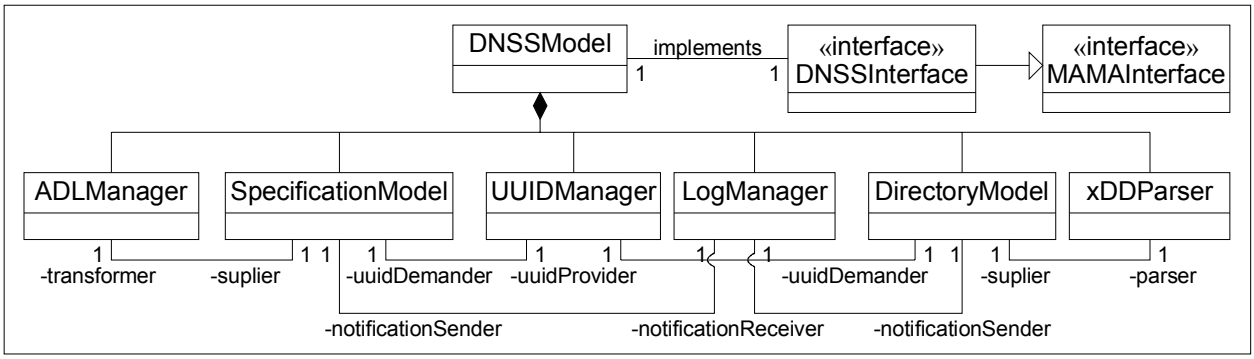

Figure 4-2: DNSS – Class Diagramm of the DNSS Model [Singh01]

The models are supported by an *ADL Manager*, an *xDD Parser*, a *Log Manager*, and a *UUID Manager*. Figure 4-2 shows the class diagram of the DNSS implementation with all sub-systems and their relationships.

4.4.2. Directory Service Sequence Diagrams

The following sequence diagrams show the realization of the use cases of the directory model of the DNSS (cf. section 3.6.1.4). The operations used in the sequence diagrams are specified in the Directory Interface (cf. section 3.6.1.5).

The following descriptions use a simplified terminology to identify the type of entry in the DIT. Directory entries are called directory entries. They are completely managed by the DNSS. Directory instance entries are called instance entries. Directory alias entries are called alias entries. The term *entry* is used when instance entries and alias entries are depicted.

4.4.2.1. Retrieval of Directory Entries

The client can search and retrieve entries. Two operations are provided. The first operation returns information on a single entry. It is parameterized with the Distinguished Name (DN) of this directory entry and the type of the entry. The second operation returns a list of all entries that belong to a certain naming context. Filter parameters can be used to restrict the list. They are described in section 3.6.1.5.

```
string getEntry(string entryDN, integer entryType);
string getEntries(string parentDN, integer entryType, boolean recursive,
                  integer fromIndex, integer toIndex);
```

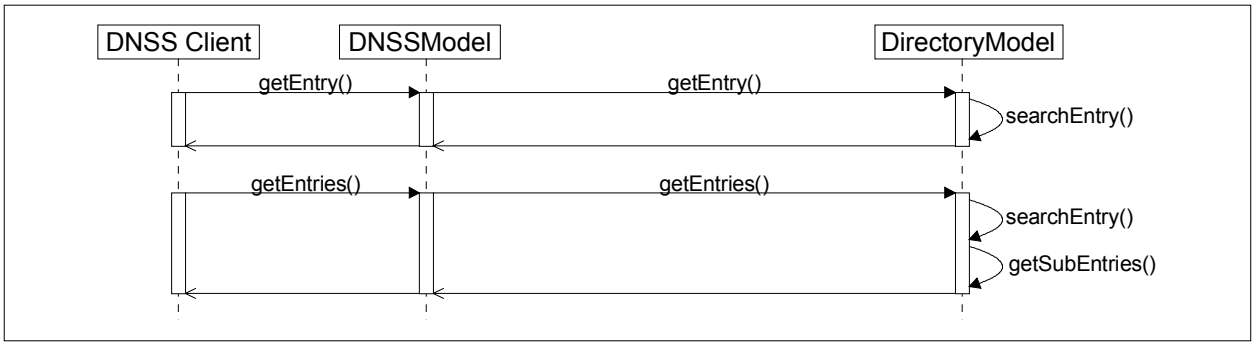

Figure 4-3: DNSS – Entry Lookup

Figure 4-3 shows a sequence diagram including both operations. The parameter *parentDN* of the operation *getEntries* specifies the root node from which the entry search should be started. The parameter *entryType* specifies the type of the entries that should be processed. The flag *recursive* indicates whether the operation should be performed on all sub-nodes of *parentDN* or not. Both operations involve the classes *DNSSModel* and *DirectoryModel*.

4.4.2.2. Registration of new Directory Entries

The registration of new entries in the DIT comprises the registration of instance entries and alias entries. For both use cases, the DNSS checks if the entry already exists in the DIT. The registration is only done, when no entry exists with the requested DN. Following this approach, existing entries cannot be overwritten with a new registration.

The registration of a new instance entry expects parameters for the DN the entry should be assigned to and the element type of the entry (e.g. module, object). Furthermore, the client should provide the specification of the element in xADL or ADL format. When no specification is supplied by the client, the specification reference of the new instance entry points to the *default* specification.

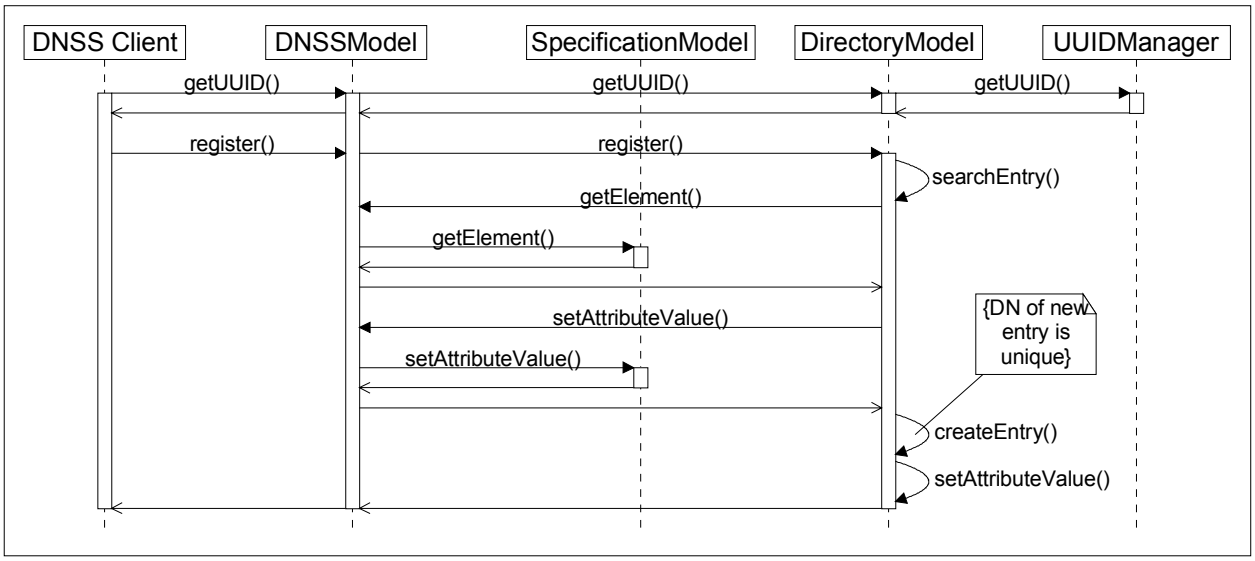

Figure 4-4: DNSS – Registration of a new Directory Entry [Singh01]

Figure 4-4 shows the sequence diagram for the registration of a new instance entry. The first step for the client is to obtain a UUID from the DNSS server. This UUID is assigned as a private attribute. The following operations are involved in this process:

```
string getElement(string entryDN, integer entryType,
                  integer formatType, boolean compact);
boolean setAttributeValue(string entryDN, integer entryType,
```

```
                            string attributeName, string[] attributeValue,
                            string uuid);
boolean register(string instanceDN, string reference, string referenceKind,
                 string objectDN, string uuid);
```

The operation comprises the classes *DNSSModel*, *DNSSModel*, and *SpecificationModel*. Additionally, the *UUIDManager* is requested to supply a new UUID for the client. Clients which have already a UUID do not need to request a new one.

The registration of a new alias entry is shown in Figure 4-5. The client must request a valid UUID for this operation, too. The registration itself is similar to the registration of a new instance entry.

```
boolean registerAlias(string aliasDN, string instanceDN, string uuid);
```

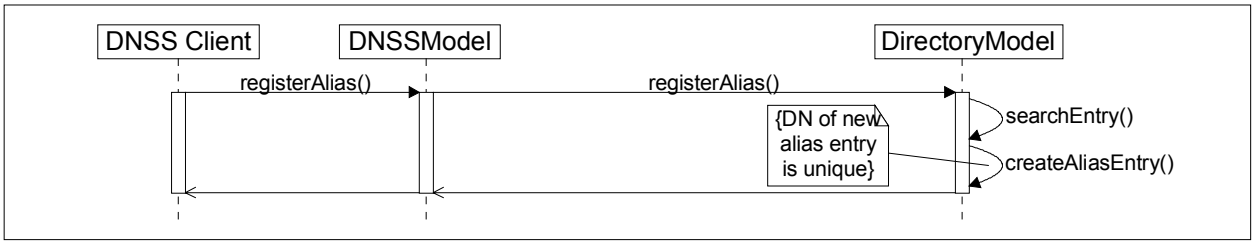

Figure 4-5: DNSS – Registration of a new Alias Entry

The operation *registerAlias* expects the DN of the new alias entry and the DN of the instance entry that should be aliased. The operation involves the classes *DNSSModel*, *DirectoryModel*, and *UUIDManager* (if the client needs to request a UUID).

4.4.2.3. Deregistration of Directory Entries

Authorized clients can deregister instance entries and alias entries. A deregistration results in the deletion of all associated data, including all sub-nodes of the entry. A client is authorized for this operation when its UUID is valid and the same as the UUID of the entry that should be removed. Only one directory entry can be removed per operation call.

```
boolean deregister(string entryDN, integer entryType, string uuid);
```

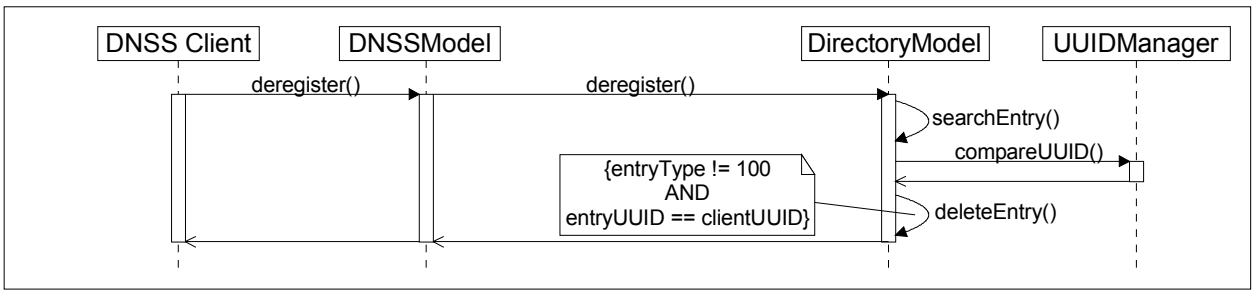

Figure 4-6: DNSS – Deregistration of a Directory Entry

Figure 4-6 shows the sequence diagram for this operation. The entry type to be deregistered cannot be a directory entry. It must be an instance entry or an alias entry. The operation involves the classes *DNSSModel*, *DirectoryModel*, and *UUIDManager*.

4.4.2.4. Modification of Directory Entries

The DN or the Relative Distinguished Name (RDN) of a registered entry (instance or alias) can be changed. The effect is that an instance entry can be moved from one position (possibly naming context) to

another. The last part of the name must be unique in the DIT. The name of a directory entry cannot be changed.

```
boolean modifyEntryName(string oldEntryDN, integer entryType,
                        string newEntryDN, string uuid);
```

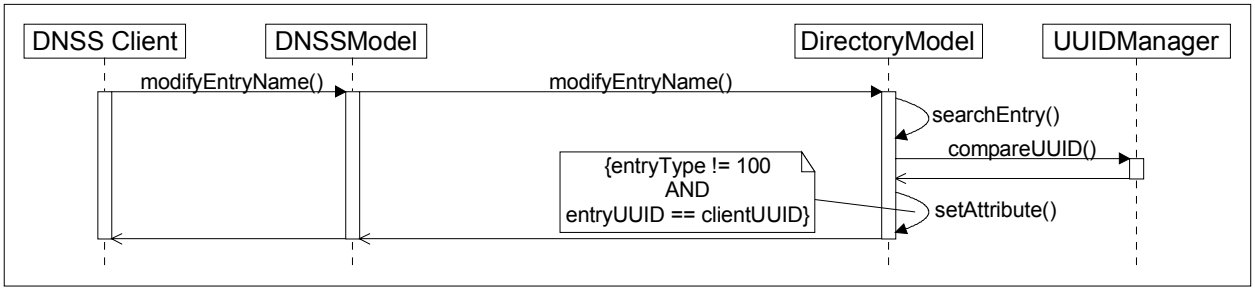

Figure 4-7: DNSS – Modification of Directory Names [Singh01]

The client must be authorized in order to process the modification. The given client's UUID and the UUID of the entry to be modified must be identical. This operation involves the classes *DNSSModel*, *DirectoryModel*, and *UUIDManager*.

4.4.2.5. Retrieval and Manipulation of Attributes

Figure 4-8 shows the two use cases that allow clients to access and to alter attributes of entries. Every attribute, except the *uuid* attribute, can be read out and manipulated. For both operations, the client must specify the DN and the attribute name of the entry. The operations can be employed for the directory model **and** the specification model. The DNSS checks the element type and decides which model is responsible for processing the request.

```
string getAttributeValue(string eDN, integer eType,
                         string attributeName);
boolean setAttributeValue(string eDN, integer eType,
                          string attributeName, string[] attributeValue,
                          string uuid);
```

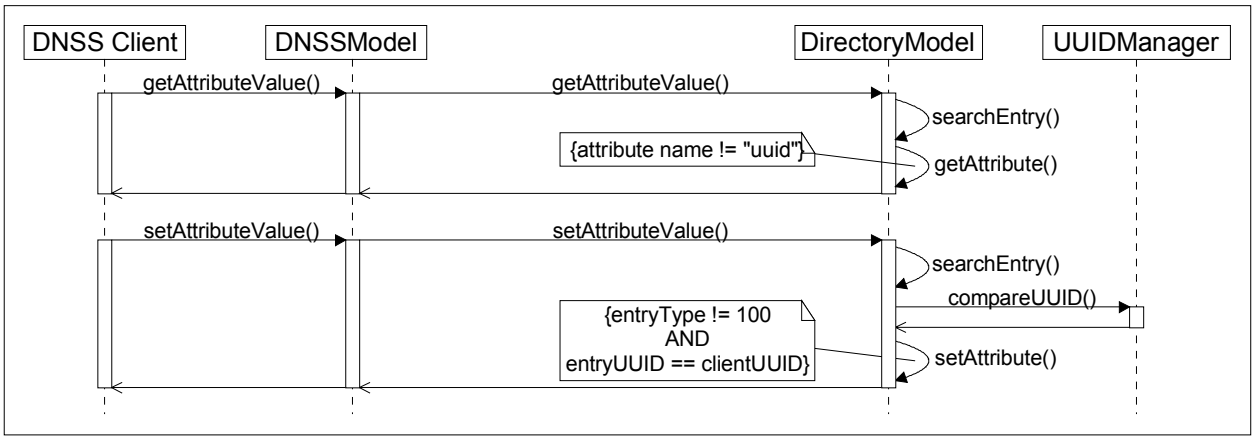

Figure 4-8: DNSS – Retrieval and Manipulation of Attributes [Singh01]

For the modification of attributes, the client's UUID must be the same as the UUID of the entry the attribute belongs to. The *UUIDManager* is employed to check the UUIDs. Both operations involve the classes *DNSSModel*, *DirectoryModel*, and *UUIDManager*. When specification elements should be processed, the class *SpecificationModel* is involved instead of *DirectoryModel*.

4.4.2.6. Retrieval of Object Specifications

A client can obtain the specification of a registered entry. In this use case, the *SpecificationModel* is also engaged, because the specification of the object is stored in the SIT. The client needs to provide the DN of the instance entry or alias entry.

```
string getObjectSpec(string instanceDN, integer formatType);
```

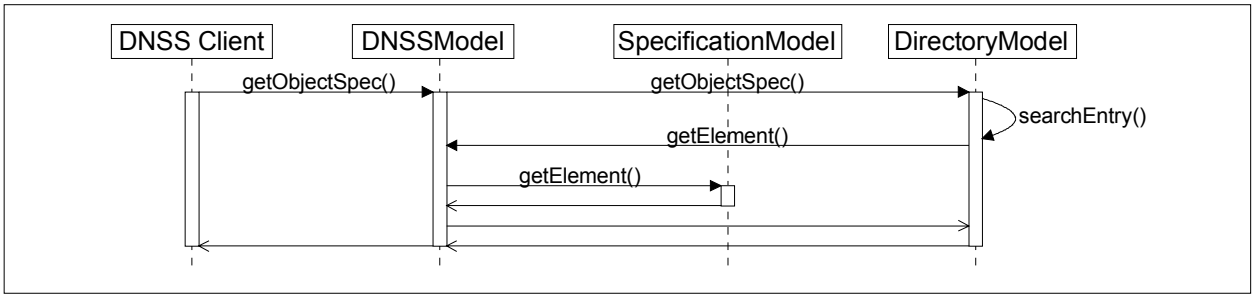

Figure 4-9: DNSS – Retrieval of Object Specifications [Singh01]

This use case involves the classes *DNSSModel*, *DirectoryModel*, and *SpecificationModel*. The entry type cannot be a directory entry. It must be an instance entry or an alias entry.

4.4.3. Specification Service Sequence Diagrams

The following sequence diagrams show the realization of the use cases of the specification model of the DNSS (cf. section 3.6.1.6). The operations used in the sequence diagrams are specified in the Directory Interface (cf. section 3.6.1.7).

4.4.3.1. Element Retrieval including Filtering and Scoping

The DNSS supports a number of operations to retrieve information about specification elements. All operations expect the DN of the specification element.

Furthermore, the element type should be identified. For all operations, the format of the returned string can be configured. This format can be either compact or complex. Compact means that only requested information about the specification element is returned. Complex means that the information can also include applied qualifiers and child elements. Additionally, the presentation format of the string can be selected. Permitted formats are ADL and xADL.

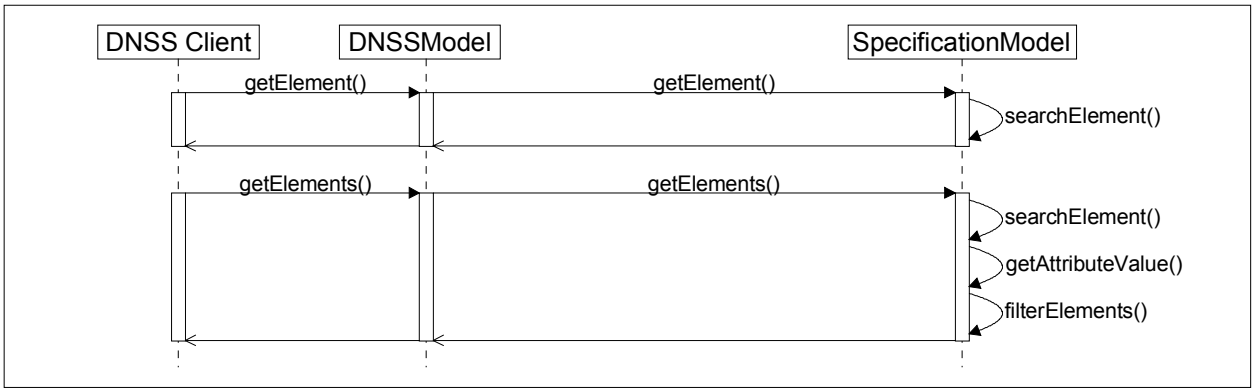

Figure 4-10: DNSS – Element Retrieval including Filtering and Scoping

The first operation provides the simple retrieval of a specification element. This operation involves the classes *DNSSModel* and *SpecificationModel* (first sequence in Figure 4-10).

```
string getElement(string elementDN, integer elementType,
                  integer formatType, boolean compact);
```

Two other operations allow selecting specific sub-elements including filtering and scoping. The operation *getElements* returns child elements that belong to a specified parent element. The operation *getElementsByValue* allows retrieving all elements filtered by the existence of a specific attribute. Additional filter parameters are the value of the attribute and the element type (cf. section 3.3.3.4). The flag *recursive* is added to the operation *getElements* to limit the amount of processed data. The specification is searched recursively when this flag is set to *true*, only.

Both operations support parameters for scoping and for specifying the format of the returned string. Scoping is realized with the parameters *fromIndex* and *toIndex* as introduced in section 3.6.1.7.

```
string getElements(string parentDN, integer elementType, boolean recursive,
                   integer fromIndex, integer toIndex,
                   integer formatTyp, boolean compact);

string getElementsByValue(string attributeName, string attributeValue,
                          integer elementType, integer fromIndex,
                          integer toIndex, integer formatType,
                          boolean compact);
```

The operation *getAttributeValues* can be employed to retrieve attribute values. These values can be further used as input parameters for *getElementsByValue*.

```
string getAttributeValue(string elementDN, integer elementType,
                         string attributeName);
```

The second sequence of Figure 4-10 shows an example for filtering and scoping. A client calls the operation *getElements* to receive a list of available specification elements. Next, the operation *getAttributeValues* can be used to read out the value of a certain attribute.

Finally, the client can invoke *getElementsByValue* to receive a list of all elements (possibly of the same type) that have the same attribute with the same value. Furthermore, the list of elements can be scoped in order to receive i.e. only the five first elements that apply to the filter parameters.

4.4.3.2. Insertion of Specifications and Elements

Figure 4-11 shows two use cases for the insertion of new specifications and new elements in the SIT. The first use case enables clients to add a new specification. The first step for the client is to obtain a new UUID from the DNSS. This UUID is assigned as a private attribute to all data of the specification. When a client sends a new specification, the DNSS checks whether the specification exists already or not. An existing specification cannot be overwritten with the operation *addSpecification*.

```
boolean addSpecification(string specificationName,
                         string specification, string uuid);
```

The operation comprises the classes *DNSSModel* and *SpecificationModel*. Additionally, the *UUIDManager* is requested to supply a new UUID for the client. Clients which have already a UUID do not need to request a new one.

The second sequence depicted by Figure 4-11 provides an operation to add new elements to a pre-existing specification. An existing specification can be modified in several ways:

- Add a new element to the specification.
- Modify an existing element. The modification is not done by the DNSS. Instead, the client must first remove the particular element and than add a new element.
- Modify parts of an element by changing the values of its attributes.

```
boolean addToSpecification(string newElement, integer elementType,
                           string parentDN, string uuid);
```

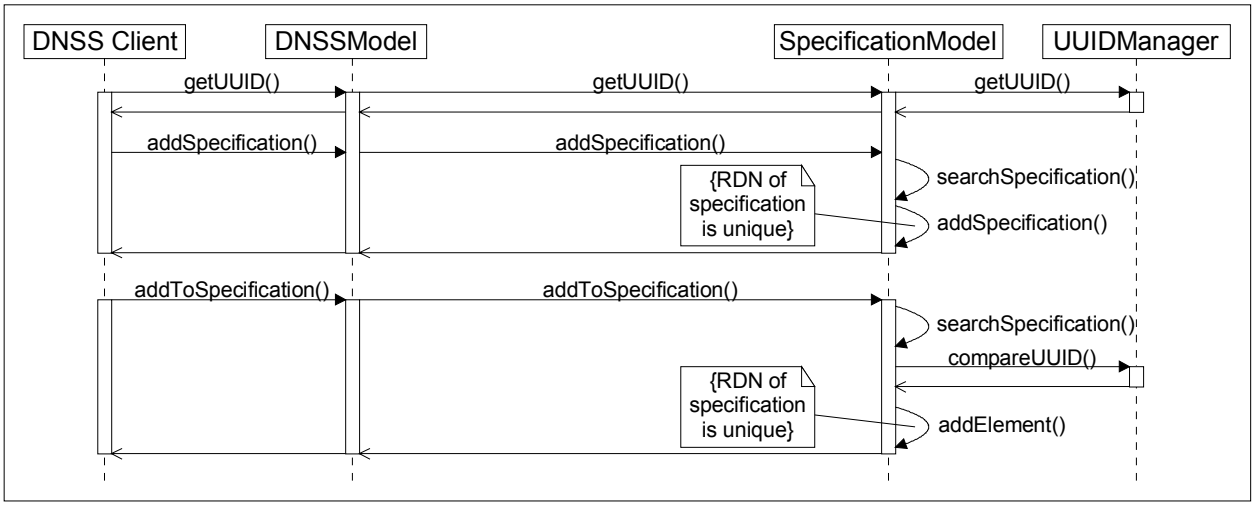

Figure 4-11: DNSS – Insertion of Specifications and Elements [Singh01]

This operation involves the classes *DNSSModel*, *SpecificationModel*, and *UUIDManager*. The UUIDs of the client and the specification must be the same. The element type can be '1', '2', '7', or '8' according to the specification of *tElementType* in section 3.3.3.4.

4.4.3.3. Remove a Specification Element

This use case allows removing specification elements from the SIT. When an object class is removed, the references of all object instances to this object class are redirected to the default object specification.

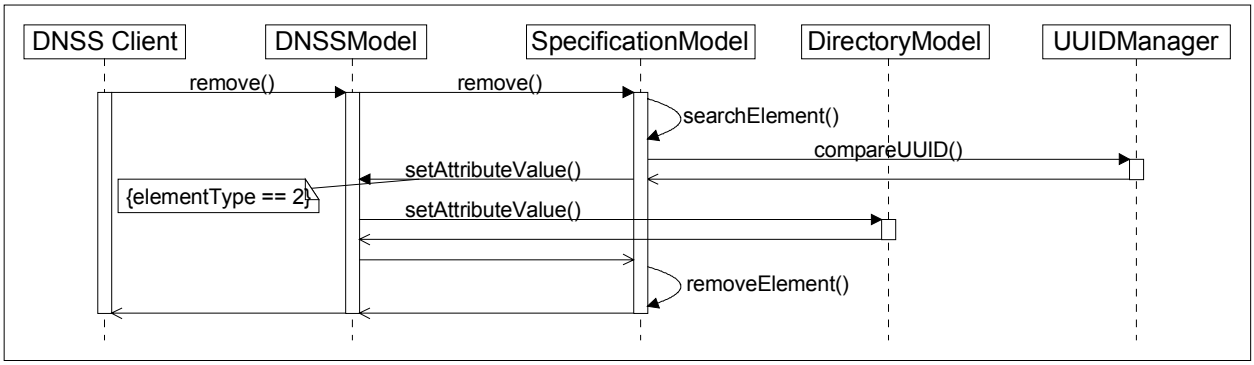

Figure 4-12: DNSS – Remove a Specification Element [Singh01]

Figure 4-12 shows the sequences of operations that are involved. First, the element is searched. Next, the UUIDs of the client and the element are compared. When the UUIDs are equal and the specification element is of type object, the directory model is requested to rearrange the references of the object instances. Finally, the element is removed from the SIT. The DNSS provides no reversal of this operation.

```
boolean remove(string elementDN, integer elementType, string uuid);
```

The operation involves the classes *DNSSModel* and *SpecificationModel*. The only constraint is that the UUIDs of the requesting client and the entry that should be removed are identical.

4.4.3.4. Retrieval of Object Instances

A client can search for every registered object instance of a known object class. The specification model searches for the object class and employs the directory model to collect all registered object instances, as shown in Figure 4-13. Obviously, instances can only be collected when they exist.

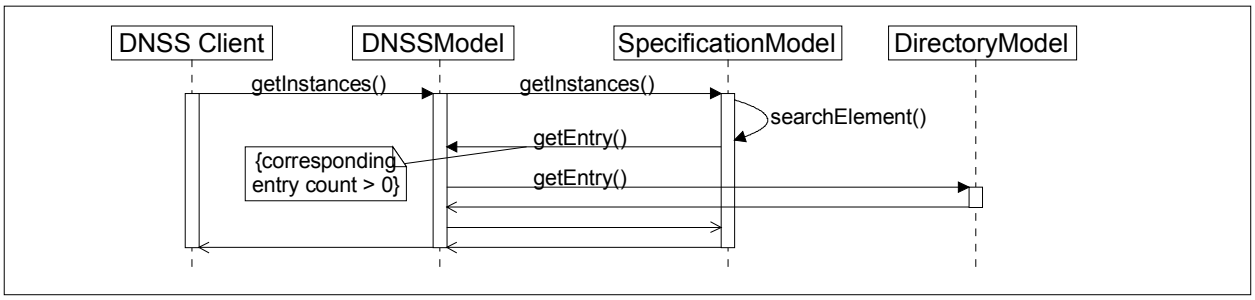

Figure 4-13: DNSS – Retrieval of Object Instances [Singh01]

Two operations are involved in this processing.

```
string getEntry(string entryDN, integer entryType);
string getInstances(string objectDN);
```

The retrieval of object instances involves, beside a DNSS client, the implementation classes *DNSSModel*, *SpecificationModel*, and *DirectoryModel*.

4.4.4. ADL Manager

Specification objects are defined in ADL or xADL format. The *ADL Manager* is the component that controls the format of input streams and output streams. Figure 4-14 shows the implementation classes of the ADL Manager. An *ADL Parser* represents a composite part in this class diagram. The ADL Manager acts as xADL supplier for the ADL Parser.

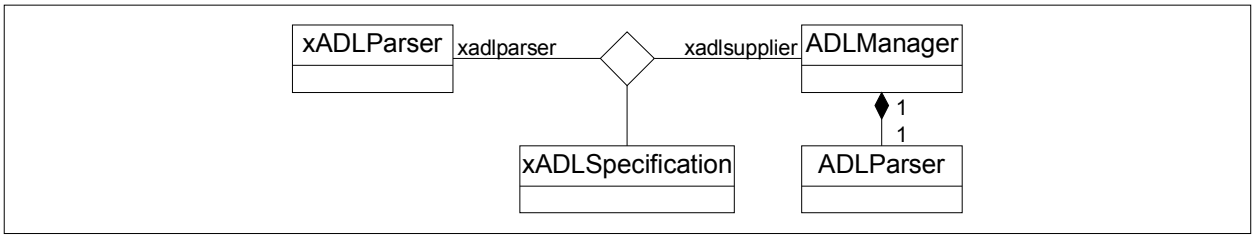

Figure 4-14: DNSS – ADL Manager [Singh01]

Figure 4-15 shows algorithm for handling both specification formats. Specifications are handled in form of strings. One specification is represented by one string, which can by of any length. The ADL Manager examines the specification string and determines its actual format. The evaluation searches for the line that starts an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
```

All valid XML files have to start with this line. When the xADL format is detected, it is forwarded to the xADL parser.

When the XML line was not found, the specification is forwarded to the ADL parser. This parser generates xADL when the specification was given in ADL format. The generated xADL format is than transferred to the xADL parser. In case the ADL parser cannot detect the actual format of the specification, it ignores the given string.

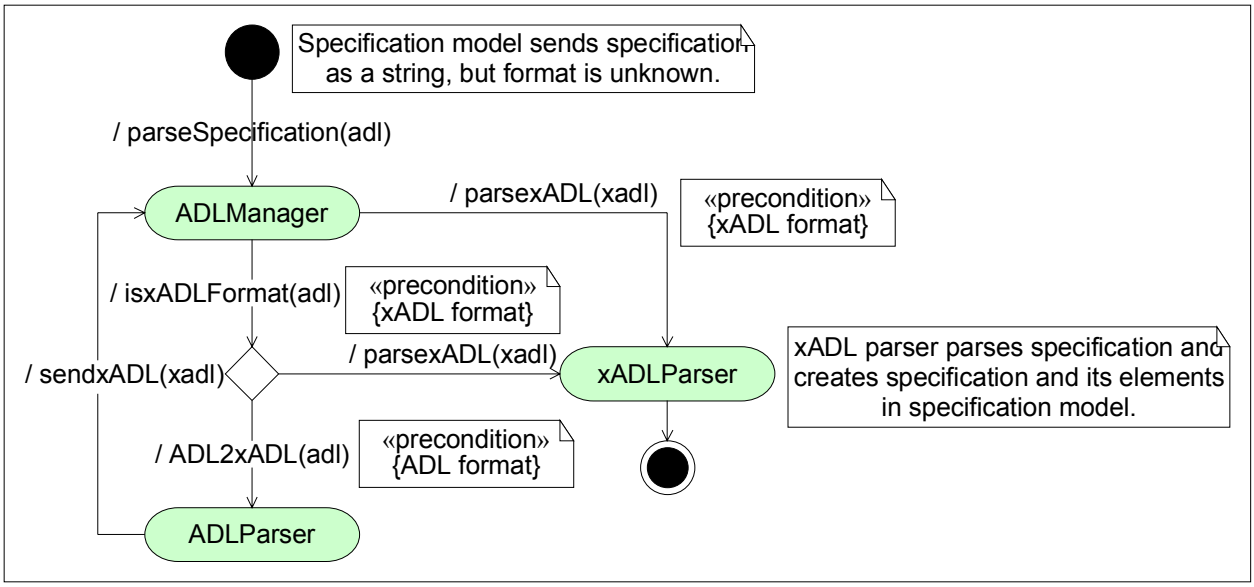

Figure 4-15: DNSS – Processing of ADL formatted Specifications [Singh01]

The ADL Manager is able to supervise conversion in both directions, from xADL to ADL and vice versa. This allows the DNSS to offer both formats to clients. The conversion of xADL to ADL is realized with three steps. First, an object creates its specification in xADL format and sends it to ADL manager. Next, the ADL manager forwards the specification with parameters to the ADL parser. Parameters tell the ADL parser to convert the given specification to ADL format. Finally, the ADL parser recognizes the parameters and transforms xADL-formatted specification in ADL format.

4.4.5. UUID Manager

The UUID Manager assigns a UUID to a directory entry or a specification element. The UUID Manager consists of two key objects, the *UUID constructor* and a *register*. The UUID constructor creates new UUID on demand. The register the maintenance of assigned UUIDs.

A register can be a list, a vector, a hash table or any other data structure. However, it should be able to save, compare, and search the UUIDs. All UUIDs contained in a register are unique. The UUID constructor is a composite part of the UUID Manager that cannot be accessed from outside the DNSS.

The two tasks of the UUID Manager are the creation of new UUIDs and the comparison of UUIDs. A UUID is assigned per client, not per request. Clients that have already obtained a UUID should reuse this UUID for further requests.

The creation of a UUID can be demanded by the *DirectoryModel* or by the *SpecificationModel*. The UUID Manager creates a new instance of the class UUID. A copy of this new UUID is stored in a local register and the UUID is returned to the requester.

The comparison of UUIDs represents the authorization of clients. Authorization is done per request, not per client. The *DirectoryModel* or the *SpecificationModel* request a UUID check. The parameter for this operation is the UUID of the client that wants to add or modify data in the DIT or the SIT. The UUID is confirmed by the UUID Manager when it exists in the local register. When the UUID is not found, the Boolean value of *false* is returned in order to mark the given UUID as not valid. This simple mechanism provides basic security for data manipulation.

4.4.6. Log Manager

The *Log Manager* is responsible for the durability of data in the DNSS. As sub-component of the DNSS, it is a passive component that acts on behalf of the *DirectoryModel* and the *SpecificationModel*. The Log Manager waits for notifications about changed data in the DIT or SIT.

The *DirectoryModel* and the *SpecificationModel* send notifications to the Log Manager, when data has been changed. This is indicated by the operations with the number 1 in Figure 4-16. The Log Manager requests the models to format their data into XML format (xADL or xDD). This XML data is than written to the log files for directory data or specification data.

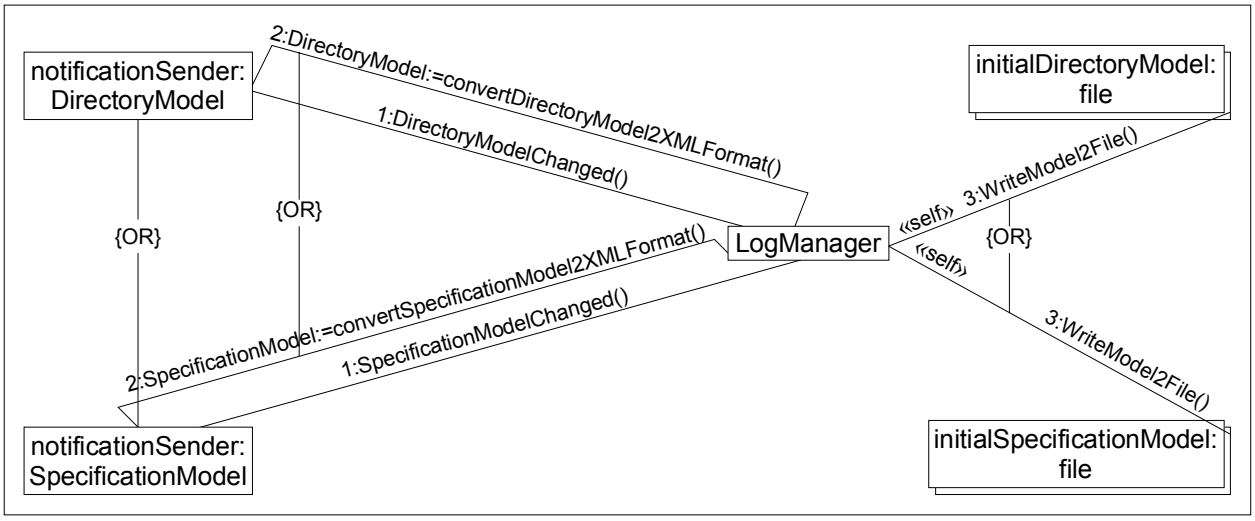

Figure 4-16: DNSS – Tasks of the Log Manager [Singh01]

The data of the directory model is written into one single file. The data of the specification model are further separated into the individual specifications contained in the SIT. Each specification is written into a dedicated file. The Log Manager creates a backup file for each data.

4.4.7. Implementation of the DNSS Server

The DNSS is implemented following the design that is described in the sections above. The employed programming language is Java. Since the implementation design and the programming language Java are object-oriented, the actual implementation of the DNSS follows the object-oriented approach.

Java has been chosen because the Java Software Development Kit (SDK) provides already a wide range of APIs and libraries that can be re-used by the DNSS. Pre-defined classes for hash tables have been utilized to enable local lists of attributes, UUIDs, and persistent data. The implementation was performed with the development tool Visual Age for Java Applications.

4.4.7.1. DNSS Server

The DNSS Server is a single Java main class. It is associated with all other classes. All implemented classes are classified according to their significance within different packages, as shown in Figure 4-17. Each package defines local classes and interfaces to other packages. Each class has been documented with the Java documentation tool. The documentation of a class includes a description of functionality. The individual classes can be characterized as follows:

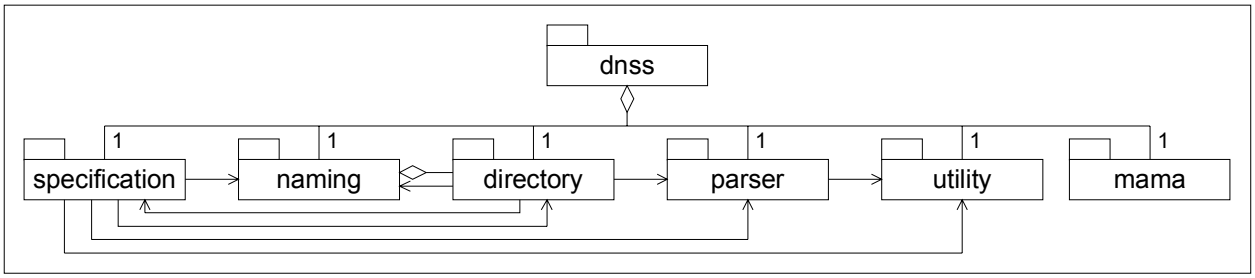

Figure 4-17: DNSS – Java Packages [Singh01]

- dnss: This is the main package of the implementation. It encloses all other source classes in its sub packages. It contains the classes *DNSS.java*, *DNSSModel.java*, and *DNSSInterface.java*. The initial files for specification, directory, and non-java files (e.g. ADL compiler) must be located in the same directory where DNSS server is executed.
- specification: This package contains all classes shown in Figure 3-23, except the class *xADLparaser*. The classes contained in this package depend on classes of other packages like *utility* and *parser*.
- naming: This package comprises the classes that are related to the naming service. The naming classes provide static methods for the naming service. For example, the classes *DirectoryModel* and *SpecificationModel* are supplied with operations for parsing of DNs.
- directory: This package is the sub package of the naming package. It contains all classes depicted in the Figure 3-20, except the class *XDDparser*.
- parser: This includes both the *XADLParser* for the *SpecificationModel* and the *XDDParser* for the *DirectoryModel*. These parsers extend the Simple API for XML (SAX; [SAX]) parser class provided by VisualAge. Both parsers have the ability to parse a file in three input formats: as a string, reading a file from a given location with the File Transfer Protocol (FTP) or the Hypertext Transfer Protocol (HTTP), and a file in a local directory.
- utility: This package contains classes that are used commonly by all packages or by the DNSS server class. E.g., the UUID class is part of this package. It fulfills the requirements prescribed in the 3.6.1.8.
- mama: This package realizes the interface to the Java port of the MAMA API.

The exchange formats (xADL and xDD) are handled by a SAX parser. SAX offers an event-driven, serial access to XML documents. In comparison to the Document Object Model (DOM; [W3C-DOM]) parsers, this characteristic allows a less expensive memory management and a more flexible handling of XML data. The implemented SAX parser creates instances of classes for each XML tag. E.g., the xADL tag *object* is represented by class *SpecificationEntryObject* and each time an *object* tag is identified an instance of this class is created.

All scenarios discussed in the sections 4.4.2 and 4.4.3 have been realized. The implementation supports also the approaches for security (section 3.6.1.8) and the distributed DNSS (section 3.6.1.9). The implementation has the following additional features:

- Searching of objects in the directory model and specification model is based on breadth-first search.
- None of the implemented classes contain an attribute for the distinguished name of an object. In the implementation, the distinguished name is treated as logical name. While parsing the XML documents, the bi-literal relationship (parent-child) between objects are created. They can be changed during runtime of the DNSS. As a result, they are provided to objects only on demand.

The resulting server with its components is depicted by Figure 4-18. The components are realized following the implementation's design (cf. Figure 4-2).

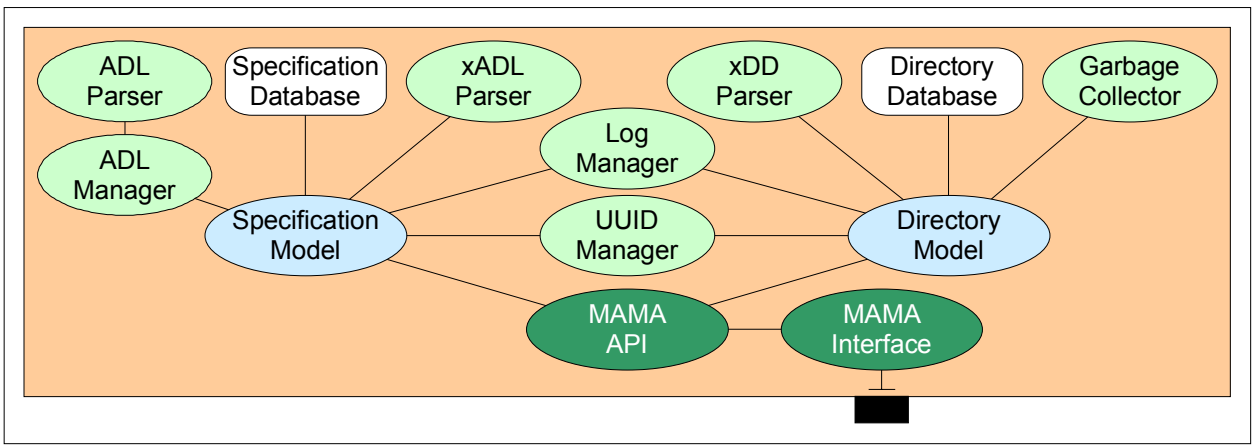

Figure 4-18: DNSS – Server Components

The DNSS supports the MAMA API and the MAMA protocol. Additionally, the DNSS Server can be accessed via Java native interfaces (Java Remote Method Invocation – RMI). The components of the DNSS Server represent instantiated classes of the introduced Java packages. The directory database and the specification database are realized in form of object instances in the server's process and with hash tables. Here, the utilization of real database products such as Oracle or MySQL might improve the performance and stability of the DNSS server.

4.4.7.2. Execution of the DNSS Server

The source code of the DNSS Server and all related classes are compiled and archived in a Java Archive (JAR). The resulting file is called *dnss.jar*. A Java Virtual Machine (VM) is needed on the host where the DNSS is supposed to be executed. Two variants for the execution exist:

```
java -jar DNSS.jar [-cwd <dirpath>] [-iord <dirpath>] [-iordm <dirpath>]
java [Java options] DNSS [-cwd <dirpath>] [-iord <dirpath>] [-iordm <dirpath>]
```

The DNSS accepts a number of command line options. The current working directory is specified with *cwd*. This directory should contain the initial files for DIT and SIT, which are permanently updated by the DNSS Server (Log Manager). The option *iord* instructs the DNSS Server to write its CORBA Interoperable Object Reference (IOR) into a file. This approach realizes that clients can resolve the initial reference to an executed DNSS Server. Finally, the option *iordm* specifies the directory path where the reference to a running DNSS master server is located.

4.4.7.3. Persistence

On startup, the DNSS server reads in two files with initial information. The file *InitialSpecification.xml* contains the initial SIT structure in xADL format. When the DNSS server is started for the first time, this file includes only the default specification. The second file *InitialDirectory.xml* includes the initial DIT in xDD format. This file is empty when the DNSS server is started for the first time.

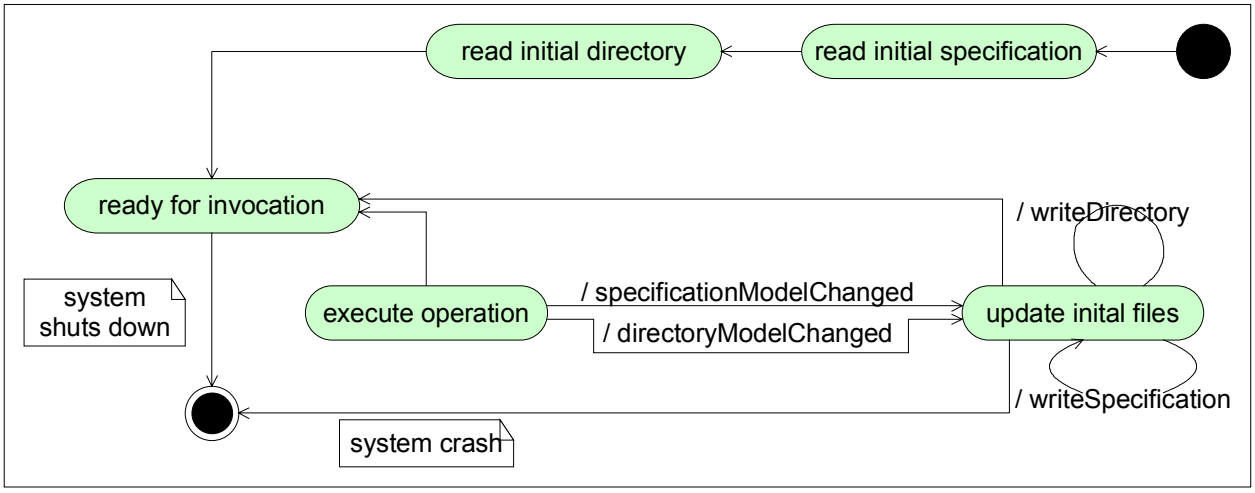

Figure 4-19: DNSS – Persistence Mechanism [Singh01]

When both files are parsed successfully, the DNSS server enters its default state and waits for invocations (cf. Figure 4-19). Each client request is processed and the server returns to the default state. When a client request modifies the DIT or the SIT, the responsible model (directory or specification) sends a notification to the Log Manager. This component receives the notifications and requests the respective model to send its current state in XML format (xADL for specification model and xDD for directory model). The received XML string is than written to the file for initial specifications or directory information.

4.4.7.4. Garbage Collector

Directory entries cannot be removed by clients (only instances and aliases). The Garbage Collector is responsible for checking the DIT every time an instance or alias was deregistered.

The Garbage Collector process is synchronized with the log manager. Whenever a change in directory model takes place, the Garbage Collector starts searching for entries that do not contain sub-entries in order to delete them.

The Garbage Collector is a process started by directory model to clean the directory model. The garbage collector filters the empty directory entries from the directory model and deletes them. A directory entry is empty if it does not contain any instance entry, alias entry, or directory entries.

4.4.7.5. Exceptions

The DNSS employs the MAMA API with the OMG IDL specifications for the interface. Errors that occur during the dynamic execution of operations of the *DirectoryModel* and the *SpecificationModel* are returned in form of exceptions. The exceptions are sent in form of a string. The following exceptions are implemented:

- *NullPointerException*: no class is found which provides invoked method.
- *IllegalArgumentException*: the semantic of parameters is not valid or a non-specified argument is included in the operation call.
- *IllegalAccessException*: the invoked method of the java class is not accessible.
- *InvocationTargetException*: the invoked method did not complete its execution

When the request is completed normally, the return value of the operation is returned to the client. Void operations do not return results, only exceptions.

4.5. XAMAV – The MAMA Visualization Service

This section describes the realization of the MAMA Visualization Service. The implemented user interface is named XAMAV – XML ADL MAMA Visualization Tool. XAMAV follows strictly the recommendation for the visualization of MAMA data as described in section 3.6.2.

XAMAV is split into three frames: Tree, Brain, and Information. The Tree Frame shows the specification data and the directory data according to the sections 3.6.2.2 and 3.6.2.3. The Brain Frame displays the structural filter, the Core Model (cf. section 3.6.2.4), and additional information. The Information Frame contains the detail information. Furthermore, the Console notifies the user about the actual processing steps and about errors that might occur. The console is realized in a separate window.

4.5.1. Implementation Backend

The general backend of this application is the Java Swing package contained in the Java SDK. It provides predefined classes for the user interface, like splitting frames and standard buttons. This functionality is used in the main application window.

4.5.1.1. Backend of the Tree Frame

The specification tree is realized with the *JTree* class. A set of classes around the *JTree* package allows the realization of a standard tree while giving full control about the look and feel.

The information for the specification tree is provided in xADL. There should be the possibility to access the XML content via the local file system for testing and demonstrations. In a second case, the access of the XML files has to be realized via HTTP to communicate with the DNSS remotely. After reading the XML file the content have to be parsed and displayed by the *JTree*.

To enable the access of XML-file content, an XML parser is implemented. Two APIs are described in [SAX] and [W3C-DOM98]. The useful API in that case is DOM, because the whole content of the XML

document is available in one stored document after the parsing process. SAX in distinction from DOM generates events during the parsing process, which have to be resolved. SAX does not provide a document after the parsing process and is not so useful for this application.

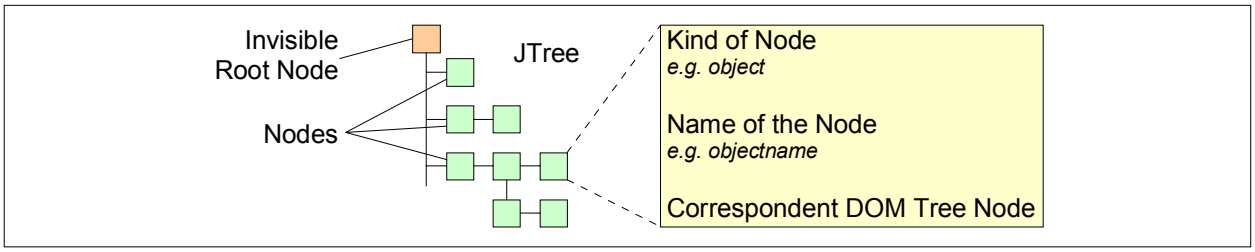

Figure 4-20: XAMAV – Concept for the Specification Tree

The nodes in the DOM tree document contain the data of the XML elements and can be accessed individually. The access and manipulation can be done with interfaces provided by DOM. To connect the XML content to *JTree* (cf. Figure 4-20), the corresponding DOM tree node is attached to the *JTree* node. It is used as a selection point for the display of detail information.

JTree provides an invisible root node, which is used to connect the module and object nodes. The display of these elements can be started at the visible top level without displaying to much structural information. A *JTree* node must contain three pieces of information. The kind of the node is used for displaying different node icons. The name is the textual description of the node, and the attached DOM node is used for further processing.

The accessing and parsing process of directory data is realized in the same way as for specification data. The same *JTree* is used to attach and remove tree nodes. Because of the invisible root node, the user has the impression that the tree is completely changed.

The icons can appear or disappear while resolving the kind of node. In one case an icon stands for an object or module node and in the other case it stands for a directory entry or directory entry instance node. The matching information is stored and resolved for the special icons. The load/reload buttons (cf. section 3.6.2.3) are implemented with the use of the Java *JButton* class.

4.5.1.2. Backend of the Brain Frame

Personal Brain by The Brain Technologies Corporation allows a user to organize information in an individual way without limiting it to a pre-determined file structure. It uses a new concept, a data format called *thoughts*. Thoughts can be about any type of information, including documents, spreadsheets, images, shortcuts, and HTML documents [Brain00a]. The user can organize the data in form of a tree, but this is not a requirement.

Relationships in Personal Brain are made by creating and arranging child, parent, and jump thoughts. The names simply describe the relationship to the active thought and to each other. Every node (thought) can be connected to every other node over the parent (top), child (bottom), or jump (left) gate. Parent thoughts are displayed directly above the active thought, child thoughts are displayed below the active thought, and jump thoughts are displayed to the left of the active thought (cf. Figure 4-21). [Brain00a]

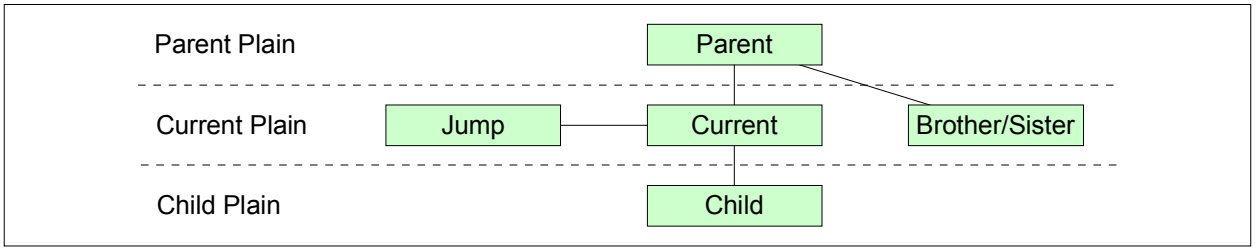

Figure 4-21: XAMAV – The Plains of the Brain

The Brain Frame is used for displaying the Core Model and the dynamic linking. It displays hierarchical and fully connected structures and has the ability to generate cross connections between the data. The following challenge is described in [Herman99]: “A fundamental challenge for information visualization applications that use graph visualization techniques for relational data sets is the scale and structural complexity of the data.” This is solved by the Brain interface in the way shown in Figure 4-21.

The static Core Model structure is displayed with *static core model thoughts*. The entry to the Core Model part is the *CORE_MODEL* thought. The elements *module*, *object*, *interface*, and *action* are connected as shown in Figure 4-22. One static thought (*qualifierdef*), which additionally includes Core Model descriptions, is added directly under the *CORE_MODEL*.

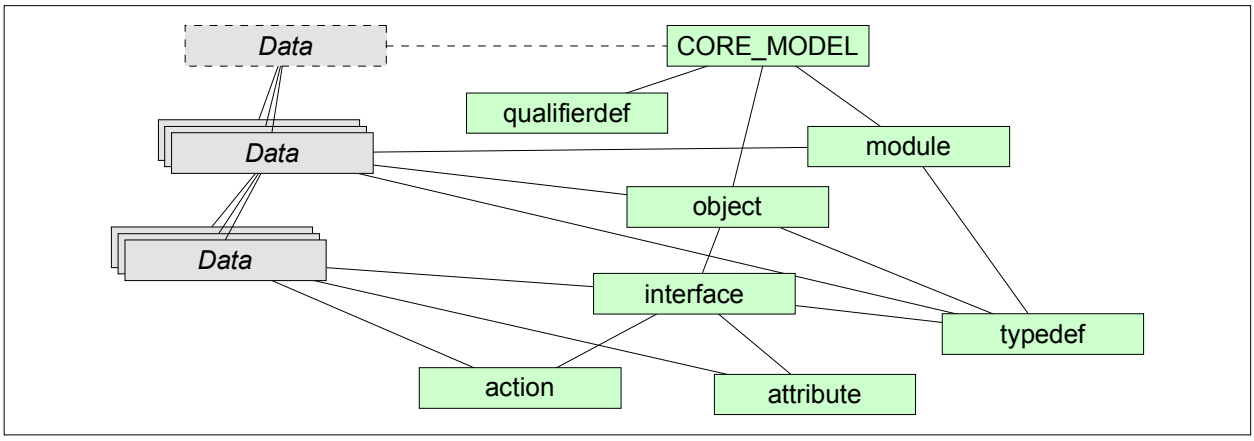

Figure 4-22: XAMAV – Core Model Structure

The Core Model data elements, defined in xADL, are connected to the static core model thoughts via a jump connection as shown in Figure 4-22. They are read by the Core Model xADL file and added dynamically. They are called *dynamic core model thoughts*. Every time a dynamic core model thought is activated, the type of the element is shown as jump connection and the detail information about this element is displayed in the Information Frame.

The connection of Core Model data with the directory and specification data is realized by structural filter thoughts. The filter thoughts, depending on the selected tree node in the directory or specification tree, are linked to the dynamic core model thoughts that contain the related information (cf. Figure 4-23).

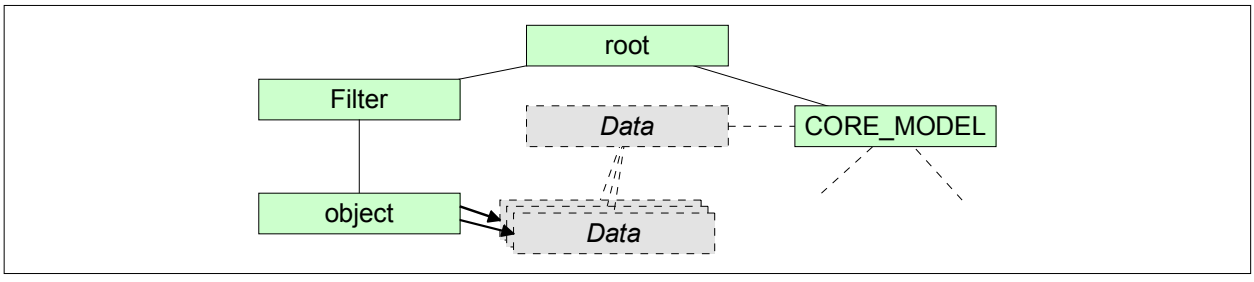

Figure 4-23: XAMAV – Dynamic Linking

For example: An object *X* is selected in the directory tree and the *object* filter thought is currently used. All *qualifierdef* thoughts of the dynamic Core Model data which match with an object qualifier are linked as a jump connection to the object filter thought. By selecting another thought, this connection is rebuilt for this element.

The dynamic core model thought is directly selectable. This provides users the possibility to obtain detailed information about the meaning of qualifiers. For an easy understanding, the static core model thoughts, dynamic core model thoughts, and the filter thoughts appear in different colors.

Figure 4-24 shows the structural filter. It returns a node graph. These nodes are called filter thoughts. *FILTER* represents the structural entry of this graph. *NO_FILTER* is added to show the unfiltered detail information.

For an easy understanding of the design the structural data filters (first level, cf. Figure 4-21) are located below *FILTER*. *Typedef* elements can be associated to the elements *module*, *object*, or *interface*. To simplify the graph without having three *typedef* thoughts, they are concentrated in one thought and linked to the module-, object- and interface filter thought. One of all filter thoughts is active every time a tree node in the directory or specification tree is selected.

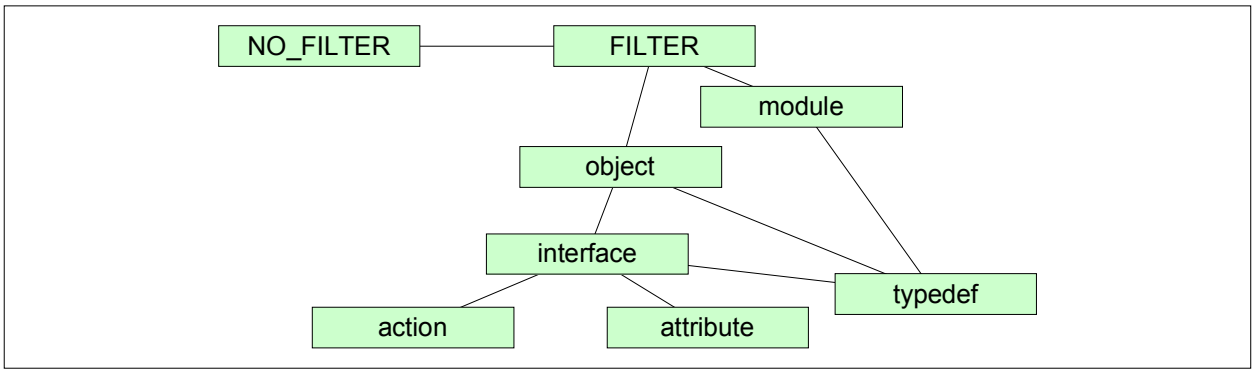

Figure 4-24: XAMAV – Filter Structure

The Brain Frame is implemented using the Brain SDK classes. Brain SDK is divided into three packages: *Database*, *User-Interface*, and *Common*. The Database package has three primary functions:

1. persistent storage and retrieval of Brains;
2. creation, association, and access of thought objects; and
3. associating thoughts and links with user defined objects.

The User-Interface package manages the visual display and communicates with the Database package to access thoughts and links. It passes three types of events to a Brain-enabled application:

1. brain events - mouse down, mouse over, and thought selection events;
2. paint events - a way to control the display of specific brain elements; and
3. application events - application close event.

The Common package contains utility classes that are used by the two other packages. The separation of the Database, User-Interface, and Common packages makes it possible to build applications and applets to create and manipulate brains without a user interface. An obvious functionality of this approach is to create programs that automatically create brains. [Brain00b]

The filter thoughts and the static core model thoughts are added statically. The Core Model data is parsed at startup. Operating with the thoughts is similar to operating with the *JTree* nodes. The DOM nodes are attached to the thoughts and are extracted by selecting the corresponding thought.

The name of the Core Model element is converted to the dynamic core model thought name and the type of the element is used to provide a jump connection to a static core model thought. On selection of static and dynamic core model thoughts or filter thoughts, different behaviors are implemented. For dynamic linking between filter and dynamic core model thoughts, the information of all relevant thoughts is stored.

4.5.1.3. Backend of the Information Frame

The Information Frame is realized with the *JEditorPane* component of the Java Swing package. This component allows to display HTML formatted text. The format definitions can be done by defining an external Cascaded Style Sheet (CSS). The source for the detail information is the DOM node attached to the specification tree structure. Starting from this node, the content of all sub nodes is written in an HTML string. This string includes the references to the external style sheet.

The Information Frame is also used for displaying external HTML pages and other plain text content. This is useful for help or documentation content. Here, the style definition used to display detail information for the Core Model differs from those used for the display of detail information of the directory or specification tree. The main difference is that for the display of *core model detail information* no separate filtering mechanism is required. The reason for this is that third level information (cf. Figure 4-21) is only displayed for this specific element (selected thought). Information about the sub elements is not displayed. They can be selected and viewed by using the Brain interface. Additional structural information as in the other detail information data is not necessary.

4.5.2. Implementation Classes

The implementation is based on the Java 1.3.1 SDK by Sun Microsystems, the XML4J classes version 1.1.16 by IBM, and the Brain SDK version 2.1 (evaluation copy) by The Brain Technologies Corporation. The Brain and XML4J classes are included in the *xamav.jar* file, where also the classes for the XAMAV application are located. Only one system class path has.

The class *Xamav* represents the core of the implementation. This class is an extension of the standard *javax.swing.JPanel* class, including the main method used for starting the application. The different components are associated with the class *Xamav* as show in Figure 4-25. The panel is split into *panes*. The following source code fragment shows in which way the class *Xamav* is defined, and how the split frame is created.

```
public class Xamav extends JPanel // class definition
public Xamav() // constructor
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZO NTAL_SPLIT);
JSplitPane splitPaneleft = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
JSplitPane splitPaneright = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
```

The main functionality is located inside the classes *XamavTree*, *XamavBrain*, and *XamavInfo*. *XamavTree* and *XamavBrain* are loaded and initialized by the *Xamav* class and added to the *JSplitPane*. *XamavInfo* is initialized by the *XamavTree* class at the first selection of the directory respectively specification tree, or by the *XamavBrain* class at the first selection of a dynamic core model thought.

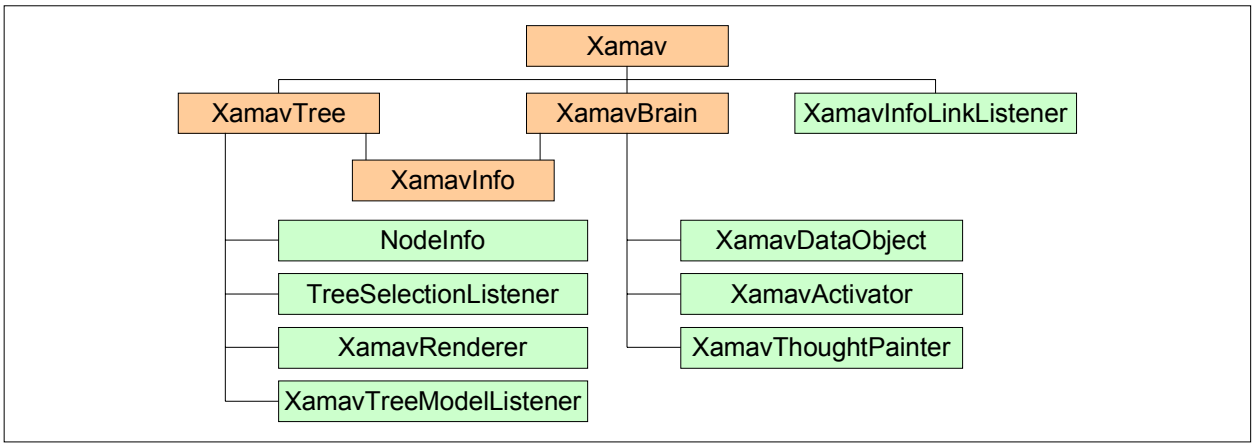

Figure 4-25: XAMAV – Application Class Association

The Information Frame is built using the *javax.swing.JEditorPane* class. The function *setText* is used to print a string inside this frame. All strings, generated by the *XamavInfo* class, are displayed in the same way. The *XamavInfoLinkListener* is an implementation of the standard *HyperlinkListener* class, which handles the activation of links in the Information Frame. The load/reload buttons are realized using the *javax.swing.JButton* class. They are added within the *XamavBrain* and *XamavTree* in a similar way as the *JEditorPane*. The following source code fragment performs the initialization of the Information Frame:

```
htmlPane = new JEditorPane(); // init class
htmlPane.setContentType("text/html");
htmlPane.addHyperlinkListener(new XamavInfoLinkListener(htmlPane));
htmlPane.setText(initialTextHTML);
JScrollPane infoPane = new JScrollPane(htmlPane);
splitPaneright.setBottomComponent(infoPane); // set splitpane
```

The instantiation of the class *Xamav* is done by the main method. After instantiation, the class is added to the *javax.swing.JFrame* and is displayed on the screen.

4.5.3. Class XamavTree

```
public class XamavTree extends JPanel // class definition
public XamavTree() // constructor
```

The class *XamavTree* contains all operations for building the directory tree and the specification tree. This includes the basic design and selection rules, the basic methods for parsing and matching the XML documents, and a class which allows DOM nodes to be attached to the *JTree* structure. During startup the *JTree*, *XamavRenderer* and *TreeSelectionListener* are initialized. All classes used with the *JTree* functionality are extensions or implementations of the standard Java classes.

Method: parsenXML

```
static public Document parsenXML(boolean url, String XMLName)
```

The *parsenXML* method parses the XML document and returns a DOM tree document depending on the parameter *url*. The parser validates the XML document when a DTD is available.

The *XMLName* parameter has to be an URL to an XML file in the local file system if the value for the parameter *url* is true. Otherwise, the URL has to start with <*http://*> and the XML file will be loaded via the HTTP protocol from the given location. The return value is a DOM document which includes all data of the XML document.

4.5.3.1. Class NodeInfo

```
class NodeInfo // class def.
public NodeInfo(Node TreeNode, String Type, String Name) // constructor
```

This class plays a key role connecting the DOM tree nodes to *JTree*. The class contains the type (e.g. *module*), the name (e.g. *app_one*), and the complete DOM tree node of this element. Inside this class, the standard Java method *toString()* is redefined to get the name of the string. The method *getDomNode()* performs the access to the DOM tree node.

Method: pref

```
public void pref(Document prefDoc) {
  // enumeration of the DOM nodes
  for (Node prefNode = prefDoc.getDocumentElement().getFirstChild();
       prefNode != null; prefNode = prefNode.getNextSibling()){
    // construct if element found
    if (prefNode.getNodeName().equals("spec")){
      // write the attribute in the variable
      XamavTree.SPEC_NAME = ((TXElement)prefNode).getAttribute("name");
      XamavTree.SPEC_TYPE = ((TXElement)prefNode).getAttribute("type");
    }
  }
}
```

The method *pref* is supplied with the DOM tree document of the *preferences.xml* file as input parameter. The values of the attributes *type* and *name* are stored in a system variable, which takes affect by loading or reloading the directory tree and the specification tree. The following source code fragment shows the three steps for accessing the DOM tree. This method is called once during the initialization phase.

Method: spec

```
public void spec(Document specDoc)
```

After the *spec* method has accessed the DOM tree of the XML document, defined in the *spec* element of *preferences.xml*, it reloads the specification tree. All existing nodes are replaced, except the invisible root node of the tree. Traversing the DOM tree is supported for a depth up to five levels. Only the *object* and *module* nodes will be connected to the tree as instances of the class *NodeInfo*.

Accessing the DOM tree, the attribute *distinguished_name* is also evaluated. If this attribute matches the last accessed directory or specification tree node, this information is stored in a hash table. Depending on these entries, the new specification tree is loaded and expanded up to that node. This method is performed on every reload of the specification tree.

Method: xdsdTree

```
public void xdsdTree(Document tdoc)
```

The *xdsdTree* method behaves similarly to the *spec* method. After the *xdsd* method has accessed the DOM tree of the XML document, defined in the *xdsd* element of *preferences.xml*, it reloads the directory tree. All existing nodes will be replaced, except the invisible root node of the tree. Traversing the DOM tree is supported for a depth up to twelve levels.

All nodes are connected to the tree as instances of the class *NodeInfo*. Accessing the DOM tree, the attribute *object_distinguished_name* is evaluated. When this attribute matches the last accessed specification or directory tree node, this information is stored in a hash table. Depending on these entries, the new directory tree is loaded and expanded up to that node.

This method is performed on every reload of the directory tree.

4.5.3.2. Class TreeSelectionListener

```
// instantiation of the standard listener class
tree.addTreeSelectionListener(new TreeSelectionListener() {
public void valueChanged(TreeSelectionEvent e)
{***** event handling ******}
```

One of the two main listeners of the application is the standard Java class *TreeSelectionListener*. The selection of single nodes placed in the directory tree and the specification tree is monitored by this listener.

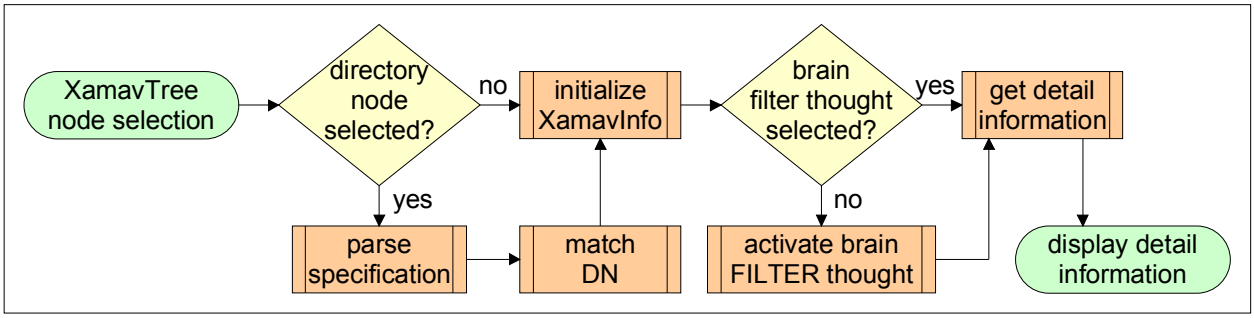

Figure 4-26: XAMAV – Tree Selection Event Handling

In case of a *TreeSelectionEvent*, the following steps are done (cf. Figure 4-26):

- unlink dynamically linked Brain thoughts;
- extract the DOM node and the name of the attached *NodeInfo* instance; and
- run different tasks according to the selected tree node.

The selection of a tree node referring to a *directoryEntryInstance* DOM node requires two more tasks to be fulfilled in comparison to the selection of other types of tree nodes

1. parse the specification XML document; and
2. compare the distinguished_name of all element nodes of this tree with the *Object_Distinguished_Name* of the selected directory tree node.

The first matching DOM node is returned. By selecting a specification tree node, this chosen node is returned. The remaining tasks are the same in both cases.

- instantiate XamavInfo;
- replace the detail information with the filter thought dependent detail information string delivered by *XamavInfo*; and
- link the dynamic core model thoughts to the filter thought syntax.

This listener is called whenever the specification or directory tree is selected.

Method: dnSpecSearch

```
public Node dnSpecSearch (Document specdoc1, String dnSearchString)
```

The directory XML file contains only information about the locations of object instances and the corresponding objects in the specification tree. This method searches for the first node inside the specification DOM tree that has the same distinguished name. The parameters for this method are the specification DOM tree and the distinguished name.

This method is called each time the directory tree or specification tree is selected.

4.5.3.3. Class XamavRenderer

```
private class XamavRenderer extends Defaul tTreeCellRenderer // class def.
public XamavRenderer() // constructor
```

All currently matched nodes are listed in the hash table *hasdisName*. Different icons are chosen by the renderer to represent a node. This depends on the type of the tree node and whether the node is listed inside the hash table or not.

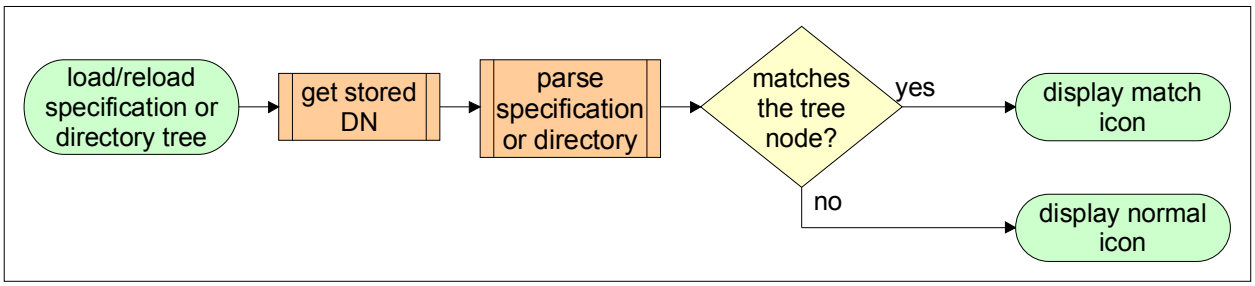

Figure 4-27: XAMAV – Reload Matching

The *renderer* updates the icons of the nodes every time after a reload or after a change of the directory tree or the specification tree. The symbols are stored in the images directory, which is located in the working directory of XAMAV. The following list shows the correspondences.

```
objectIcon = "images/oi.gif"
```

```
objectMatchIcon = "images/omi.gif"
moduleIcon = "images/mi.gif"
moduleMatchIcon = "images/mmi.gif"
directoryEntryIcon = "images/dei.gif"
directoryInstanceIcon = "images/dii.gif"
directoryInstanceMatchIcon = "images/dimi.gif"
```

4.5.4. Class XamavBrain

```
public class XamavBrain extends BrainApp // class def.
public XamavBrain() // constructor
```

The class *XamavBrain* includes all operations for building and initializing the Brain. The initialization phase comprises the building of the static thoughts, storing their identifiers, and the parsing of the Core Model file. The following steps are the conversion of the relevant DOM nodes to Brain thoughts and attaching the DOM nodes, building the tree, and connecting each thought with a static core model thought.

During this process, the number of core model thoughts is tested if it exceeds an admitted value. After this, the element name and Brain thought pairs are stored in hash tables for dynamic linking. The classes *XamavThoughtPainter* and *XamavActivator* are initialized. All classes sharing functionality with the Brain are extensions or implementations of the Brain SDK classes.

XamavBrain is initialized once during the launch of the application.

4.5.4.1. Class XamavDataObject

```
class XamavDataObject implements DataObject // class definition
public XamavDataObject (Node node) // constructor
```

This class is similar to the class *NodeInfo*. It is used to connect the DOM tree information to the Brain thoughts. The type of an element (e.g. *module*) determines the kind of connection between the dynamic core model thought and the static core model thought. The thought name is set to the name of the element (e.g. *app_one*) and the complete DOM tree node of this element is attached to the thought.

Instances of the class are created during the dynamic core model thought building by using the method *setDataObject()*. The method *getXDONode()* realizes the access to the DOM node.

4.5.4.2. Class XamavActivator

```
class XamavActivator implements ThoughtActivator // class definition
public void processEvent (ThoughtEvent e) // method for event processing
```

The second main listener of the application is the *XamavActivator*. The listener controls the selection of the Brain thoughts (cf. Figure 4-28). In addition to this *ThoughtEvent* handling, the type of thought has to be tested and different tasks have to be done. These tasks are:

- unlink and link the dynamic linked core model thoughts with a filter thought in case of a filter thought selection and display the filtered detail information;
- display the dynamic core model thought detail information if selected;
- display the buttons *Help*, *About* or *Preference* information if selected;
- open the *Open File* or *Open URL* window; and
- exit the application if *Quit* is selected.

This activator is called whenever a thought is selected.

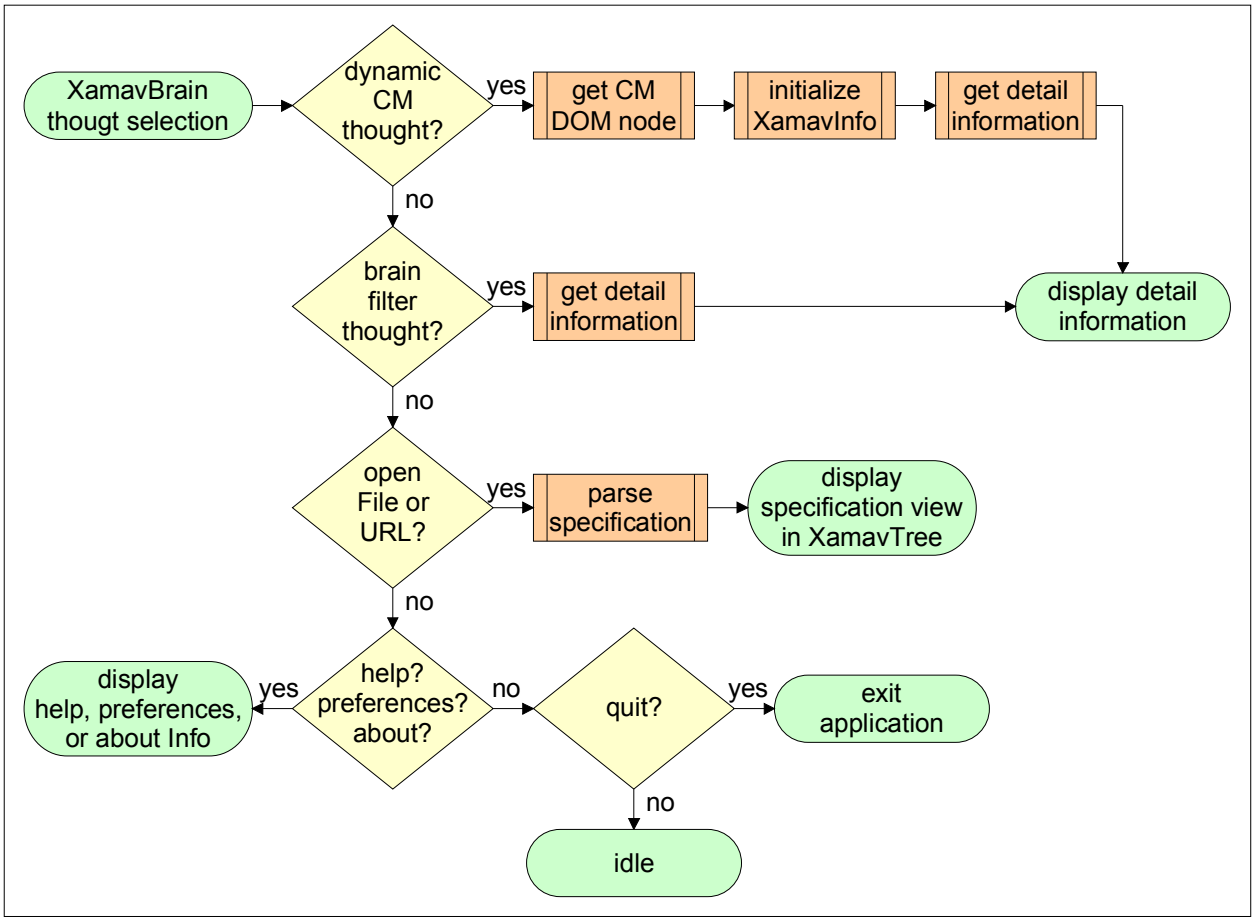

Figure 4-28: XAMAV – Thought Event Handling[3]

4.5.4.3. Class XamavThoughtPainter

```
class XamavThoughtPainter implements ThoughtPainter // class definition
public void preDraw(ThoughtRep rep) // method for recursive thought painting
```

The class tests recursively the type of every thought and paints it in the type dependent color. This drawing is performed once at startup of the application. The instantiation of this class is done in *XamavBrain*.

4.5.5. Class XamavInfo

```
public class XamavInfo // class definition
public XamavInfo(Node node) // constructor
```

The detail information is displayed in the Information Frame. This frame is initialized at the application's startup. The classes which are used for generating this frame are extensions or implementations of the *javax.swing.JEditorPane* classes. One feature of *JEditorPane* is displaying of HTML content formatted with a CSS, which is used in this implementation. The *XamavInfo* class creates different strings for the detail information in HTML syntax and provides the use of an external CSS.

The parameter node, which is required for instantiating the class, is the starting point for the string creation after string initialization. The information needed for the dynamic linking of the dynamic core model thoughts is stored in a hash table during processing phase.

[3] CM – Core Model

The instantiation is done by selecting nodes in the directory tree or the specification tree and by selecting dynamic core model thoughts in the Brain.

4.5.5.1. String Calculations

The string calculations have two main tasks. The first task is generating HTML by the use of CSS. The second task is generating strings for the different filter options. To perform the calculations in an effective way, they are done accessing the DOM tree. Nine different calculations are completed after one processing phase. A change of a filter thought causes a reload of the Information Frame containing the new detail information string without any recalculations. The nine parts of the detailed information are listed below.

```
public String nofilter_info = HTML_INI; //NO_FILTER thought selection
public String filter_info = HTML_INI;   //FILTER thought selection
public String module_info = HTML_INI;   //module thought selection
public String object_info = HTML_INI;   //object thought selection
public String interface_info = HTML_INI;//interface thought selection
public String action_info = HTML_INI;   //action thought selection
public String attribute_info = HTML_INI;//attribute thought selection
public String typedef_info = HTML_INI;  //typedef thought selection
public String cm_info = HTML_INI;       //dynamic core model thought selection
```

The initialization string starts with <*html*>. The following header information defines that a CSS is used and where it is located. The name (*xamav.css*) and the location (XAMAV working directory) of the external CSS file are fixed. The following syntax shows the entire HTML initialization string:

```
String HTML_INI =
"<html><head><link rel=\"stylesheet\" type=\"text/css\"
href=\"file:xamav.css\">"+"<style type=\"text/css\"></style></head><body>";
```

The information of every accessed node is put in a string. This string is concatenated for every type of filter. The code example below describes an object node:

```
// DOM-Tree access
name           = ((TXElement)child).getAttribute("name");
object_extends = ((TXElement)child).getAttribute("extends");

// String calculation for the filter_info ******************************
filter_info = filter_info+
            "<li class=\"f.object\">Object : <font class=\"f.object.name\">"
            +name+"</font></li>";

// String calculation for the nofilter_info ****************************
nofilter_info = nofilter_info+
              "<li class=\"f.object\">Object : <font class=\"f.object.name\">"
              +name+ "</font></li><font class=\"f.object.ext\"> Extends: "
              +" <font class=\"f.object.ext.name\">"+object_extends
              +"</font></font>";
```

4.5.5.2. Organization of CSS

The organization of the CSS is done by class definitions. The major class separation is done for the filter and Core Model. The other separations are done for the xADL elements, attributes and values of the specification, and Core Model document. A list description is added for all elements, which are directly selectable by the corresponding filter and dynamic core model thought. For the class definitions, the following syntax was chosen:

- (list entry). filter/core model. element name. value of attribute type =”name”
- .filter/core model. element name. attribute type. value of attribute type

The definition of the separate style sheet document is built with a subset of CSS1 styles. The usable styles are defined at the top of the style sheet document. The complete style sheet definition can be found in Appendix C.4.2. The following fragment of the style sheet document (*xamav.css*) shows class definitions for the element object.

```
/* object styles (filter)*/
li.f.object { color:red; font:18pt Helvetica;}
.f.object.name { color:black; font:18pt Helvetica;}
.f.object.ext { color:red; font:12pt Helvetica;}
.f.object.ext.name { color:black; font:12pt Helvetica;}
```

4.5.6. Dynamic Linking and Unlinking of Thoughts

The information for dynamic linking and unlinking of thoughts is stored in hash tables. Different matching tables are used for the elements *qualifierdef*, *object*, *attribute*, and *typedef* dynamic core model thoughts. The linking tables are built for module, object and interface filter thoughts. For the linking and unlinking mechanism, the tables are created in three different steps.

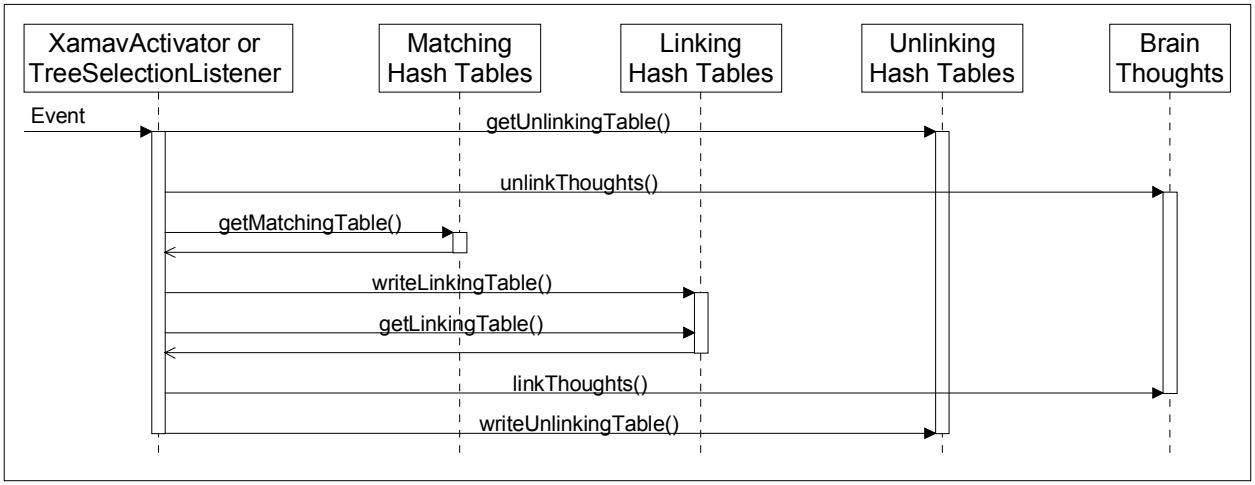

Figure 4-29: XAMAV – Dynamic Linking of Thoughts

1. All matching tables are created while building the dynamic core model thoughts at startup of the application. They are static while the application is running.
2. The linking tables are created on every initialization of *XamavInfo*. This is done on every selection change of the *XamavTree*.
3. The unlinking table is a copy of the linking table which is responsible for the actual thought linking.

The key and value entry pairs of all hash tables are identical. The attribute name of the element is used as key (e.g. *<qualifierdef name="Abstract" ...> ; key = "Abstract"*). The dynamic core model thought with the corresponding thought name is the value entry of the hash table. The linking and unlinking process enumerates the correspondent hash table. It links/unlinks the actual selected/deselected thought to the thoughts listed in the table. The dynamic linking process is illustrated in Figure 4-29.

4.5.7. XAMAV User Interface (Manual)

The main components of the application are shown in Figure 4-30. They are the Brain Frame, the Tree Frame, the Information Frame, and the Console window. The Brain Frame displays the Brain consisting of thoughts. The Tree Frame displays the directory tree or the specification tree. The Information Frame displays the detail information and the Console window shows error messages or processing information.

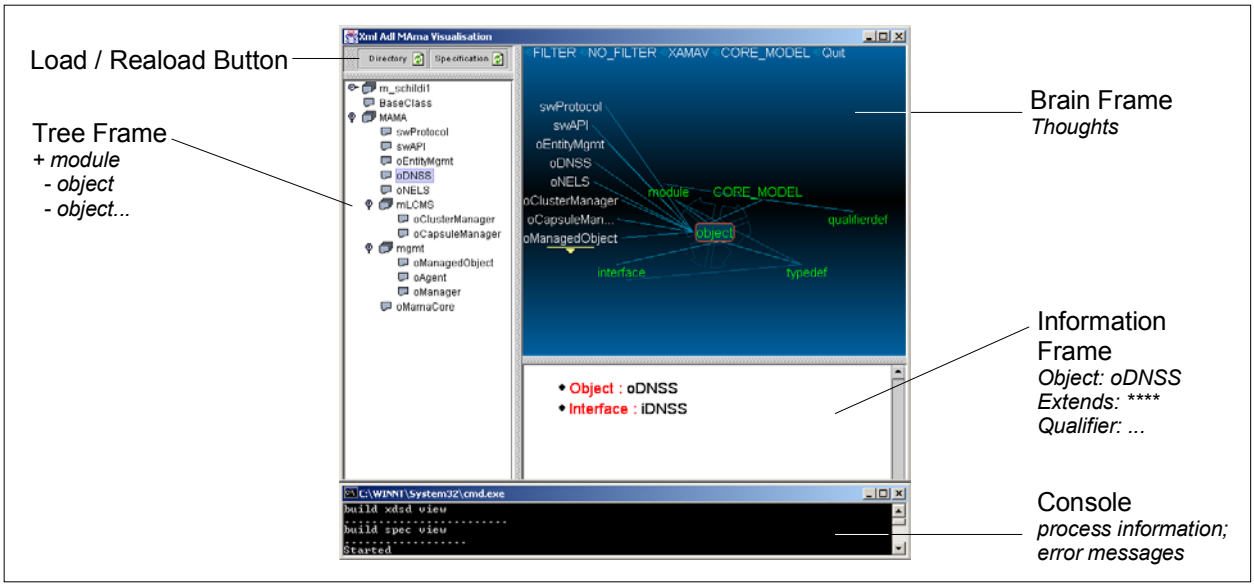

Figure 4-30: XAMAV – User Interface

4.5.7.1. Installation and Startup

The hardware and software requirements are a Microsoft Windows or UNIX platform with a Java Runtime Environment (JRE) version 1.3.1 or higher. XAMAV is delivered as a Zip archive. The installation directory has to be created manually e.g. *C:\program files\XAMAV* (Windows system) or */opt/XAMAV* (UNIX system). The archive has to be copied into that directory and to be extracted. The following files and folders are present after extraction:

- xamav.jar – Java archive include all necessary classes;
- preferences.xml – preferences file for XAMAV;
- xamav.css – CSS for the XAMAV Information Frame;
- help.html/css.html – help information pages;
- xdd.xml/spec.xml/cm.xml – demonstration files for the three types of input data;
- images – directory which contains the Tree Frame symbols; and
- xamav.bat – Batch file for launching XAMAV.

The system variable *CLASSPATH* has to be changed. The path to *xamav.jar* has to be added. After setting the path, XAMAV can be started by using the batch file or by the command *java Xamav*. This command has to be executed in the working directory of XAMAV e.g. *C:\program files\XAMAV*. The filenames for the specification data, directory data, and the Core Model data can be edited in the *preferences.xml* file. XAMAV requires an XML file with the root node *<xamav_preferences>* and three child nodes: *<spec...>*, *<xdsd...>* and *<cm...>*. The following code example shows the *preferences.xml*:

```
<xamav_preferences>
<spec type="FILE" name="spec.xml"/>
<xdsd type="FILE" name="xdsd.xml"/>
<cm type="FILE" name="cm.xml"/>
</xamav_preferences>
```

4.5.7.2. The Tree Frame

The Tree Frame displays the directory and the specification tree. Directory tree information is displayed by pressing the *Directory* button. Specification tree information is displayed by pressing the *Specification* button. The navigation through both trees is similar. The tree icons show what tree is actually loaded.

The document specified in the *xdsd* element of the preferences file is displayed in the directory tree. The *DirectoryEntry* element is displayed as an expandable node and the *DirectoryEntryInstance* element is displayed as a leaf. The specification tree, specified in the spec element of the preferences file, displays the element *module* as an expandable node and the element *object* as a leaf.

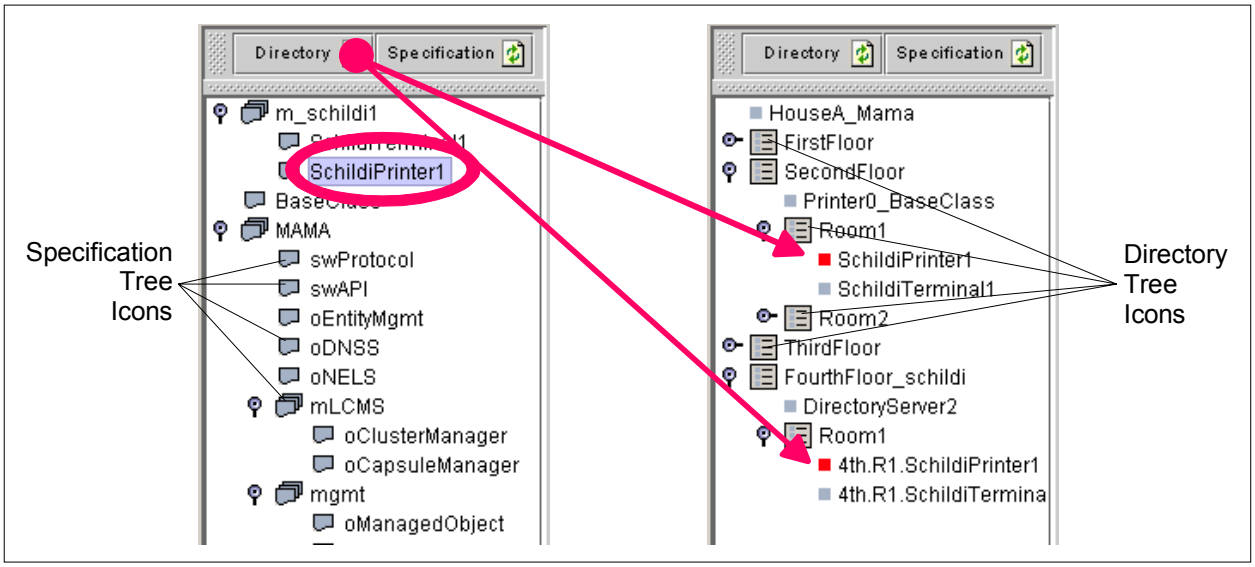

Figure 4-31: XAMAV – Load with Specification and Directory Tree

Figure 4-31 shows the directory tree and the specification tree. Furthermore, it shows the effect of loading a tree. To illustrate this on an example: When the object *SchildiPrinter1* in the specification tree is selected and the *Directory* button is pressed, the Tree Frame switches the display to the directory tree and shows all instances of the object *SchildiPrinter1* with a red icon. The displayed information in the Information Frame is not changed. The Information Frame displays detailed information, when a node of the Tree Frame was selected.

A selection of an expandable node in the directory tree displays no detail information because of its property. By switching between the two trees, a red icon appears at the corresponding class in the specification tree or at the instances in the directory tree. All instances of the same class are marked with the red icon after a reload of the directory tree.

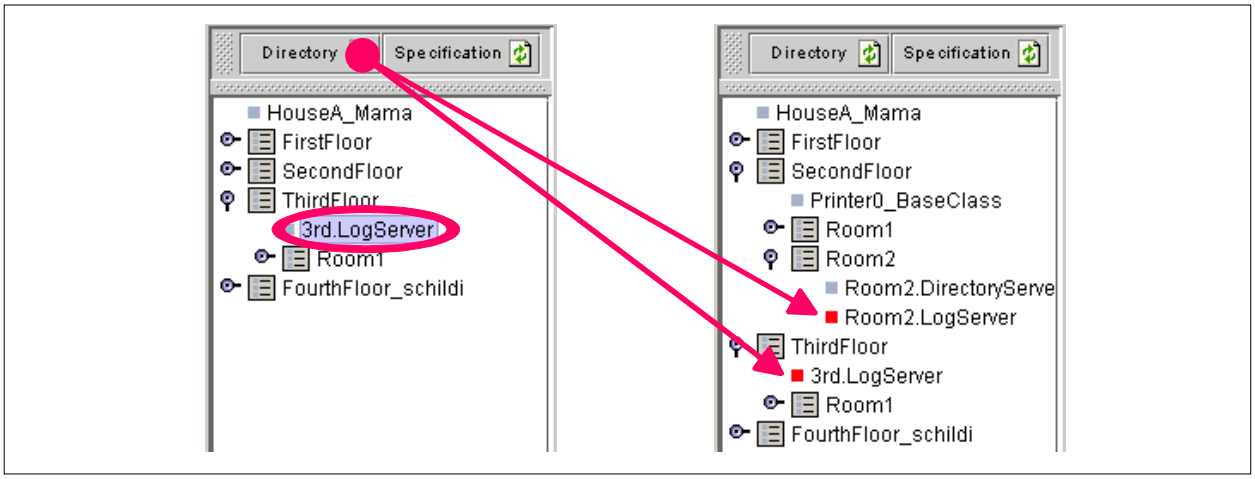

Figure 4-32: XAMAV – Reload of the Directory Tree

Figure 4-32 illustrates how the reload takes effect. When the node *3rd.LogServer* in the directory tree is selected and the *Directory* button is pressed, all instances of the same object are shown with a red icon.

4.5.7.3. The Information Frame

The Information Frame displays three kinds of data. It shows the detail information of the selected object, module, or instance node of the directory tree or the specification tree. Furthermore it shows detail information of the Core Model, when the thoughts are selected in the Brain Frame. Beside this, the Information Frame is used to display some general information like preferences or help information. Figure 4-33 shows the displayed detail information.

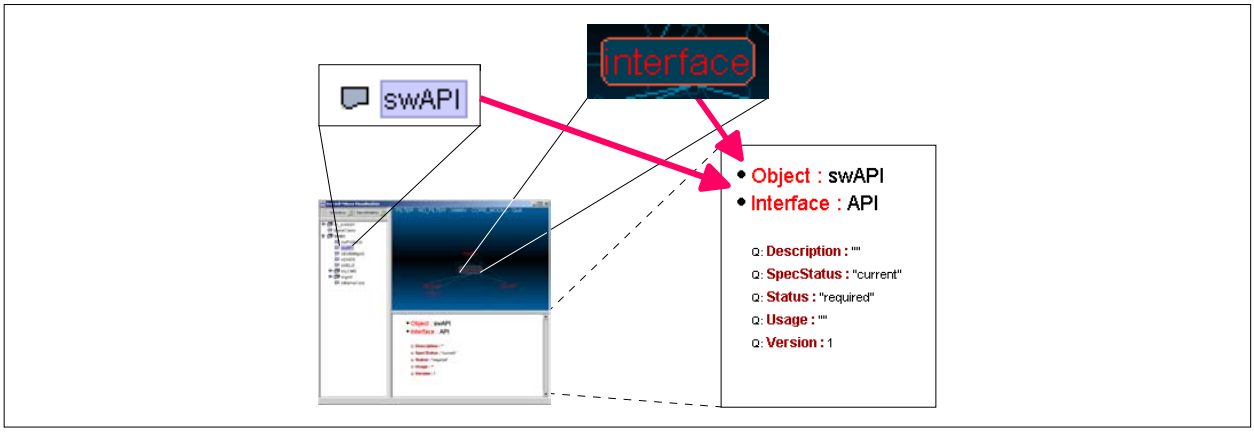

Figure 4-33: XAMAV – Interface Filter

4.5.7.4. The Brain Frame

The Brain Frame consists of different types of thoughts. These types are:

- yellow thoughts (administrative functions);
- red thoughts (structural filter functions);
- green thoughts (static Core Model structure); and
- white thoughts (dynamic Core Model elements).

The yellow thoughts, including the blue Quit thought, represent the entrance and exit of XAMAV. Further functions of these thoughts are: display the help information (*Help*), display the about information (*About*), and display the current preferences (*Preferences*). The content is displayed in the Information Frame. The root thought of the Brain is XAMAV. It connects all thoughts. Pins (*thought aliases*) are placed on the top of the brain. They allow a faster access to these thoughts. (cf. Figure 4-34).

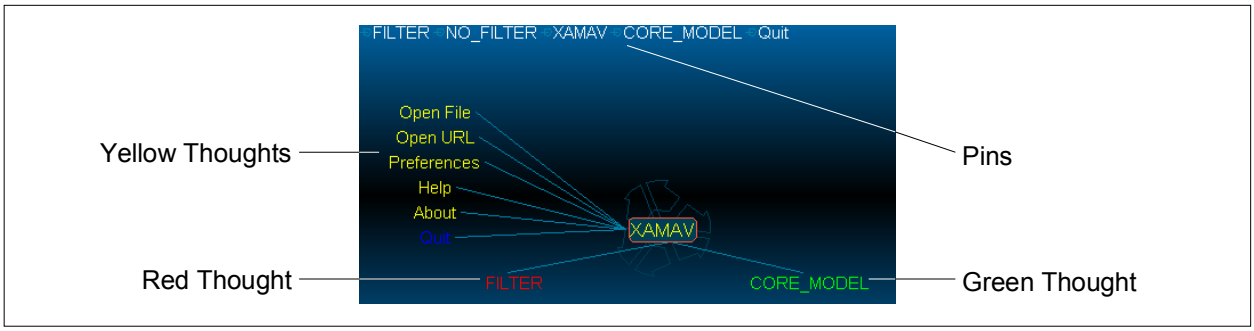

Figure 4-34: XAMAV – Brain Frame with Thoughts

The red thoughts are the filters for the detail information. The activation of the thoughts leads to filtering the detail information:

- *FILTER* – module, object, and interface names are displayed;
- *NO_FILTER* – all detail information is displayed; and
- detail information of the actual selection is displayed.

Example: When the object filter thought is activated, all detail information of the current selected object in the Tree Frame is displayed. The detail information of the object's substructures (interfaces or typedef) is not displayed. This information is only displayed if an interface or typedef filter thought is activated.

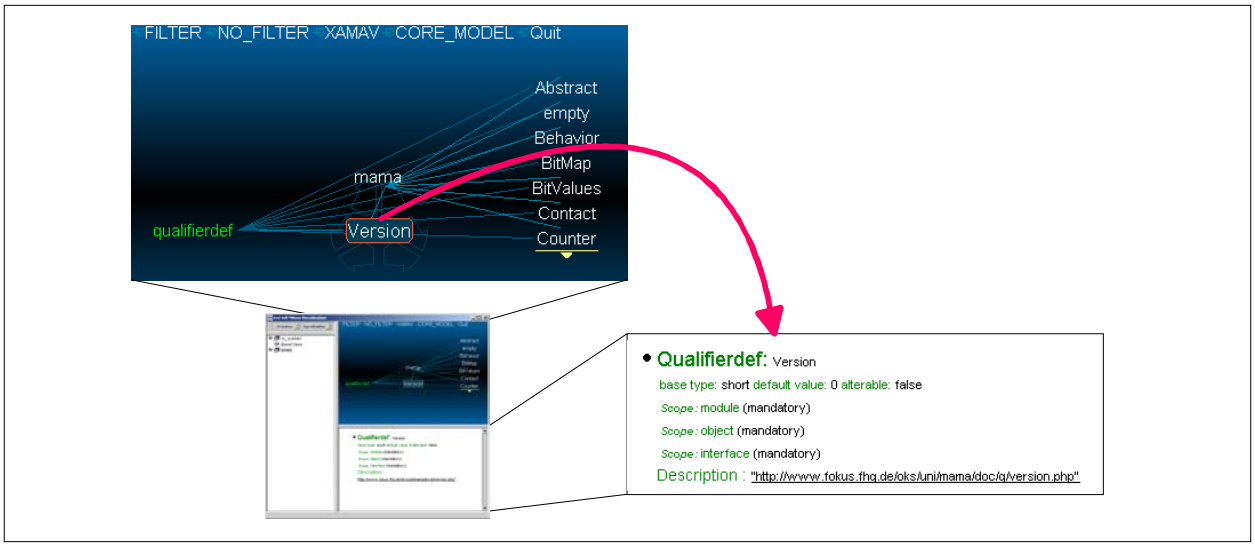

Figure 4-35: XAMAV – Core Model Information

The green thoughts show the structure of the Core Model data and represent the key to the white thoughts. The white thoughts contain the Core Model elements by name. They are connected via a jump connection to the green thoughts with the correspondent kind. By activating one of the white thoughts, the detail information is displayed in the Information Frame (cf. Figure 4-35).

Pins are selectable buttons on the top of the Brain frame. They are aliases of existing thoughts and activate the corresponding thought on selection. Result is a change in the Brain Frame: The thought selected by the activation of a pin is displayed in the center of the Brain Frame. The Brain Frame offers five different pins that enable an immediate context switch. The pins, as depicted by Figure 4-34, are *FILTER*, *NO_FILTER*, *XAMAV*, *CORE_MODEL*, and *Quit*. The XAMAV thought (root thought) is used to rebuild the initial constellation of the Brain thoughts.

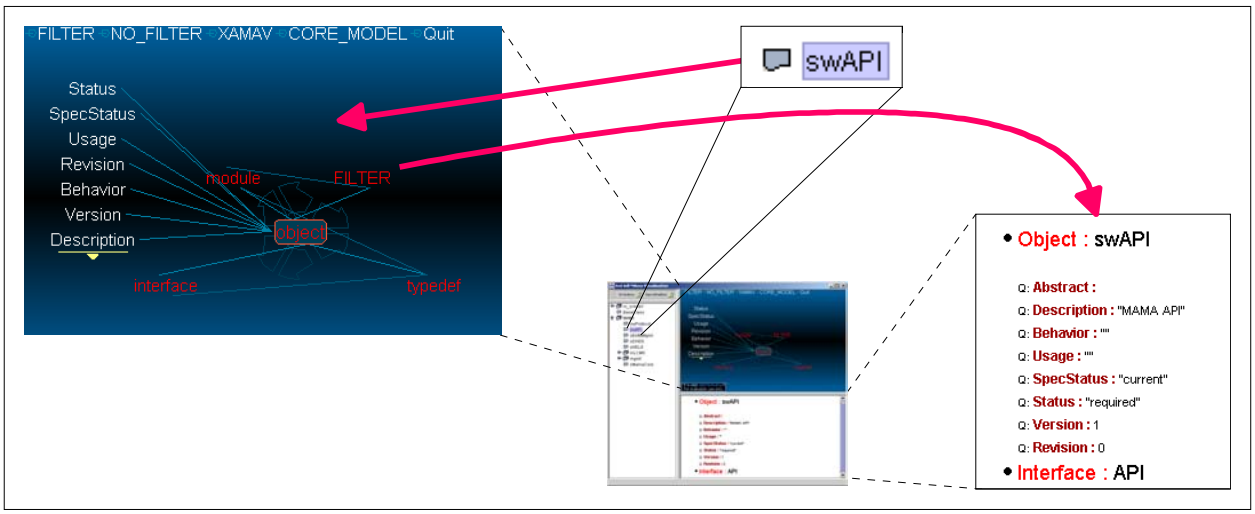

Figure 4-36: XAMAV – Object Filter inclusive Linking of Core Model Thoughts

Connection links between a red thought and white thoughts appear when the Core Model data corresponds with elements located in directory tree or specification tree. Figure 4-36 illustrates how the connection appears in the Brain Frame.

4.6. Notification Event and Log Service

The Notification Event and Log Service (NELS) has been implemented in the first step as a stand-alone object. In the second step, this stand-alone object has been included in the MAMA API. This approach has been chosen to avoid an extra, centralized server within a MAMA execution environment. The overhead the NELS functionality produced in the API was acceptable.

With the integration of the API and the NELS, all MAMA objects within a domain can play the role of an NELS. Each object registers with the DNSS of its domain. The next step is the search for NELS functionality. This search is done with a request on the DNSS. When no NELS is found, the object itself can provide NELS functionality. Because this mechanism is realized by the API, it is completely transparent for the core object.

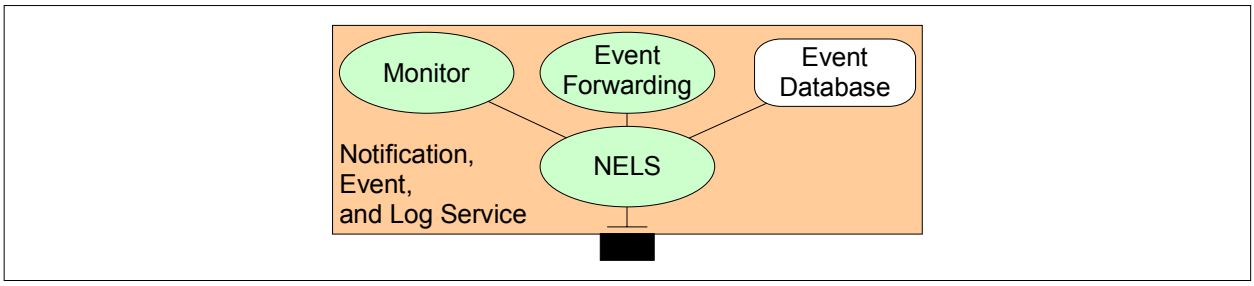

Figure 4-37: Services – Notification Event and Log Service

Figure 4-37 shows the realization of the NELS. The implementation consists of four objects. The object *NELS* implements the operations of the NELS as described in section 3.6.3. The object *Event Forwarding* handles all client subscriptions and evaluates to which clients an incoming ticket should be sent. The object *Monitor* is a console window that prints received tickets. The object *Event Database* is a name-value list that manages all received tickets. This functionality has been separated from the other objects in order to allow the usage of a real database in the future.

4.7. Lifecycle Management Service

Figure 4-38 depicts the control of a single MAMA application object. The capsule manager forwards lifecycle operations to the cluster manager that invokes them on the lifecycle interface of the application object. This interface is specified in the Core Model and realized by the MAMA API. The lifecycle operations are transparent to the core object.

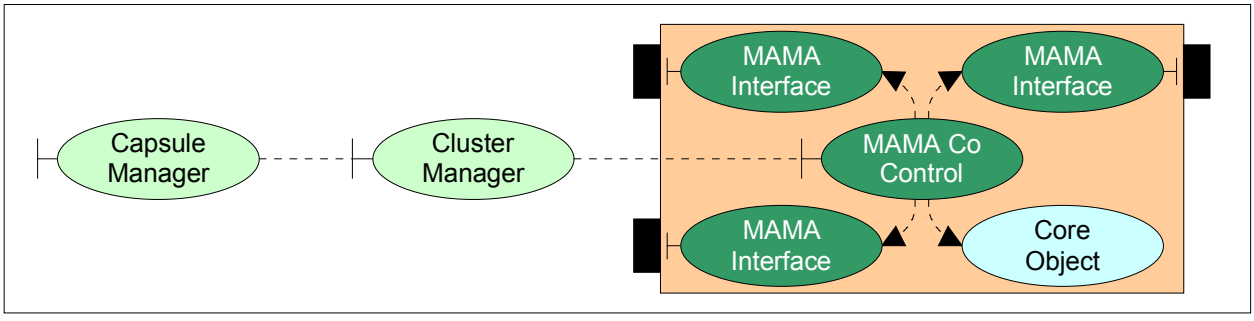

Figure 4-38: LCMS – Engineering View for a specific Object Class

The capsule manager can control more object instances of different object classes by the help of object specific cluster managers. With this mechanism, the lifecycle control of objects is decoupled from the control of a set of object instances. The capsule manager can implement templates for a configuration of objects. Following this idea, a set of objects can be instantiated at the same time.

The functionality of the object's control interface was realized within the MAMA API. The functionality is completely hidden to the application programmer. A management application can access the control interface with standard MAMA mechanism (that is, it calls the ADL typed operations).

A configuration of MAMA objects is responsible for the creation of the capsule and the cluster manager. These managers are realized as standard MAMA applications. They can be configured via command line options and preference files in order to server for a particular type of objects or object groups.

Chapter 5
Summary

This chapter concludes this thesis. The chapter summarizes the basic ideas and results of this thesis. The major objective of this thesis is to develop an approach for the integration of middleware and management concepts. Due to the current situation, which can be described by interworking instead of integration, this work has investigated how management concepts can be applied to middleware.

5.1. Conclusions

Embedded into the concepts of I-centric Communication, this work has and related developments have presented a way for the integration of management and middleware concepts. Starting with the identification of major activities, followed by the definition of a general framework and the derivation of a specific architecture, a prototype implementation has been developed. It shows the integration of concepts from both areas middleware and management.

The challenging aspect of this work has been on the one hand to the diversity of available concepts and on the other hand the fact that emerging concepts and technologies are going to change all facets of software development. In chapter one, the section 1.1 describes the areas of application with their particular requirements. This thesis recognizes telecommunication, network computing, devices and wearables, and context-aware applications. All these areas demand for integrated solutions in order to optimize the handling of applications, services, and resources. The approach of this thesis combines a basic assumption, a general framework, and a specific architecture.

The basic assumption for this work was a clear identification of areas of concern and activities of distributed systems. Section 2.1 defines five terms that can be applied to middleware and management systems. They cover all aspects of a distributed system: interface to users and costumers (*use*, *operation*), requirements of network and service operators (*control*), and long term operation (*administration*, *maintenance*). Based on these definitions, the two major activities of a distributed system are identified as *information mapping* and *system management*. Both activities are related to a layered model that depicts applications, services, and resources. An approach for the integration of middleware and management must support the mapping of information between layers and the management of individual layers supporting use, operation, control, administration, and maintenance of a distributed system.

The general framework offers concepts and rules. It is based on the introduced assumption and an evaluation of target environments. The concepts of the framework cover objectives and requirements. The major objective of the framework is to provide a distributed system an integrated mechanism for operation and management. The requirements depict important issues such as technology independence, portability, and scalability. The rules of the general framework reflect the concepts within a multi-layered model. The conceptual model provides the basis for the development of a specific architecture. The model itself includes rules for four specific problem contexts: applications, objects, services, and technology. The model sets the focus for a specific architecture on the planes regarded to objects and services.

The Middleware and Application Management Architecture (MAMA) is the specific architecture that is directly derived from the general framework. Six recommendations form the architecture a Meta Schema (object model), the Application Definition Language (ADL) for the specification of applications, a Schema and a Core Model with generic specifications for applications and the architecture itself, an Application Protocol, an Application Programming Interface (API), and a set of Application Services to support applications. Furthermore, the MAMA describes a method for the realization of distributed applications employing all six recommendations. Each recommendation starts with specific objectives and requirements. This enables the substitution of the concrete technologies that have been developed with

other, more appropriate or environment specific solutions without changing the architecture itself and without a negative impact to the other recommendations and technologies.

The specific architecture was implemented following the objectives of the framework to show that the provided recommendations are a solution that is lightweight, open, smart, and service generic.

- *Lightweight* as the solution can be widely adopted by vendors and providers of different size and market penetration. This means the solution take into account commonly accepted principles already adopted by service providers and network operators.
- *Open* as the solution includes well-defined interfaces. Interoperability with legacy systems is a key issue for a smooth integration. This thesis reflects the term open also to indicate that the market demands for an easy adaptation of new technology and interworking with other systems.
- *Smart* as the solutions must reflect the intrinsically dynamic aspects (particularly for the subscription, deployment, and session set up process of applications, services, and resources), enabling a flexible adaptation to customer requirements and operator/provider needs. This will is supported through the use of meta-data repositories throughout the whole system life-time to provide a comprehensive knowledge base improving multi-domain service provisioning and also in the operations/maintenance phase.
- *Service generic* as the solution is independent of the actual services that are offered. Nobody can predict if there is a killer-application for future services and which one this might be. The preferred applications, services and resources will surely differ significantly within different countries and different cultural groups.

This thesis describes a new approach for the support of distributed applications. However, some issues have been considered to be out of the scope for this approach. These issues are security, testing, and tool support. All of them are important for the specification and operation of distributed applications. They have been introduced when appropriate to show the places where further investigation on these issues is necessary. Some aspects of security have been recognized in two recommendations of the architecture (Application Protocol and Application Services). In both recommendations, a basic set of security options has been included. Testing is supported for the formal processing of ADL specifications with the implemented parser. Further formal test methods have not been developed. The support of tools for specification, development, and deployment is limited to the ADL parser and the visualization service.

The basic result of this work is the conclusion that an approach that integrates basic concepts of middleware and management – compared to the state of the art gateways between both worlds, provides benefits in many different areas. The unification of use, operation, and control with the tasks of maintenance and administration minimizes the effort that has to be spent for the mid- and long term operation of a distributed application. The independence of concrete middleware and management technologies improves the portability of applications. An application designer and programmer can concentrate on its actual task – the realization of profitable applications instead of dealing with constantly changing technological issues. The simplicity of the six recommendations of the architecture allowed for a simple and lightweight implementation that can be employed in many different environments, starting from small devices up to complex and huge service platforms.

5.2. Outlook

5.2.1. Scalability, Portability, and Application

The implementation of the developed architecture allows tailoring for various purposes and environments. While the first implementation is focused on the evaluation of the concepts of the architecture, the scalability in many directions was a major objective of this thesis.

The developed architecture and the realized implementation can be employed for two purposes – complementary management system to support existing applications and for the development of distributed applications that are completely based on MAMA. The architecture already provides an algorithm for the definition of distributed applications including the identification of managers, agents, and managed ob-

jects. This algorithm provides a sound tool for the use of the architecture for both purposes. The results of this work have been used and will be further used for several projects and systems that are related to the research activities of the department for Open Communication Systems (OKS) at the Technical University Berlin and the Fraunhofer institute FOKUS.

5.2.1.1. Telecommunication – Managing a Unified Messaging System

OKS has developed a Unified Messaging System (UMS). MAMA has been employed to build a complementary management system for the UMS. For the management system, the algorithm described in section 3.7 has been used. The functional description of the UMS ([vdMeer00b]) and the technical description of its components ([Dutkowski01]) have been carefully processed to indicate relevant management function, specifications for agents and managed objects, and the hierarchical structure of the management system itself.

Result was a management system that covered the complete functionality of the UMS with appropriate management functions. The system was assembled out of eleven managed objects, four agents, one local manager, and one global manager. It allowed for an easy management of user related information (which are spread over multiple objects) and the monitoring of the actual system behavior.

5.2.1.2. Internet – Maintaining a World Wide Web Server

OKS offers all information related to lecturing and research via a complex World Wide Web (WWW) server. One important issues is the monitoring of this WWW server in order to analyze the user behavior (which pages are viewed how often from different users), to notify any case of an error, and to maintain and update information.

The MAMA implementation has been used in combination with a number of scripts to realize those objectives. All scripts are handled as managed objects. The scripts allowed to extract OKS related information from the log files of the WWW server and to add information on the OKS web pages. This small and simple system can be enhanced to offer a complete solution for the monitoring of WWW servers that deal with complex information.

5.2.1.3. I-centric Communication

Some aspects of the general framework and the application services have already been introduced into the concept of I-centric Communications [vdMeer00a]. Furthermore, the specific architecture and the realized applications can now be used as a platform for deploying I-centric Communication systems within heterogeneous environments. Another challenge for further developments is the adaptation of MAMA within the I-centric Communication System to improve the operation, control, ad maintenance of the system itself.

5.2.1.4. Network Appliances

One emerging issues in telecommunications is the convergence of information and communication networks. Here, the control of network appliances is a key factor for the telecommunication business to create value-added services that can attract new groups of users. This thesis already provides a scalable and portable mechanism to control and to maintain resources. The application of the MAMA implementation for the control of IP devices, which are currently developed by Fraunhofer FOKUS, would be an excellent evaluation of the results of this thesis.

5.2.2. Related Work

In the last years, a number of projects have been started aiming for the integration of management and middleware. A couple of those projects have still investigated into the development of gateways and new methods for the mapping of information from and to a management architecture (like TMN[1]) from and to

[1] Telecommunication Management Network

a middleware architecture (like CORBA[2]). The ACTS projects FlowTrhu, REFORM, and VITAL are candidates for this category of projects (cf. [Pavlou99]). The other category of projects goes a similar way as this thesis does. Applications are decoupled from underlying technology and an architecture takes care of standard tasks. These projects do not always recognize the management aspect.

5.2.2.1. The JXTA Project

In 2001, Bill Joy and Mike Clary from Sun Microsystems started the JXTA project as a set of open, generalized peer-to-peer protocols [JXTA]. JXTA promotes the communication among applications, administrative commands for peer-to-peer tasks, a small core, supports multiple platforms, and addresses security. Focusing on the support of peer communication, the JXTA project promises to harmonize recent developments generating software for interoperable peer-to-peer systems.

5.2.2.2. The Ninja Project

The Ninja project was started from Matt Welsh at the University of California at Berkeley [Ninja]. This projects aims to develop a software infrastructure for next generation Internet applications. The Ninja project can be seen as a source for further developments of MAMA, including approaches for wide-area state management, automatic service composition, and mobile code for service deployment.

5.2.2.3. CORBA MAN

The CORBA MAN project addresses issues of telecommunication management that involves the use of CORBA [CORBA-MAN]. The basic approach is to develop a CORBA management agent for the fault management of embedded CORBA systems.

5.2.2.4. AlbatrOSS

The IST project AlbatrOSS – Architecture for Location Based Applications of Third generation Operation Support Systems – was started on March 1st 2002. Aim of this project is to develop an Operation Support System (OSS) for 3rd generation mobile networks, including services for the Virtual Home Environment (VHE). This thesis has already contributed to this project. Furthermore, the results of this thesis will be used as one basis for the definition of the OSS architecture.

[2] Common Object Request Broker Architecture

List of Figures

List of Tables

References

References in this document are constructed by the main authors name and the year of publication. For documents that are produced by an organization, the reference contains the organizations acronym and an abbreviation of the document title. Documents of project milestones are a combination of an acronym of the project name and an acronym describing the deliverable number.

References from the World Wide Web (WWW) are accompanied with the Uniform Resource Locator (URL) that points to the actual document in the WWW. Additionally, each WWW reference is provided with information when the author of this document has last visited the related document. It is most likely, that the content of the WWW document has changed since this last visit or that the WWW document is no longer available at all. All referenced documents from the WWW can be requested from the author.

[3GPP-OSAReq] 3GPP: *Stage 1Service Requirements for the Open Service Access (OSA)*. 3rd Generation Partnership Project, Technical Specification, Release 5, Document Number TS 22.127, June 2001

[Abowd99] Abowd, G. D.; Dey, A. K. et. al.: *Towards a Better Understanding of context and Context-Awareness*. First International Symposium on Handheld and Ubiquitous Computing, HUC'99, Karlsruhe, Germany, September, 27-29, 1999

[ANTLR-Man] Terence Parr et. al: *ANTLR Reference Manual*. Manual for ANTLR Version 2.7.1, October 1, 2000
available at <http://www.antlr.org> (last visited 03/14/02)

[Arbanowski00a] Arbanowski, St., van der Meer, S., Popescu-Zeletin, R.: *I-centric Services in the Area of Telecommunication 'The I-Talk Service'*. Proc. of 6th IFIP Conference on Intelligence in Networks, SmartNet 2000, Vienna, Austria, September 18-22, 2000, pp. 499-508

[Arbanowski99] Arbanowski, St.; van der Meer, S.: *Service Personalization for Unified Messaging Systems*. Proc. of the 4th IEEE Symposium on Computers and Communications, ISCC'99, Red Sea, Egypt, July 6-8, 1999

[Arbanowski00b] Arbanowski, St.; Waterstrat, H.; van der Meer, S.; Popescu-Zeletin, R.: *Open Profiling for Ubiquitous Computing*. Proc. of the 1st Workshop on Ubiquitous Computing, PACT 2000, Philadelphia, PA, October 15–19, 2000

[Arbanowski98] Arbanowski, St., Breugst, M., Busse, I., Magedanz, T.: *Impact of Standard Mobile Agent Technology on Telecommunications*. Proc. of the 5th Conference on Computer Communications, AFRICOM-CCDC'98, Tunis, October 20-22, 1998

[Badach94] Badach, A., Hoffmann, E., Knauer, O.: *High Speed Internetworking: Grundlagen und Konzepte des FDDI- und ATM-Einsatzes*. Addison Wesley, Reading, Massachusetts, 1994

[Badach97] Badach, Antol: *High Speed Internetworking: Grundlagen, Kommunikationsstandards, Technologien der Shared und Switched LANs*. 2. aktualisierte und überarbeitete Auflage, Addison Wesley, Longman, 1997

[Balzert99] Helmut Balzert: *Lehrbuch Grundlagen der Informatik*. Spektrum Akad. Verlag; Heidelberg/Berlin; 1999

[Bapat94] Bapat, S.: *Object-oriented Networks*. Prentice Hall, Englewood Cliffs, New Jersey, 1994

[Booz96] Booz Allen & Hamilton: *Telekommunikation in der Welt von morgen: Marktstrategien, Konzepte und Kompetenzen für das 21.* Jahrhundert. Institut für Medienentwicklung und Kommunikation (IMK), Verlagsgruppe Frankfurter Allgemeine Zeitung, 1997

[Brain00a] The Brain: *PersonalBrain Manual*. Version 1.74, included in PersonalBrain 1.74, TheBrain Technologies Corp. Santa Monica (CA), 2000

[Brain00b] The Brain: *BrainSDK Documentation*. Version 2.1, included in BrainSDK 2.1, TheBrain Technologies Corp. Santa Monica (CA), 2000

[Breugst98] Markus Breugst, Thomas Magedanz: *Mobile Agents – Enabling Technology for Active Intelligent Network Implementation*. IEEE Network Magazine, Special Issue on Active and Programmable Networks, Volume 12, No 3, May/June 1998

[Campolargo99] Mario Campolargo: *Information society – R&D Challenges and Opportunities*. Invited Speech, 4th International Symposium on Autonomous Decentralized Systems, ISADS'99, Tokyo, Japan, March 21-23, 1999

[Cerf00] Vinton G. Cerf: *Internet: Transforming the Developing World*. Planery Session, 5th IEEE Intelligent Network Workshop, IN2000, Cape Town, South Africa, May, 7-11, 2000

[CORBA] OMG: *The Common Object Request Broker: Architecture and Specification*. Minor Editorial Version CORBA 2.4.1, OMG Document 00-11-03, November 2000 (Revision 2.4 October 2000)

[CORBA-ES] OMG: *Event Service Specification*. Version 1.0, OMG Document 00-06-15, OMG, June 2000

[CORBA-MAN] CORBAMAN: *CORBA Management Agent Generic and NE specific information model*. Revision 1.1, March 6th, 2001
available at <http://www.corbaman.com> (last visited 03/14/02)

[CORBA-NotS] OMG: *Notification Service Specification*. Version 1.0, OMG Document 00-06-20, OMG, June 2000

[CORBA-NS] OMG: *Naming Service Specification*. Revised Version, OMG Document 01-02-65, OMG, February 2001

[CORBA-TMN] OMG: *Interworking between CORBA and TMN Systems Specification*. Version 1.0, OMG Document 00-08-01, OMG, August 2000

[DCE-RPC] Open Group Technical Standard: DCE1.1: *Remote Procedure Call*. Document Number C706, August 1997
<http://www.opengroup.org/publications/catalog/c706.htm> (visited on 09/07/2001)

[DIN96] DIN EN ISO 9241-10: *Grundsätze der Dialoggestaltung*. 1996

[DMTF-CIM] DMTF: *Common Information Model (CIM) Specification*. DMTF, Version 2.2, June 14, 1999

[Draft-ASN1NG] O. Dubuisson, P. H. Griffin, M. Perin, A. Sarma, B. Scott, A. Triglia: *ASN.1 for SMIng*. IETF Draft, Networking Group, November 13, 2001
Document expires May 13, 2002

[Draft-Config] M. MacFaden, J. Saperia, W. Tackabury: *Configuring Networks and Devices With SNMP*. IETF Draft, Snmpconf Working Group, Version 8, May 10, 2002
Document expires November 2002

[Draft-Hohno] H. Ohno, R.Atarashi: *The Emergency Communications on the internet*. IETF Draft, Version 0, November 2001

[Draft-IEPREP] Hal Folts: *Emergency Telecommunications Service in Evolving Networks*. IETF Draft, Version 0, February 15th, 2002

[Draft-LDUP] Ed Reed, Uppili Srinivasan: *LDAP Replication Architecture*. IETF Draft, Version 0.7, March 2002

[DRAFT-OPES] G. Tomlinson, R. Chen, M. Hofmann: *A Model for Open Pluggable Edge Services*. IETF Draft, November 20th, 2001

[Draft-SMIng] F. Strauss, J. Schoenwaelder: *SMIng - Next Generation Structure of Management Information*. IETF Draft, Networking Group, July 20, 2001
Document expired January 18th, 2002

[Draft-UUID] Paul J. Leach, Rich Salz: *UUIDs and GUIDs*. IETF Draft, Networking Group, February 4, 1998
Document expired since August 4, 1998, no new version available

[Dutkowski01] Simon Dutkowski: *Design Patterns for Distributed Communication Systems*. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), August 23, 2000

[Eckert97] K.P. Eckert, M. Festini, P. Schoo, G. Schürmann: *TANGRAM: Development of Object-oriented Frameworks for TINA-C-based Multimedia Telecommunication Applications*. Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems, ISADS'97, Berlin, Germany, April 9-11, 1997

[EU-FP6Draft] Commission of the European Communities: *Proposals for COUNCIL DECISIONS concerning the specific programs implementing the Framework Program 2002-2006 of the European Community for research, technological development and demonstration activities*. Presented by the Commission, COM(2001) 279 final, Brussels, Belgium, 30.05.2001
<http://www.cordis.lu/rtd2002/fp-debate/cec.htm> (last visited 11/01/01)

[Fahner02] Michael Fahner: *Information Visualization for Distributed Systems*. Diplomarbeit, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), January 28, 2002

[FODC] Free On-line Dictionary of Computing
<http://foldoc.doc.ic.ac.uk> (last visited 01/10/02)

[Fritzsch01] Wolfram Fritzsch: *Integrated Management of Distributed Components*. Diplomarbeit, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), June 17, 2001

[Funabashi00] Motoshiba Funabashi et.al.: *Development of Open Service Collborative Platform for Coming ECs*. by International Joint Efforts. SSGRR 2000, August 2000

[Geihs01] Kurt Geihs: *Middleware Challenges Ahead*. IEEE Computer, Volume 34, No 6, June 2001

[Hegering99] Hegering et.al.: *Integriertes Management vernetzter Systeme*. 1. Auflage, dpunkt – Verlag für digitale Technologie, Heidelberg, Germany, 1999

[Herman99] I.Herman, M.S.Marshall, G.Melancon, D.J.Duke, M Delest, J.-P.Domenger: *Skeletal Images as Visual Cues*. in Graph Visualisation. in Data Visualisation 99, Springer, Vienna, 1999

[Heuer97] Andreas Heuer: Objektorientierte Datenbanken: Konzepte, Modelle, Standards und Systeme, 2. Auflage, Addison Wesley Longman Verlag GmbH, Bonn, 1997

[Hewett96] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong and Verplank: *ACM SIGCHI Curricula for Human-Computer Interaction*. ACM SIGCHI, 1992, 1996
< http://sigchi.org/cdg/cdg2.html> (last visited 09/01/01)

[IANA-OS] IANA: *Operating System Names*. Last updated 04/29/02
<http://www.iana.org/assignments/operating-system-names> (last visited 05/13/02)

[IBM99] IBM: *Discovering Devices and Services in Home Networks*. IBM White Paper, June1999

[IEEE-1003.2] IEEE/ANSI Std 1003.2 & IEEE/ANSI 1003.2a-1999 (ISO/IEC 9945-2): *Information Technology-Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities*.

[IETF-RFC1157] J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin: *Simple Network Management Protocol (SNMP)*. IETF RFC 1157, May 01, 1990

[IETF-RFC1213] K. McCloghrie, M.T. Rose: *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*. IETF RFC 1213, Mar 01, 1991

[IETF-RFC1493] E. Decker, P. Langille, A. Rijsinghani, K. McCloghrie: *Definitions of Managed Objects for Bridges*. IETF RFC 1493, July, 1993

[IETF-RFC1738] T. Berners-Lee, L. Masinter, M. McCahill: *Uniform Resource Locators (URL)*. IETF RFC 1738, December 1994

[IETF-RFC1759] R. Smith, F. Wright, T. Hastings, S. Zilles, J. Gyllenskog: *Printer MIB*. IETF RFC 1759, March 1995

[IETF-RFC1777] W. Yeong, T. Howes, S. Kille: *Lightweight Directory Access Protocol*. IETF RFC 1777, March 1995

[IETF-RFC1905] J. Case, K. McCloghrie, M. Rose, S. Waldbusser: *Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF RFC 1905, January 1996

[IETF-RFC1906] J. Case, K. McCloghrie, M. Rose, S. Waldbusser: *Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)*. IETF RFC 1906, January 1996

[IETF-RFC1958] B. Carpenter et. Al.: *Architectural Principles of the Internet*. IETF RFC 1958, June 1996

[IETF-RFC2119] Bradner, S.: *Key words for use in RFCs to Indicate Requirement Levels*. BCP 14, IETF RFC 2119, March 1997.

[IETF-RFC2234] D. Crocker: *Augmented BNF for Syntax Specifications: ABNF*. IETF RFC 2234, November 1997

[IETF-RFC2251] M. Wahl, T. Howes, S. Kille: *Lightweight Directory Access Protocol (v3)*. IETF RFC 2251, December 1997

[IETF-RFC2287] Krupczak, C. and J. Saperia: *Definitions of System-Level Managed Objects for Applications*. IETF RFC 2287, February 1998.

[IETF-RFC2396] T. Berners-Lee, R. Fielding, L. Masinter: *Uniform Resource Identifiers (URI): Generic Syntax*. IETF RFC 2396, August 1998

[IETF-RFC2578] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser: *Structure of Management Information Version 2 (SMIv2)*. STD 58, IETF RFC 2578, April 1999.

[IETF-RFC2579] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser: *Textual Conventions for SMIv2*. STD 58, IETF RFC 2579, April 1999.

[IETF-RFC2616] R. Fielding et. al.: *Hypertext Transfer Protocol -- HTTP/1.1*. IETF RFC 2616, June, 1999

[IETF-RFC2929] D. Eastlake, E. Brunner-Williams, B. Manning: *Domain Name System (DNS) IANA Considerations*. IETF RFC, September, 2000

[IETF-RFC3216] C. Elliott, D. Harrington, J. Jason, J. Schoenwaelder, F. Strauss, W. Weiss: *SMIng Objectives*. IETF RFC 3216, December 2001

[Inet01] Fraunhofer Gesellschaft FOKUS: I-net – Individual-centric Networking. Verbundprojekt, Meilenstein 1, 2001

[ISO11578] ISO/IEC 11578:1996 : *Information technology - Open Systems Interconnection - Remote Procedure Call*. Edition 1
<http://www.iso.ch/cate/d2229.html> (visited on 09/07/2001)

[ISO14882] ISO/IEC 14882: *Programming language C++*. ISO, Geneva, Switzerland, September 1, 1998

[ISO14977] ISO/IEC 14977: *Information technology -- Syntactic metalanguage -- Extended BNF*. ISO, Geneva, Switzerland, 1996

[ISO8601] ISO 8801: *Data elements and interchange formats -- Information interchange -- Representation of dates and times*. ISO, Geneva, Switzerland, 2000

[ISO8879] ISO 8879: *Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML)*. ISO, Geneva, Switzerland, 1986

[ITU-M3000] ITU-T Recommendation M.3000 (02/00): *Telecommunications management network – Overview of TMN Recommendations*.

[ITU-M3010] ITU-T Recommendation M.3010 (02/00): *Telecommunications management network – Principles for a telecommunications management network*.

[ITU-Q1201] ITU-T Recommendation I.312/Q.1201 (10/92): *Principles of the Intelligent Network Architecture*. Geneva, Switzerland, October, 1992

[ITU-X200] ITU-T Recommendation X.200: *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. International Telecommunication Unit, Geneva, Switzerland, July 1994

[ITU-X208] ITU-T Recommendation X.200: *Open Systems Interconnection – Model and Notation – Specification of Abstract Syntax Notation One (ASN.1)*. International Telecommunication Unit, Geneva, 1993

[ITU-X210] ITU-T Recommendation X.210: *Information Technology – Open Systems Interconnection – Basic Reference Model: Conventions for the Definition of OSI Services*. International Telecommunication Unit, Geneva, November 1993

[ITU-X500] ITU-T Recommendation X.500 (1993): *Information technology – Open Systems Interconnection – The Directory: Overview of concepts, models and services*. International Telecommunication Unit, Geneva, 1993

[ITU-X501] ITU-T Recommendation X.501 (08/97): *Information Technology – Open Systems Interconnection – The Directory: Models*. International Telecommunication Unit, Geneva, August 1997

[ITU-X700] ITU-T Recommendation X.700: *Management Framework for Open Systems Interconnection (OSI) for CCITT Applications*. International Telecommunication Unit, Geneva, September 1992

[ITU-X701] ITU-T Recommendation X.701 (1997): *Information technology – Open Systems Interconnection – Systems management overview*.

[ITU-X710] ITU-T Recommendation X.710 (1997): *Information technology – Open Systems Interconnection – Common management information service*.

[ITU-X720] CCITT Recommendation X.720 (1992): *Information technology – Open Systems Interconnection – Structure of management information: Management information model*.

[ITU-X721] CCITT Recommendation X.721 (1992): *Information technology – Open Systems Interconnection – Structure of management information: Definition of management information*.

[ITU-X722] CCITT Recommendation X.722 (1992): *Information technology – Open Systems Interconnection – Structure of management information: Guidelines for the definition of managed objects*.

[ITU-X901] ITU-T Recommendation X.901 (1997): *Information technology – Open Distributed Processing – Reference model: Overview*.

[ITU-X920] ITU-T Recommendation X.920 (1997): *Information technology – Open Distributed Processing – Interface Definition Language*.

[JAVA-J2SE] Sun Microsystems: *J2SE API Specification Version: 1.3*. Sun Microsystems, Palo Alto (CA),2000

[JAVA-JDMK] Sun Microsystems: *Java Dynamic Management Kit White Paper*. Sun Microsystems, Palo Alto, 2000

[JAVA-JMX] Sun Microsystems: *Java Management Extensions – Instrumentation and Agent Specification 1.0*. Final Release, Sun Microsystems, Palo Alto, July, 2000

[JNDI-API] Sun Microsystems: *Java Naming and Directory Interface – Application Programming Interface (JNDI API)*. JNDI 1.2/Java 2 Platform, Standard Edition, v 1.3, Sun Microsystems, July 14, 1999

[John96] Bonnie E. John, David E. Kieras: *Using GOMS for Interface Design and Evaluation: Which Technique?* Association for Computing Machinery Inc., Pittsburgh (PA) / Ann Arbor (MI), 1996

[JXTA] JXTA home page <http://www.jxta.org> (last visited 05/20/02)

[Lee98] Tim Berners-Lee: *Web Architecture from 50,000 feet*. Created 1998, last updated 25 February 2002
<http://www.w3.org/DesignIssues/Architecture> (last visited 05/14/02)

[Linington95] Linington, P.: *Why objects have multiple interfaces*. ISO/IEC JTC1 SC21 / WG7 Draft answer to Q7/5, 1995

[LPG95] Sven Goldt, Sven van der Meer, Scott Burkett, Matt Welsh: The *Linux Programmer's Guide*. Version 0.4, published in "The New Book of LINUX", Just Computers!, Rohnert Park, CA, USA, 1995

[Luckenbach99] Thomas Luckenbach: *Seamless Integration of Infranetworks into the Internet: The I-Cube-C Project*. - in Proc. of the Home Networking 11053, London, 14-15 September, 1999

[Magedanz96] Magedanz, T., Popescu-Zeletin, R: *Intelligent Networks. Basic Technology, Standards and Evolution*. Int. Thomson Computer Press, London, 1996

[Magedanz99a] Thomas Magedanz: *Intelligent Network Evolution – Middleware Technologies for Open Service Architectures*. Habilitationsschrift, Technische Universität Berlin, Fachbereich Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), October, 1999

[Magedanz00] Thomas Magedanz et.al.: *Towards an Integrated Architecture fort he Harmonization of PSTN and Internet Services*. International Conference on Intelligence in Networks, ICIN 2000, Bordeaux, France, January 18-20, 2000

[Magedanz01] Thomas Magedanz: *Enhancing Parlay with Mobile Code Technologies*. Proc. of the 6th IEEE Intelligent Network Workshop, IN2001, Boston, MA, USA, May 6-9, 2001

[Maruyama00] Hiroshi Maruyama, Kent Tamura, Naohiko Uramoto: *XML and Java Developing Web Applications*. Addison-Wesley, Reading (MA), 2000

[Mbedi00] Mbedi, E.: *A CORBA-based Approach to access distributed SQL-databases*. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), February 26, 2000

[Millerl98] Eric Miller: *An Introduction to the Resource Description Framework*. In: D-Lib Magazine, May 1998

[MS-ADArch] Microsoft Corporation: *Active Directory Architecture*. White Paper, Microsoft Corporation, 2002
<http://www.microsoft.com/technet/prodtechnol/ad> (last visited 05/19/02)

[MSDN-UI01] Microsoft: *User Interface Design and Development*. Microsoft Developer Network MSDN
<http://msdn.microsoft.com/library/> (last visited 10/23/01)

[Müller02] Wolfgang Müller: *P2P in kommerziellen Anwendungen*. iX, Magazin für professionelle Informationstechnik, Nr. 4/02, HeiseVerlag

[Munzner00] Tamara Munzner: *Interactive Visualization of Large Graphics and Networks*. PhD Thesis, Stanford University, 2000

[Ninja] Ninja Project home page <http://ninja.cs.berkeley.edu> (last visited 05/20/02)

[NMF-GB909] Network Management Forum: *SMART TMN Technology Integration Map*. Network Management Forum, Document GB909, Issue 1, April, 1998

[OMG-OMA] Object Management Group: *A discussion of the Object Management Architecture*. OMG Document 00-06-41, OMG, January, 1997

[OMG-UML] Object Management Group: *OMG Unified Modeling Language Specification*. OMG, Version 1.3, First Edition, March, 2000

[Orfali96] Robert Orfali, Dan Harkey, Jeri Edwards: *The essential distributed object survival guide*. John Willey & Sons, New Yprk, 1996

[OSGi-FAQ] OSGi: *Frequently Asked Questions*.
< http://www.osgi.org/about/faqs.asp > (last visited 11/12/01)

[P614-D11] Eurescom Project P614, Implementation Strategies for Advanced Access Networks: *Deliverable 11 – Broadband Home Network for residential and small business*. Eurescom, December 1998

[P812-D1] Eurescom Project P812-GI, TMN Evolution – Service Provider's Needs for the Next Millennium: *Deliverable 1 – TMN Evolution*. Volume 2 of 2, Annexes, Eurescom, March, 1999

[Palme02] Jacob Palme: *Acomparision of ABNF, ASN.1, and XML*. Course Internet application protocols and Standards, last revised March 18th 2002
<http://www.dsv.du.se/jpalme/internet-course/Int-app-prot-kurs.html>
(last visited 05/19/02)

[Parlay-API] Parlay Group: *Framework Interfaces, Parlay Service View*. Parlay API, Version 2.1, June 26th, 2000
available at <http://www.parlay.org> (last visited 12/14/01)

[Parr95] T.J. Parr, R.W. Quong: *ANTLR: A Predicated –LL(k) Parser Generator*. Software-Practice and Experiences, Vol. 25(7), July 1995

[Pavlou99] G. Pavlou: *A Novel Approach for Mapping OSI-SM / TMN Model to ODP / OMG CORBA*. Proc. of Integrated Network Management VI, Proceedings of the IEEE/IFIP Integrated Management Symposium (IM '99), Boston, USA, May 1999

[Pentland99] Pentland, A.: *Perceptual Intelligence*. First International Symposium on Handheld and Ubiquitous Computing, HUC'99, Karlsruhe, Germany, September, 27-29, 1999

[Pfeifer97] Pfeifer, T.; Arbanowski, St.; Popescu-Zeletin, R.: *Resource Selection in Heterogeneous Communication Environments using the Teleservice Descriptor*. 4rd COST 237 Workshop, Lisboa, Dec. 15-19, 1997

[Pfeifer98] Pfeifer, T.; van der Meer, S.: *The Active Store providing Quality Enhanced Unified Messaging*. Proc. of the 5th Conference on Computer Communications, AFRICOM-CCDC'98, Tunis, October 20-22, 1998

[Pfeifer99] Pfeifer, T.: *Internet – Intranet – Infranet: A Modular Integrating Architecture*. Proc. Of the 7th IEEE Workshop on Future Trends of Distributed Computer Systems, FTDCS'99, Cape Town, South Africa, December 20-22, 1999

[Raymond95] Raymond, K: *Reference Model of Open Distributed Processing (RM-ODP): Introduction*. Proc. of the International Conference on Open Distributed Processing, ICODP'95, Brisbane, Australia, 20 - 24 February, 1995

[Rekesh99] John Rekesh: *UpnP, Jini and Salutation – A look at some popular coordination frameworks for future networking devices*. California Software Labs, June 17th, 1999 availeble at <http://cswl.com/whiteppr/tech> (last visited 05/01/02)

[Röhricht01] Röhricht, J., Schlögerl, C.: *cBusiness*. Addison Wesley, München, 2001

[Roth98] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Philip J. Stroffolino, John A. Kolojejchick & Carolyn Dunmire: *Visage: A User Interface Environment for Exploring Information*. MAYA Design Group, Pittsburgh (PA), 1998

[Salutation-OV] Salutation Consortium: *Overview*. <http://www.salutation.org/Overview.htm> (last visited 11/01/01)

[SAX] SAX Project <http://www.saxproject.org> (last visited 05/19/02)

[Scheer02] August Wilhelm Scheer, Thomas Feld, Sven Zang: *Vitamin C für Unternehmen – Collaborative Business*. Reihe „Kompendium der neuen BWL", Frankfurter Allgemeine Zeitung, Nr. 53, Seite 25, Frankfurt, 4. März 2002

[Schlosser97] Otto Schlosser, Jun Suzuki: *Mac OS 8 Human Interface Guidelines*. Apple Computer (CA), Technical Publications, 1997

[Schmid01] Alexander Schmid, Markus Völker, Eberhard Wolff: *IT-Ameisen – Software Zukunft: Peer-toPeer Systeme, Agenten, mobiler Code*. iX, Magazin für professionelle Informationstechnik, Nr. 8/01, HeiseVerlag

[Scowen93] Scowen, R.S.: *Extended BNF - A Generic Base Standard*. In Proc. 1993 Software Engineering Standards Symposium (SESS'93), Brighton, UK, 30 August - 3 September 1993. IEEE Computer Society Press

[Shirky00] Clay Shirky: *What is P2P … and what isn't*. <http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html> (last visited 11/01/01)

[Singh01] Mandeep Singh: *Integrated Management Services for Directory, Naming, and Specification of Distributed Components*. Diplomarbeit, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), December 20, 2001

[Steglich98] Steglich, St.: *Management of Distributed Objects in Middleware Platforms*. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), April 21, 1998

[Stroustrup92] Bjarne Stroustrup: *Die C++ Programmiersprache*. Second Edition, München, Addison-Wesley, Reading, Massachusetts, 1992

[SunOS-chmod] SunOS 5.6 manpage for command chmod, /usr/man/cat1/chmod.1

[Tannenbaum96] Andrew S. Tannenbaum: *Computer Networks*, Prentice Hall International, Third Edition, 1996

[Thai01] Thuan Thai, Hoang Q. Lam: *.NET Framework Essentials*. First Edition, O'Reilly & Associates, Sebastopol, CA, USA, 2001

[Tian01] Min Tian: *Applying Network Connection Technology for Dynamic Provisioning of User Interfaces*. Diplomarbeit, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), May 31, 2001

[TINA-BM] TINA-C: *TINA Business Model and Reference Points*. TINA-C Deliverable, TINA 1.0, Version 4.0, May 20, 1997

[TINAC] <http://www.tinac.com/about/principles_of_tinac.htm> (visited 11/01/01)

[TINA-CMC] TINA-C: *Computational Modeling Concepts*. TINA-C Deliverable, TINA 1.0, Version 3.2, Archiving Label TP_HC.012_3.2_96, TINA-C, May 17, 1996

[TINA-EMC] TINA-C: *Engineering Modeling Concepts (DPE Architecture)*. TINA-C Deliverable, Version 2.0, Archiving Label TB_NS.005_2.0_94, TINA-C, December 1994

[TINA-OCP] TINA-C: *Overall Concepts and Principles of TINA*. TINA-C Deliverable, TINA 1.0, Version 1.0, Document Label TB_MDC.018_1.0_94, TINA-C, February 17, 1995

[TINA-ODL] TINA-C: *TINA Object Definition Language Manual*. TINA-C Deliverable, TINA 1.0, Version 2.3, Archiving Label TR_NM.002_2.2_96, TINA-C, July 22, 1996

[Tiziana96] Tiziana Catarci, Shi-Kuo Chang, Maria F. Costabile, Stefano Levaldi, Giuseppe Santucci. *A Graph-based Framework for Multiparadigmatic Visual Access to Databases*. Universita di Roma "La Sapienza", Roma, 1996

[TMF-ACT01-99] TeleManagement Forum: *Requirements of Management of ORB-based Telecommunication Management Building Blocks*. TIM/ACT Working Document, Issue 1, August 2nd, 2000

[Tönnby00] Ingmar Tönnby: *It's all about co….* 7th International Conference on Intelligence in Services and Networks, IS&N 2000, Athens, Greece, February, 23-25, 2000

[vdMeer96] van der Meer, S.: *Dynamic Configuration Management of the Equipment in Distributed Communication Environments*. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, Fachgebiet für Offene Kommunikationssysteme (OKS), October 6, 1996

[vdMeer99a] van der Meer, S.; Arbanowski, St., Magedanz, M.: *An Approach for a 4th Generation Messaging System*. Proc. of the 4th International Symposium on Autonomous Decentralized Systems, ISADS'99, Tokyo, Japan, March 21-23, 1999

[vdmeer99b] van der Meer, S., Arbanowski, St., Popescu-Zeletin, R.: *Environment-aware Applications: Integrating Mobile Communications and Ubiquitous Computing*. Proc. of 7th Annual International conference on Advances in Communication and Control, COMCON 7, Athens, Greece, June28 – July 2, 1999

[vdMeer99c] van der Meer, S.; Arbanowski, St.; Popescu-Zeletin, R.: *A Platform for Environment-Aware Applications*. Proc. of the 1st International Symposium on Handheld and Ubiquitous Computing, HUC'99, Karlsruhe, Germany, September 27-29, 1999

[vdMeer99t1] van der Meer, S., Arbanowski, St.: *Management of Autonomous Decentralized Systems*. Tutorial at the 4th International Symposium on Autonomous Decentralized Systems, ISADS'99, Tokyo, Japan, March 21-23, 1999

[vdMeer00a] van der Meer, S., Arbanowski, St., Steglich, St., Popescu-Zeletin, R.: *The Human Communication Space – Towards I-centric Communications*. Workshop on "The What, Who, Where, When, Why and How of Context-awareness", CHI 2000, The Hague, Netherlands, April 1-6, 2000

[vdMeer00b] van der Meer, S., Arbanowski, St., Steglich, St.: *Flexible Control of Media Gateways for Service Adaptation*. Proc. of the 5th IEEE Intelligent Network Workshop, IN2000, Cape Town, South Africa, May, 7-11, 2000

[vdMeer00c] van der Meer, S., Arbanowski, St.: *Service Interoperability through advanced Media Gateways*. Proc. of 6th IFIP Conference on Intelligence in Networks, SmartNet 2000, Vienna, Austria, September 18-22, 2000

[vdMeer00t1] van der Meer, S., Arbanowski, St.: *Flexible Media and Content Adaptation for Communication Systems.* Proc. of the 5th IEEE Conference on Protocols for Multimedia Systems, PROMS 2000, Krakow, Poland, October 22-25, 2000

[vdMeer01a] van der Meer, S., Arbanowski, St.: *From Unified Messaging towards I-centric - Services for the Virtual Home Environment*. Proc. of the 6th IEEE Intelligent Network Workshop, IN2001, Boston, MA, USA, May 6-9, 2001

[W3C] World Wide Web Consortium Home Page
<http://www.w3.org> (last visited 11/01/01)

[W3C-DOM] World Wide Web Consortium: *Document Object Model*.
<http://www.w3.org/DOM> (last visited 05/19/02>

[W3C-DOM98] W3C: *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation 1 October 1998
<http://www.w3.org/TR/1998/RECDOM-Level-1-19981001>, (last visited 01/25/02)

[W3C-HTML] HyperText Markup Language Home Page; http://www.w3.org/MarkUp/
(last visited 11/12/01)

[W3C-URI] World Wide Web Consortium: *Naming and Addressing: URIs, URLs, ….*W3C Architecture Document, December 12th, 2001
<http://www.w3.org/Addressing/> (last visited 01/04/02)

[W3C-WS] World Wide Web Consortium: *Web Services Activity*
<http://www.w3.org/2002/ws/> (last visited 11/01/01)

[W3C-XML] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0* - W3C Recommendation, 10-February-1998;
<http://www.w3.org/TR/1998/REC-xml-19980210> (last visited 11/01/01)

[W3C-XML-10P] World Wide Web Consortium: *XML in 10 points*;
<http://www.w3.org/XML/1999/XML-in-10-points> (last visited 11/01/01)

[W3C-XSLT] World wide Web Consortium: XSL Transformations (XSLT), Version 1.0 - W3C Recommendation 16 November 1999;
<http://www.w3.org/TR/1999/REC-xslt-19991116> (last visited 11/01/01)

[Wright96] Peter C. Wright, Bob Fields, Michael D. Harrison: *Distributed Information Resources: A New Approach to Interaction Modelling*. Dept. of Computer Science University of York, Helsington/York, 1996

[WSI-Intro] WS-I: Introduction Presentation. WS-I, February 6, 2002
available at <http://www.ws-i.org/Documents.aspx> (last visited 03/13/02)

[WSI-Profiles] WS-I: Web Services Profiles – An Introduction. WS-I
available at <http://www.ws-i.org/Documents.aspx> (last visited 03/13/02)

[WSI-WWW] Web Services Interoperability Organization Home Page
<http://www.ws-i.org> (last visited 03/13/02)

[Zeltserman99] David Zeltserman: *A practical Guide to SNMPv3 and Network Management*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999

Acronyms

This chapter includes all acronyms that have been used within this document. This includes acronyms shown in figures, which are not always mentioned elsewhere in the document. The list also takes account of words and terms that are often misunderstood as acronyms or that look like acronyms. The actual meaning of those terms is described very shortly.

2PC	Two Phase Commit
3GB	3G Beyond
3GPP	3rd Generation Partnership Program
3PC	Three Phase Commit
ABNF	Augmented Backus-Naur Form
ACL	Agent Control Language
AD	Active Directory (Microsoft)
ADL	Application Definition Language
AN	Active Networks
ANSI	American National Standards Institute
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation 1
ASP	Application Service Provider
ASR	Automatic Speech Recognition
ATM	Asynchronous Transfer Mode
BCP	Best Current Practice
BER	Basic Encoding Rules
BNF	Backus-Naur Form
CC/PP	Composite Capabilities/Preference Profiles
CCITT	Comité Consultatif International de Télégraphique et Téléphonique
CDN	Content Delivery Network
CDR	Common Data Representation
CIM	Common Information Model
CMIP	Common Management Information Protocol
CMIS	Common Management Information Service
COPS	Common Open Policy Service
CORBA	Common Object Request Broker Architecture
COST	European Co-operation in the field of Scientific and Technical Research
CPE	Customer Premises Equipment
CR	Carriage Return
CVS	Concurrent Versions System
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DECT	Digital Enhanced Cordless Telecommunications
DEN	Directory Enabled Networks
DES	Data Encryption Standard
DIM	Desktop Management Interface
DIT	Directory Information Tree
DLL	Dynamic Link Library
DMTF	Distributed Management Task Force
DN	Distinguished Name
DNS	Domain Name Service
DNSS	Directory Naming and Specification Service
DOM	Document Object Model
DOS	Disc Operating System
DPE	Distributed Processing Environment
DSA	Directory System Agent
DSL	Digital Subscriber Line
DTD	Document Type Definition
DWIM	do what I mean
EBNF	Extended Backus-Naur Form
EIB	European Installation Bus

EJB	Enterprise Java Beans
FCAPS	Fault, Configuration, Accounting, Performance, and Security
FDDI	Fiber Distributed Data Interface
FOKUS	Research Institute for Open Communication Systems
FTP	File Transfer Protocol
GDMO	Guidelines for the Definition of Managed Objects
GIF	Graphics Interchange Format
GOMS	Goals, Operators, Methods, Selection rules
GPRS	General Packet Radio System
GSM	Global System for Mobile communications
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDL	Interface Definition Language
IEC	International Engineering Consortium
IEEE	Institute of Electrical and Electronic Engineers
IERS	International Earth Rotation Service
IETF	Internet Engineering Task Force
IFIP	International Federation for Information Processing
IIOP	Internet Inter-ORB Protocol
IN	Intelligent Networks
IN-NG	IN Next Generation
IOR	Interoperable Object Reference
IP	Internet Protocol
iPCSS	intelligent Personal Communication Support System
IPTel	IP Telephony
IrDA	Infrared Data Association
ISO	International Standardization Organization
IT	Information Technology
ITU	International Telecommunication Unit
IVR	Interactive Voice Response
J2EE	Java 2 Platform, Enterprise Edition
JAR	Java Archive
JDMK	Java Distributed Management Kit
JMX	Java Management Extension
JNDI	Java Naming and Directory Interface
JTC	Joint Technical Committee
KQML	Knowledge Query and Manipulation Language
LCMS	Lifecycle and Configuration Management Service
LDAP	Lightweight Directory Access Protocol
LF	Line Feed
MAMA	Middleware and Application Management Architecture
MDA	Model Driven Architecture
MGCP	Media Gateway Control Protocol
MIB	Management Information Base
MO	Managed Object
MOF	Managed Object Format
MOF	Meta Object Facility
NELS	Notification, Event, and Log Service
NGOSS	Next Generation Operation System Support
NRIM	Network Resource Information Model
NVL	Name-Value List
OCR	Optical Character Recognition
ODL	Object Definition Language
ODP	Open Distributed Processing
OKS	Offene Kommunikationssysteme (Open Communication Systems)
OMG	Object Management Group
ORB	Object Request Broker
OSA	Open Service Access
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PC	Personal Computer

PDA	Personal Digital Assistant
PDU	Protocol Data Unit
PHP	PHP Hypertext Preprocessor
POSIX	Portable Operating System Interface
QoS	Quality of Service
RCS	Revision Control System
RDN	Relative Distinguished Name
RFC	Request for Comment
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
RM-OSI	Reference Model for Open System Interconnection
RPC	Remote Procedure Call
RSA	Rivest, Shamir, and Adleman
SAP	Service Access Point
SAX	Simple API for XML
SDK	Software Development Kit
SDO	Super Distributed Object
SGML	Standard Generalized Markup Language
SIG	Special Interest Group
SIT	Specification Information Tree
SLA	Service Level Agreement
SMI	Structure of Management Information
SMIng	Structure of Management Information Next Generation (SNMP)
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
STD	Standard Document
TCP	Transmission Control Protocol
TID	Transaction Identifier
TINA	Telecommunication Information Networking Architecture
TINA-C	Telecommunication Information Networking Architecture Consortium
TM Forum	TeleManagement Forum
TMN	Telecommunication Management Network
TOM	Telecom Operations Map
TTS	Text To Speech
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
UMS	Unified Messaging System
UMTS	Universal Mobile Telecommunication System
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unified Identifier
VHE	Virtual Home Environment
VM	Virtual Machine
W3C	World Wide Web Consortium
WBEM	Web-Based Enterprise Management
WG	Working Group
WS	Web Services
WS-I	Web Services Interoperability Organization
WWW	World Wide Web
xADL	eXchange ADL
XAMAV	XML ADL MAMA Visualization tool
xDD	eXchange Data Definition
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformation
yacc	Yet Another Compiler Compiler
yp	Yellow Pages

Appendix A
Conventions in this Document

This appendix describes the conventions that have been applied to this document. Most conventions are taken from international standards or industry standards. Further explanations and in-depth descriptions of the used conventions can be found in the referenced documents.

A.1 Typographical Conventions

Specifications given in a formal syntax and code examples are presented in the following form:

```
interface CodeExample {
  action showIt([IN] string param1, [IN, OUT] param2);
}
```

An italic format is used to identify terms that are specific to the described topic. Terms given in a bold format are emphasized to indicate that they form an important part of the described topic.

A.2 Keywords that indicate Requirement Levels

In the Internet standard documents, IETF[1] RFCs[2], it is current practice to indicate the requirement level. [IETF-RFC2119] defines those words and describes the way they should be interpreted by IETF standards. These definitions have been applied to this document. The following list is taken from [IETF-RFC2119]:

1. MUST – This word, or the terms “REQUIRED” or “SHALL”, mean that the definition is an absolute requirement of the specification.
2. MUST NOT – This phrase, or the phrase “SHALL NOT”, mean that the definition is an absolute prohibition of the specification.
3. SHOULD – This word, or the adjective “RECOMMENDED”, mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
4. SHOULD NOT – This phrase, or the phrase “NOT RECOMMENDED” mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
5. MAY – This word, or the adjective “OPTIONAL”, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation, which does not include a particular option, MUST be prepared to interoperate with another implementation that does include the option, though perhaps with reduced functionality. In the same vein, an implementation that does include a particular option MUST be prepared to interoperate with another implementation that does not include the option (except, of course, for the feature the option provides.)

[1] Internet Engineering Task Force

[2] Request For Comment

A.3 Languages and Symbol Tables for Grammar Specifications

This document contains specifications of grammars that are presented in the Extended Backus-Naur Form (EBNF), the Augmented Backus-Naur Form (ABNF), and using symbols of the ANother Tool for Language Recognition (ANTLR). Specifications from the OMG[3] are presented in their original form, which is similar to EBNF. If not specified otherwise, all grammars are given in EBNF.

A.3.1 Extended Backus-Naur Form

EBNF is a meta-language and an ISO[4] standard described in [ISO14977]. A good introduction to EBNF can be found in [Scowen93]. Table A-1 shows the basic symbols and their meaning.

EBNF	Operator	Meaning
unquoted words		Non-terminal symbol
" ... "		Terminal symbol
'...'		Terminal symbol
(...)		Brackets to group symbols (it is obvious convenience to use brackets in their ordinary mathematical sense)
[...]		Optional symbols (the enclosed symbols may occur zero or one time)
{ ... }		Symbols repeated zero or more times (repetition)
{ ... }-		Symbols repeated one or more times
=	in	Defining symbol
;	post	Rule terminator
|	in	Alternative
,	in	Concatenation
-	in	Except
*	in	Occurrences of
(* ... *)		Comment
? ... ?		Special sequence

Table A-1: EBNF Symbols [Scowen93]

The following exceptions are made for every EBNF specification in this document.

- The original comment symbols ("(*" and "*)") are not used. Comments are similar to C++ single line comments. Each comment starts with "//" and ends automatically at the end of the line.
- The *concatenation* symbol is not used. The concatenation of symbols is expressed by white spaces.

A.3.2 ANTLR Symbols

The parser implemented for the Application Definition Language (ADL) is based on ANTLR. Although this tool accepts EBNF grammars, there are some differences between the symbols of EBNF presented in the last section and ANTLR symbols. Therefore, Table A-2 shows the ANTLR symbols.

[3] Object Management Group

[4] International Standardization Organization

Symbol	Description	Symbol	Description
(...)	Subrule	|	Alternative operator
(...)*	Closure subrule	..	Range operator
(...)+	Positive closure subrule	~	Not operator
(...)?	Optional	.	Wildcard
{ ... }	Semantic action	=	Assignment operator
[...]	Rule arguments	:	Label operator, rule start
{ ... }?	Semantic predicate	;	Rule end (termination symbol)
(...)=>	Syntactic predicate	< ... >	Element options

Table A-2: ANTLR Symbols [ANTLR-Man]

A.3.3 EBNF used for OMG IDL Specifications

Symbol	Description
::=	Is defined to be
|	Alternative
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times.
+	The preceding syntactic unit can be repeated one or more times.
{}	The enclosed syntactic units are grouped as a single syntactic unit.
[]	The enclosed syntactic units are optional – may occur zero or more times.

Table A-3: EBNF used for OMG IDL Specifications [CORBA]

A.4 Graphical Notations

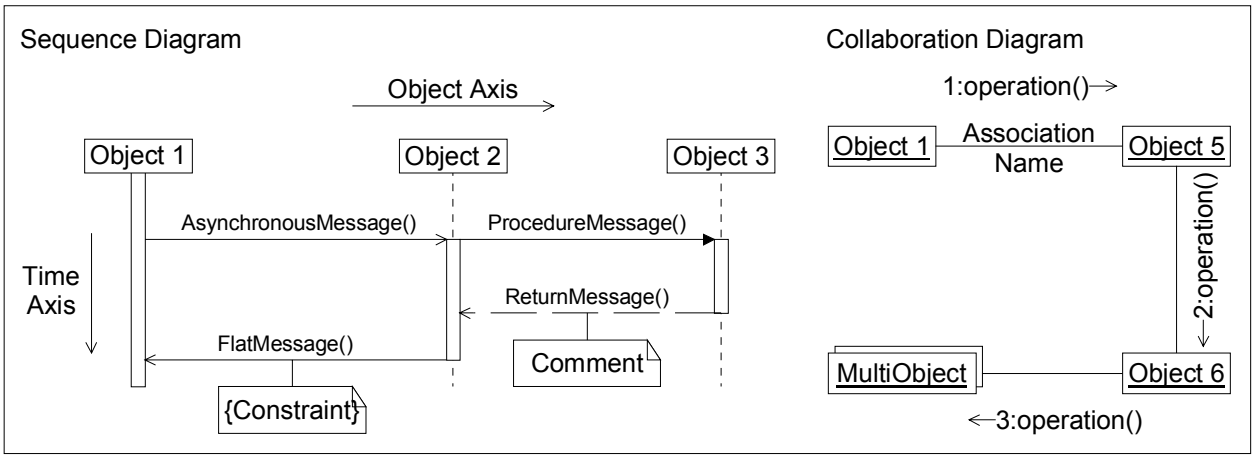

Figure A-1: Graphical Conventions – UML Sequence and Collaboration Diagram [OMG-UML]

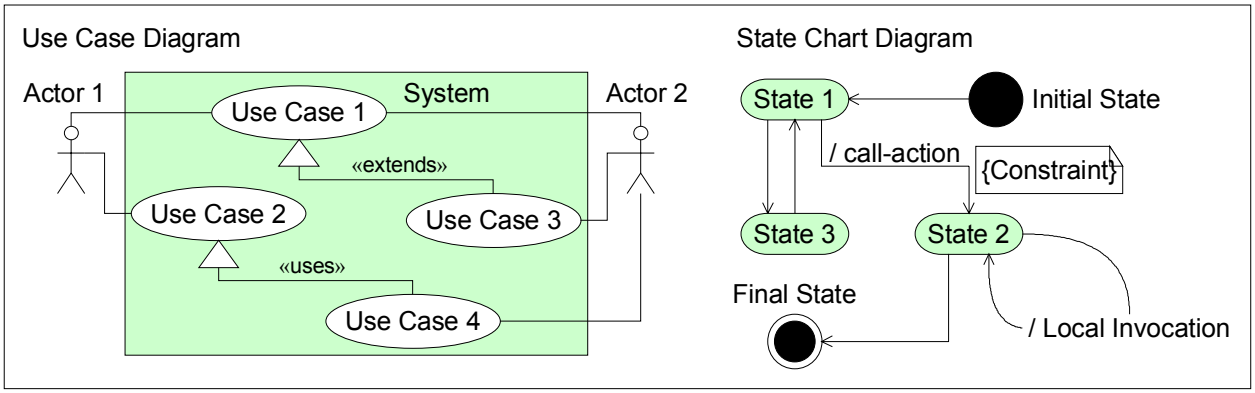

Figure A-2: Graphical Conventions – UML Use Case and State Chart Diagrams [OMG-UML]

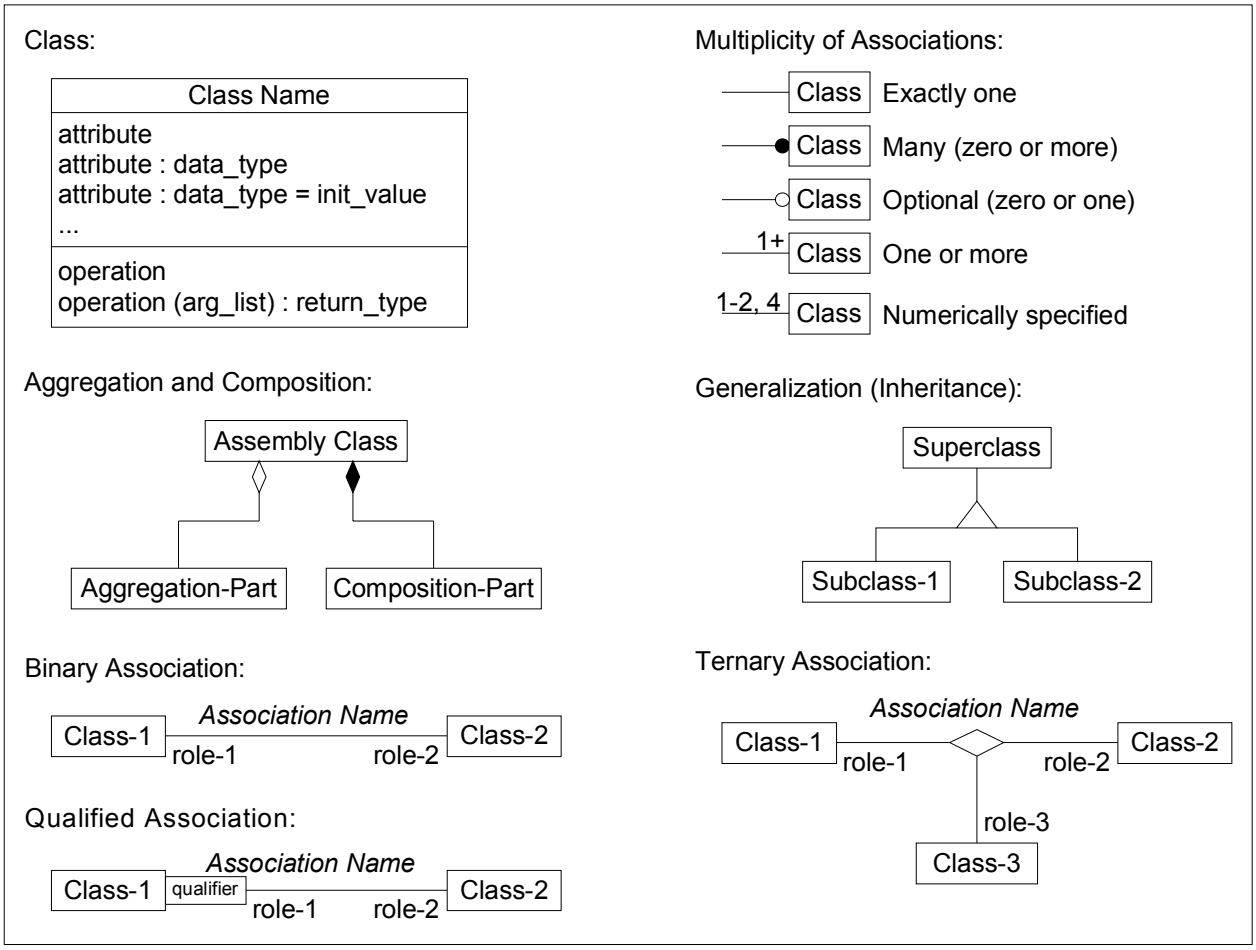

Figure A-3: Graphical Conventions – Information Modeling [OMG-UML]

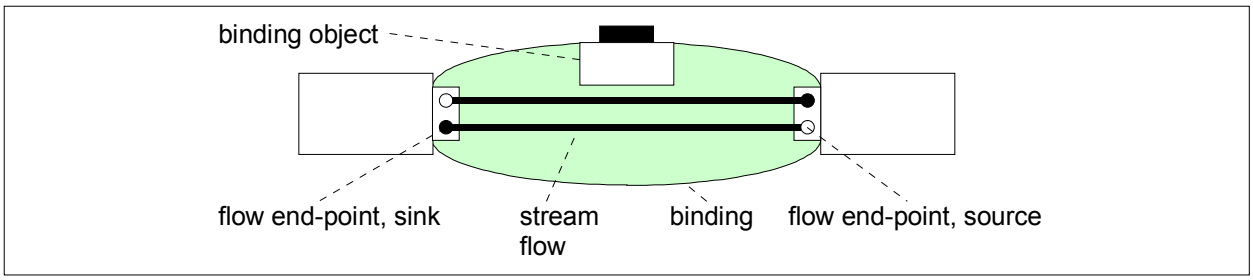

Figure A-4: Graphical Conventions – Explicit Computational Binding [TINA-CMC]

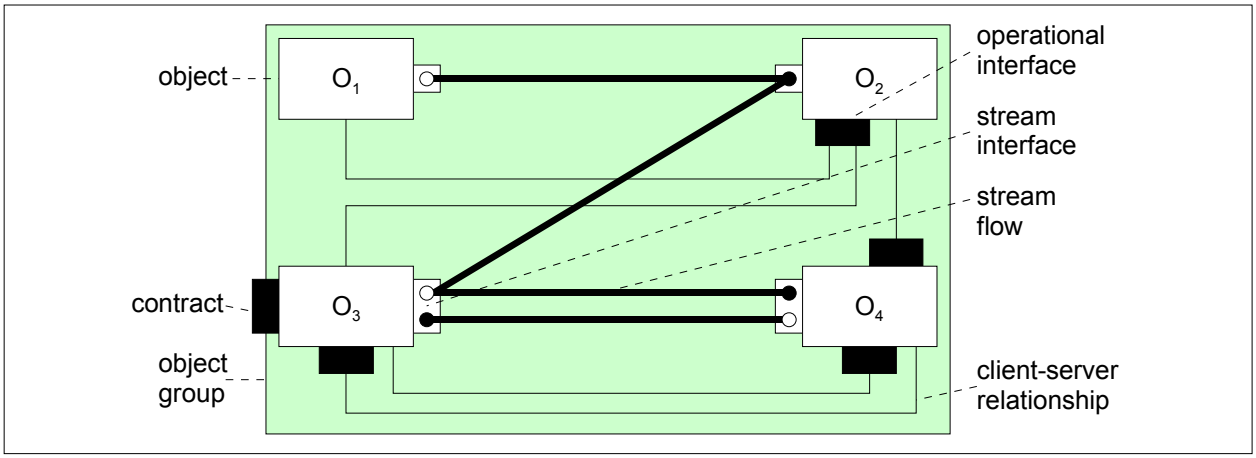

Figure A-5: Graphical Conventions – Computational Modeling [TINA-CMC]

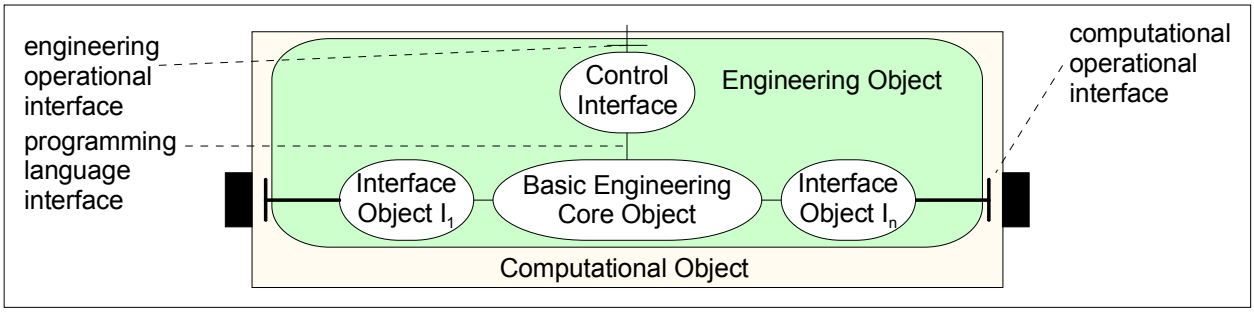

Figure A-6: Graphical Conventions – Engineering Modeling [Eckert97]

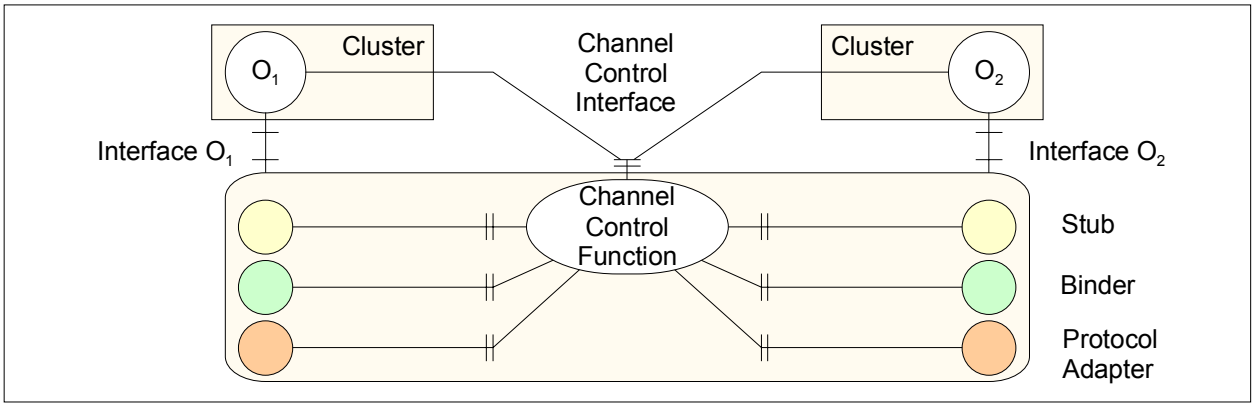

Figure A-7: Graphical Conventions – Explicit Engineering Binding [TINA-EMC]

A.5 References

References in this document are constructed by the main authors name and the year of publication. For documents that are produced by an organization, the reference contains the organizations acronym and an abbreviation of the document title. Documents of project milestones are a combination of an acronym of the project name and an acronym describing the deliverable number.

References from the World Wide Web (WWW) are accompanied with the Uniform Resource Locator (URL) that points to the actual document in the WWW. Additionally, each WWW reference is provided with information when the author of this document has last visited the related document. All referenced documents from the WWW can be requested from the author.

Appendix B
Application Definition Language

B.1 Lexical Conventions

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with dieresis
Ff	Upper/Lower-case F	Åå	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	Ææ	Upper/Lower-case A diphthong A with E
Hh	Upper/Lower-case H	Çç	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	Èè	Upper/Lower-case E with grave accent
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with dieresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with dieresis
Qq	Upper/Lower-case Q	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Öö	Upper/Lower-case O with dieresis
Vv	Upper/Lower-case V	Øø	Upper/Lower-case O with oblique stroke
Ww	Upper/Lower-case W	Ùù	Upper/Lower-case U with grave accent
Xx	Upper/Lower-case X	Úú	Upper/Lower-case U with acute accent
Yy	Upper/Lower-case Y	Ûû	Upper/Lower-case U with circumflex accent
Zz	Upper/Lower-case Z	Üü	Upper/Lower-case U with dieresis
		ß	Lower-case German sharp S
		ÿ	Lowercase Y with dieresis

Table B-1: ADL Lexical Conventions – Alphabetic Characters [CORBA]

Char.	Description	Char.	Description	Char.	Description
!	exclamation point	[	left square bracket	-	soft hyphen
"	double quote	\	reverse solidus, backslash	®	registered trade mark sign
#	number sign	]	right square bracket	¯	macron
$	dollar sign	^	circumflex	°	ring above, degree sign
%	percent sign	_	low line, underscore	±	plus-minus sign
&	ampersand	`	grave vulgar	²	superscript two
´	apostrophe	{	left curly bracket	³	superscript three
(	left parenthesis	|	vertical line	´	acute
)	right parenthesis	}	right curly bracket	µ	micro
*	asterisk	~	tilde	¶	pill crow
+	plus	¡	inverted exclamation mark	·	middle dot
,	comma	¢	cent sign	¸	cedilla
-	hyphen, minus sign	£	pound sign	¹	superscript one
.	period, full stop	¤	currency sign	º	masculine ordinal indicator
/	solidus, slash	¥	yen sign	»	right angle quotation mark
:	colon	¦	broken bar	¼	vulgar fraction 1/4
;	semicolon	§	section/paragraph sign	½	vulgar fraction 1/2
<	less-than sign	¨	apostrophe	¾	vulgar fraction 3/4
=	equals sign	©	copyright sign	¿	inverted question mark
>	greater-than sign	ª	feminine ordinal indicator	×	multiplication sign
?	question mark	«	left angle quotation mark	÷	division sign
@	commercial at	¬	not sign		

Table B-2: ADL Lexical Conventions – Graphic Characters [CORBA]

Description	Abbreviation	ISO 646 Octal Value	Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007	Backspace	BS	010
horizontal tab	HT	011	newline	NL, LF	012
vertical tab	VT	013	form feed	FF	014
carriage return	CR	015			

Table B-3: ADL Lexical Conventions – Formatting Characters [CORBA]

Char.	Descr.	Char.	Descr.	Char	Descr.	Char.	Descr.	Char.	Descr.
0	zero	1	one	2	two	3	three	4	four
5	five	6	six	7	seven	8	eight	9	nine

Table B-4: ADL Lexical Conventions – Numeric Characters

Description	Sequence	Description	Sequence	Description	Sequence
newline	\n	backslash	\\	horizontal tab	\t
question mark	\?	vertical tab	\v	single quote	\`
backspace	\b	double quote	\"	carriage return	\r
octal number	\ooo	form feed	\f	hexadecimal number	\xhh
alert	\a	Unicode character	\uhhhh		

Table B-5: ADL Lexical Conventions – Escape Sequences [CORBA]

B.2 Keywords

action	alterable	attribute	boolean	char	descr	double	FALSE
float	interface	long	mandatory	module	NULL	object	octet
optional	parameter	qualifier	required	scope	short	string	signed
struct	TRUE	type	typedef	unsigned	void		

Table B-6: ADL – Keywords

B.3 EBNF Grammar

The grammar of the Application Definition Language (ADL) was specified using the Extended Backus-Naur Form (EBNF) following the rules described in Appendix A.3. For the realization of the parser with ANother Tool for Language Recognition (ANTLR), this EBNF specification had been slightly rewritten following the symbols that ANTLR accepts (cf. Appendix A.3.2). The definitions in the Appendix B.3.1 up to B.3.8 are presented in EBNF; the specifications in Appendix B.3.9 follow ANTLR symbol rules.

B.3.1 Specification and Definition

```
01 specification     = {qualifier_def} {definition};
02 definition        = [qualifier_list]
                       (type_def SEMI | object SEMI | module SEMI);
```

B.3.2 Qualifier Definition

```
03 qualifier_def     = qualifier_header qualifier_body SEMI;
04 qualifier_header  = qualifier_literal identifier COLON;
05 qualifier_body    = qualifier_type COMMA qualifier_alt COMMA
                       qualifier_scope COMMA qualifier_descr;
06 qualifier_alt     = alterable_literal LPAREN alterable RPAREN;
07 alterable         = true_literal | false_literal;
08 qualifier_scope   = scope_literal
                       LPAREN scope_rank {COMMA scope_rank} RPAREN;
09 scope_rank        = LBRACK element COMMA rank RBRACK
10 element           = (module_literal    | object_literal
                       |interface_literal | action_literal
                       |attribute_literal | parameter_literal);
11 rank              = required_literal | mandatory_literal
```

```
                      | optional_literal;
12 qualifier_descr    = descr_literal LPAREN string_value RPAREN;
13 qualifier_type     = type_literal LPAREN
                        base_type_spec {array_declarator} default_value RPAREN;
14 default_value      = ASSIGN constant_value;
```

B.3.3 Qualifier List

```
15 qualifier_list     = LBRACK qualifier {COMMA qualifier} RBRACK;
16 qualifier          = identifier [qualifier_param];
17 qualifier_param    = LPAREN constant_value {COMMA constant_value} RPAREN;
```

B.3.4 Type Definition, Module, and Object

```
18 type_def           = (typedef_literal type_spec | struct_type_spec);
19 type_spec          = simple_type_spec identifier;
20 module             = module_header LCURLY {definition} RCURLY;
21 module_header      = module_literal identifier;
22 object             = object_header LCURLY object_body RCURLY;
23 object_header      = object_literal identifier [inheritance_spec];
24 object_body        = {object_export};
25 object_export      = [qualifier_list] (interface SEMI | type_def SEMI);
26 inheritance_spec   = COLON scoped_name;
27 scoped_name        = [[SCOPEOP] identifier {SCOPEOP identifier} SCOPEOP]
                        identifier;
```

B.3.5 Interface, Attribute, Action, and Parameter

```
28 interface          = interface_header LCURLY interface_body RCURLY;
29 interface_header   = interface_literal identifier;
30 interface_body     = {interface_export};
31 interface_export   = [qualifier_list]
                        (attribute SEMI | action SEMI | type_def SEMI);
32 attribute          = attribute_header;
33 attribute_header   = attribute_literal simple_type_spec identifier;
34 action             = action_header
                        LPAREN [param_decl {COMMA param_decl}] RPAREN;
35 action_header      = action_type_spec identifier;
36 action_type_spec   = simple_type_spec | void_literal;
37 param_decl         = [qualifier_list] simple_type_spec identifier;
```

B.3.6 Basic Types

```
38 struct_type_spec   = struct_literal identifier LCURLY {member}- RCURLY;
39 member             = [qualifier_list] type_spec SEMI;

40 simple_type_spec   = (scoped_name | base_type_spec) {array_declarator};
41 base_type_spec     = (numeric_type | char_type | string_type
                        |boolean_type | octet_type);
42 array_declarator   = LBRACK RBRACK;

43 numeric_type       = integer_type | floating_pt_type;
44 floating_pt_type   = float_literal | [long_literal] double_literal;
```

```
45 integer_type      = [unsigned_literal | signed_literal]
                       (short_literal | (long_literal [long_literal]));
46 char_type         = char_literal;
47 string_type       = string_literal;
48 boolean_type      = boolean_literal;
49 octet_type        = octet_literal;
```

B.3.7 Literals and Keywords

```
50 module_literal    = "module";
51 object_literal    = "object";
52 interface_literal = "interface"
53 typedef_literal   = "typedef";
54 action_literal    = "action";
55 qualifier_literal = "qualifier";
56 scope_literal     = "scope";
57 type_literal      = "type";
58 attribute_literal = "attribute";
59 parameter_literal = "parameter";
60 descr_literal     = "descr";
61 alterable_literal = "alterable";
62 required_literal  = "required";
63 mandatory_literal = "mandatory";
64 optional_literal  = "optional";

65 null_literal      = "NULL";
66 true_literal      = "TRUE";
67 false_literal     = "FALSE";

68 void_literal      = "void";
69 float_literal     = "float";
70 double_literal    = "double";
71 short_literal     = "short";
72 long_literal      = "long";
73 unsigned_literal  = "unsigned";
74 signed_literal    = "signed";

75 char_literal      = "char";
76 string_literal    = "string";

77 boolean_literal   = "boolean";
78 octet_literal     = "octet";
79 struct_literal    = "struct";

80 constant_value    = integer_value | char_value   | string_value
                     | boolean_value | binary_value | float_value;
```

B.3.8 Values

```
81 binary_value      = BINARY;
82 integer_value     = INT | OCTAL | HEX;
83 float_value       = FLOAT;
84 char_value        = CHAR_LITERAL;
85 string_value      = strings {strings} | null_literal;
86 strings           = STRING_LITERAL;
87 boolean_value     = true_literal | false_literal;
```

B.3.9 Specifications for Lexicographic Analysis

```
SEMI                = ';';
LPAREN              = '(';
RPAREN              = ')';
LBRACK              = '[';
RBRACK              = ']';
LCURLY              = '{';
RCURLY              = '}';
COLON               = ':';
COMMA               = ',';
ASSIGN              = '=';
SCOPEOP             = "::";
WS_                 = (' ' | '\t' | '\n' | '\r');
PREPROC_DIRECTIVE   = '#' (~'\n')* '\n';
SL_COMMENT          = "//" (~'\n')* '\n';
ML_COMMENT          = "/*"
                      (STRING_LITERAL
                      | CHAR_LITERAL | '\n' | '*' ~'/' | ~'*')*
                      "*/";
CHAR_LITERAL        = '\'' ( ESC | ~'\'' ) '\'';
STRING_LITERAL      = '"' (ESC | ~'"')* '"';
ESC                 = '\\' ( 'n' | 't' | 'v' | 'b' | 'r' | 'f' | 'a'
                           | '\\'| '?' | '\''| '"'
                           | ('0' | '1' | '2' | '3') (OCTDIGIT (OCTDIGIT)? )?
                           | 'x' HEXDIGIT (HEXDIGIT)?
                           );
VOCAB               = '\3'..'\377';
DIGIT               = '0'..'9';
OCTDIGIT            = '0'..'7';
HEXDIGIT            = ('0'..'9' | 'a'..'f' | 'A'..'F');
BINARYDIGIT         = ('0' | '1');
HEX                 = ("0x" | "0X") (HEXDIGIT)+;
BINARY              = ("b" | "B") (BINARYDIGIT)+;
INT                 = (DIGIT)+                  // base-10
                      ['.' (DIGIT)* [('e' | 'E') ['+' | '-'] (DIGIT)+]
                                    |('e' | 'E') ['+' | '-'] (DIGIT)+
                      ];
FLOAT               = '.' (DIGIT)+ [('e' | 'E') ['+' | '-'] (DIGIT)+];
IDENT               = ('a'..'z' | 'A'..'Z' | '_')
                      ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
```

B.4 xADL

The Document Type Definition (DTD) for the eXchange ADL (xADL) was realized following the XML[1] recommendations of the W3C[2] [W3C-XML]. This DTD was the basis for the EBNF. For the realization of the parser with ANTLR, this EBNF specification had been slightly rewritten following the symbols that ANTLR accepts (cf. Appendix A.3.2). The definitions in the Appendix B.4.1 are given as XML DTD, the specification in Appendix B.4.2 and B.4.3 are presented in EBNF.

[1] eXtensible Markup Language

[2] World Wide Web Consortium

B.4.1 Document Type Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT collection (specification | module    | object    | qualifierdef
                     |typedef       | interface | attribute | action
                     |parameter     | member)*>
<!ELEMENT specification (qualifierdef*, (typedef | object | module)*)>
<!ATTLIST specification
    name CDATA #IMPLIED
    distinguished_name CDATA #IMPLIED
    uuid CDATA #IMPLIED>
<!ELEMENT qualifierdef (scope+, description)>
<!ATTLIST qualifierdef
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    base_type (short | long | long_long | long_double | double
              |float | char | string | boolean | octet) #REQUIRED
    array_dim CDATA #IMPLIED
    signed (false | true) #IMPLIED
    default_value CDATA #REQUIRED
    alterable (false | true) #REQUIRED>
<!ELEMENT module (qualifier*, (typedef | object | module)*)>
<!ATTLIST module
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED>
<!ELEMENT object (qualifier*, (interface | typedef)*)>
<!ATTLIST object
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    extends CDATA #IMPLIED>
<!ELEMENT interface (qualifier*, (attribute | action | typedef)*)>
<!ATTLIST interface
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED>
<!ELEMENT attribute (qualifier*)>
<!ATTLIST attribute
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    type CDATA #IMPLIED
    base_type (short | long | long_long | long_double | double
               |float | char | string | boolean | octet) #IMPLIED
    signed (false | true) #IMPLIED
    array_dim CDATA #IMPLIED>
<!ELEMENT action (qualifier*, parameter*)>
<!ATTLIST action
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    type CDATA #IMPLIED
    base_type (short | long | long_long | long_double | double
               |float | char | string | boolean | octet | void) #IMPLIED
    signed (false | true) #IMPLIED
    array_dim CDATA #IMPLIED>
<!ELEMENT parameter (qualifier*)>
<!ATTLIST parameter
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    type CDATA #IMPLIED
    base_type (short | long | long_long | long_double | double
               |float | char | string | boolean | octet) #IMPLIED
    signed (false | true) #IMPLIED
    array_dim CDATA #IMPLIED>
<!ELEMENT typedef (qualifier*, member*)>
```

```
<!ATTLIST typedef
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    type CDATA #IMPLIED
    base_type (short | long | long_long | long_double | double
              |float | char | string | boolean | octet | struct) #IMPLIED
    signed (false | true) #IMPLIED
    array_dim CDATA #IMPLIED>
<!ELEMENT qualifier (constant_value*)>
<!ATTLIST qualifier
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED>
<!ELEMENT scope EMPTY>
<!ATTLIST scope
    element (module    | object | interface
            |attribute | action | parameter) #REQUIRED
    rank (optional | required | mandatory) #REQUIRED>
<!ELEMENT member (qualifier*)>
<!ATTLIST member
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    type CDATA #IMPLIED
    base_type (short | long | long_long | long_double | double
              |float | char | string | boolean | octet) #IMPLIED
    signed (false | true) #IMPLIED
    array_dim CDATA #IMPLIED>
<!ELEMENT description (#PCDATA)>
<!ELEMENT constant_value (#PCDATA)>
```

B.4.2 EBNF Grammar

```
start            = xmlstart doctype specification;
xmlstart         = LT_ QUESTION oxml_literal
                   {STRING_LITERAL ASSIGN STRING_LITERAL} QUESTION GT;
doctype          = LT_ NOT odoctype_literal specification_literal
                   IDENT STRING_LITERAL GT;
specification    = LT_ specification_literal name GT
                   {qualifierdef} {type_def | object | module}-
                   LT_ DIV specification_literal GT;
qualifierdef     = (LT_ qualifierdef_literal
                     {name     | d_name        | base_type | array_dim
                     |x_signed | default_value | alterable}
                   GT) {scope | description}
                   LT_ DIV qualifierdef_literal GT;
scope            = LT_ scope_literal {element | rank} DIV GT
                   {LT_ scope_literal {element | rank} DIV GT};
description      = LT_ description_literal GT STRING_LITERAL
                   LT_ DIV description_literal GT;
module           = LT_ module_literal {name | d_name} GT
                   [qualifier] {type_def | object | module}-
                   LT_ DIV module_literal GT;
object           = LT_ object_literal (name | d_name | extends_o)* GT
                   [qualifier] {interface | type_def}
                   LT_ DIV object_literal GT;
interface        = LT_ interface_literal {name | d_name} GT
                   [qualifier] {attribute | action | type_def}
                   LT_ DIV interface_literal GT;
attribute        = LT_ attribute_literal
                   {name | d_name | type | base_type | x_signed | array_dim}
                   GT [qualifier] LT_ DIV attribute_literal GT;
```

```
action           = LT_ action_literal
                   {name | d_name | type | base_type | x_signed | array_dim}
                   GT [qualifier] [parameter {parameter}]
                   LT_ DIV action_literal GT;
parameter        = LT_ parameter_literal
                   {name | d_name | type | base_type | x_signed | array_dim}
                   GT [qualifier] LT_ DIV parameter_literal GT;
type_def         = LT_ typedef_literal
                   {name | d_name | type | base_type | x_signed | array_dim}
                   GT {qualifier] {member} LT_ DIV typedef_literal GT;
member           = LT_ member_literal
                   {name | d_name | type | base_type | x_signed | array_dim}
                   GT [qualifier] LT_ DIV member_literal GT
                   { LT_ member_literal
                    {name | d_name | type | base_type | x_signed | array_dim}
                     GT [qualifier] LT_ DIV member_literal GT
                   };
qualifier        = LT_ qualifier_literal {name | d_name} GT
                   [constant_value {constant_value}]
                   LT_ DIV qualifier_literal GT
                   { LT_ qualifier_literal {name | d_name} GT
                     [constant_value {constant_value}]
                     LT_ DIV qualifier_literal GT
                   };
constant_value   = LT_ constantvalue_literal GT
                   (integer_value | char_value | boolean_value
                   |binary_value  | float_value
                   |STRING_LITERAL {STRING_LITERAL}
                   ) LT_ DIV constantvalue_literal GT;

default_value    = defaultvalue_literal ASSIGN STRING_LITERAL;
extends_o        = extends_literal ASSIGN STRING_LITERAL;
element          = element_literal ASSIGN STRING_LITERAL;
rank             = rank_literal ASSIGN STRING_LITERAL;
alterable        = alterable_literal ASSIGN STRING_LITERAL;
name             = name_literal ASSIGN STRING_LITERAL;
type             = type_literal ASSIGN STRING_LITERAL;
base_type        = basetype_literal ASSIGN STRING_LITERAL;
x_signed         = signed_literal ASSIGN STRING_LITERAL;
array_dim        = arraydim_literal ASSIGN STRING_LITERAL;
d_name           = distinguishedname_literal ASSIGN STRING_LITERAL;

specification_literal     = "specification";
module_literal            = "module";
object_literal            = "object";
interface_literal         = "interface";
typedef_literal           = "typedef";
action_literal            = "action";
qualifier_literal         = "qualifier";
qualifierdef_literal      = "qualifierdef";
scope_literal             = "scope";
type_literal              = "type";
attribute_literal         = "attribute";
parameter_literal         = "parameter";
description_literal       = "description";
alterable_literal         = "alterable";
required_literal          = "required";
mandatory_literal         = "mandatory";
optional_literal          = "optional";
signed_literal            = "signed";
name_literal              = "name";
distinguishedname_literal = "distinguished_name";
basetype_literal          = "base_type";
```

```
extends_literal           = "extends";
arraydim_literal          = "array_dim";
rank_literal              = "rank";
element_literal           = "element";
member_literal            = "member";
constantvalue_literal     = "constant_value";
defaultvalue_literal      = "default_value";
null_literal              = "null";
false_literal             = "false";
true_literal              = "true";
oxml_literal              = "xml";
odoctype_literal          = "DOCTYPE";

binary_value     = BINARY;
integer_value    = INT | OCTAL | HEX;
float_value      = FLOAT;
char_value       = CHAR_LITERAL;
strings          = STRING_LITERAL;
boolean_value    = true_literal | false_literal;
```

B.4.3 Specifications for Lexicographic Analysis

```
QUESTION            = '?';
ASSIGN              = '=';
LT_                 = '<';
GT                  = '>';
DIV                 = '/';
WS_                 = (' ' | '\t' | '\n' | '\r');
CHAR_LITERAL        = '\'' ( ESC | ~'\'' ) '\'';
COMMENT             = "<!--" COMMENT_DATA "-->";
STRING_LITERAL      = '"' (ESC | ~'"')* '"';
ESC                 = '\\' ( 'n' | 't' | 'v' | 'b' | 'r' | 'f' | 'a'
                           | '\\'| '?' | '\''| '"'
                           | ('0' | '1' | '2' | '3') (OCTDIGIT (OCTDIGIT)? )?
                           | 'x' HEXDIGIT (HEXDIGIT)?
                           );
VOCAB               = '\3'..'\377';
DIGIT               = '0'..'9';
OCTDIGIT            = '0'..'7';
HEXDIGIT            = ('0'..'9' | 'a'..'f' | 'A'..'F');
BINARYDIGIT         = ('0' | '1');
HEX                 = ("0x" | "0X") (HEXDIGIT)+;
BINARY              = ("b" | "B") (BINARYDIGIT)+;
INT                 = (DIGIT)+                  // base-10
                      ['.' (DIGIT)* [('e' | 'E') ['+' | '-'] (DIGIT)+]
                                    |('e' | 'E') ['+' | '-'] (DIGIT)+
                      ];
FLOAT               = '.' (DIGIT)+ [('e' | 'E') ['+' | '-'] (DIGIT)+];
```

Appendix C
MAMA Specifications

This appendix contains the ADL[1] definitions for a MAMA[2] system. The definitions start with the skeleton, which in fact is a file that includes all specifications. Except for the skeleton, the definitions contain no descriptive qualifiers. Furthermore, the qualifiers SpecStatus and Status are only presented when they are not used with their default values ("current" for SpecStatus and "required" for Status).

```
#include "qualifier_def.adl" // qualifier definitions

[Contact(     "Sven van der Meer, Wolfram Fritzsch, Mandeep Singh Multani"
              "                  [vdmeer|fritzsch|ricky]@cs.tu-berlin.de]"),
 Organization("                                Technical University Berlin"),
 Description("The module MAMA contains all specifications of the Core Model"
             "including the MAMA protocol, the MAMA API, and the basic"
             "application services."
             "To improve the understanding of the specifications, the"
             "following naming conventions are applied:"
             "- module identifiers start with an 'm'"
             "- object identifiers start with an 'o'"
             "- interface identifiers start with an 'i'"
             "- attribute identifiers start with an 'a'"
             "- type definitions start with an 't'"
             "- structure identifiers start with an 's'"
             "- no special recommendations are given for actions and"
             "  parameters."),
 Version(1), Revision(0), SpecStatus("current"), Status("required")]
module MAMA{

  #include "typedefs.adl"    // start with the basic type definitions
  #include "protocol.adl"    // the MAMA protocol, in form of an 'abstract'
                             // object
  #include "api.adl"         // the MAMA API, in form of an 'abstract' object
  #include "entity-mgmt.adl" // MAMA entity management, in form of an
                             // 'abstract' object
  #include "dnss.adl"        // Directory, Naming, and Specification Service
  #include "nels.adl"        // Notification, Event, and Log Service
  #include "lcms.adl"        // Lifecycle and Configuration Management Service

  #include "mgmt.adl"        // Definitions for the construction of Management
                             // Systems with MAMA
};
```

[1] Application Definition Language

[2] Middleware and Application Management Architecture

C.1 Core Model

C.1.1 Qualifiers

Qualifier	Type	Default	Alterable	Module	Object	Interface	Action	Attribute	Parameter
Abstract	boolean	FALSE	FALSE	-	Optional	-	-	-	-
ArrayType	string	bag	FALSE	-	-	-	-	Optional	Optional
Behavior	string	NULL	FALSE	-	Mandatory	Mandatory	Mandatory	Optional	-
BitMap	string[]	NULL	FALSE	-	-	-	-	Optional	Optional
BitValues	string[]	NULL	FALSE	-	-	-	-	Optional	Optional
Contact	string	NULL	FALSE	Mandatory	-	-	-	-	-
Counter	boolean	FALSE	FALSE	-	-	-	-	Optional	-
Description	string	NULL	FALSE	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	-
DisplayHint	string	NULL	TRUE	-	Optional	Optional	-	Optional	-
DisplayName	string	NULL	TRUE	-	-	-	-	Optional	-
Group	string	NULL	TRUE	-	Optional	Optional	Optional	Optional	-
History	string	NULL	FALSE	Optional	-	-	-	-	-
In	boolean	TRUE	FALSE	-	-	-	-	-	Optional
MaxLen	long	1024	TRUE	-	-	-	-	Optional	Optional
MaxValue	long	1024	TRUE	-	-	-	-	Optional	Optional
MinLen	long	0	TRUE	-	-	-	-	Optional	Optional
MinValue	long	0	TRUE	-	-	-	-	Optional	Optional
Organization	string	NULL	FALSE	Mandatory	-	-	-	-	-
Out	boolean	FALSE	TRUE	-	-	-	-	-	Optional
Owner	string	NULL	TRUE	-	Optional	Optional	Optional	Optional	-
Permission	octet	0755	TRUE	-	Optional	Optional	Optional	Optional	-
Quality	string	NULL	FALSE	-	Optional	Optional	Optional	-	-
RegisteredAs	string	NULL	FALSE	Optional	Optional	Optional	-	-	-
Revision	integer	0	FALSE	Mandatory	Optional	Optional	-	-	-
SpecStatus	string	current	FALSE	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	Optional
Status	string	optional	FALSE	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	Optional
StepIndex	long	1	FALSE	-	-	-	-	Optional	-
Units	string	NULL	FALSE	-	-	-	-	Optional	Optional
Usage	string	NULL	FALSE	-	Mandatory	Mandatory	Mandatory	-	-
ValueMap	string[]	NULL	FALSE	-	-	-	-	Optional	Optional
Values	string[]	NULL	FALSE	-	-	-	-	Optional	Optional
Version	integer	0	FALSE	Mandatory	Mandatory	Mandatory	-	-	-
Wildcards	boolean	FALSE	FALSE	-	-	-	-	Optional	Optional
xmlDTD	string	NULL	TRUE	-	-	Optional	Optional	Optional	-

Table C-1: Core Model – Qualifier Matrix

```
qualifier Abstract: type(boolean = FALSE), alterable(FALSE),
    scope([object, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/abstract.php");

qualifier ArrayType: type(string = "bag"), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/arraytype.php");

qualifier Behavior: type(string = NULL), alterable(FALSE),
    scope([object, mandatory], [interface, mandatory],
          [action, mandatory], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/behavior.php");

qualifier BitMap: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/bitmap.php");

qualifier BitValues: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/bitvalues.php");

qualifier Contact: type(string = NULL), alterable(FALSE),
    scope([module, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/contact.php");

qualifier Counter: type(boolean = FALSE), alterable(FALSE),
    scope([attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/counter.php");

qualifier Description: type(string = NULL), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory], [action, optional],
          [attribute, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/description.php");

qualifier DisplayHint: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/displayhint.php");

qualifier DisplayName: type(string = NULL), alterable(TRUE),
    scope([attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/displayname.php");

qualifier Group: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/group.php");

qualifier History: type(string = NULL), alterable(FALSE),
    scope([module, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/history.php");

qualifier In: type(boolean = TRUE), alterable(FALSE),
    scope([parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/in.php");

qualifier MaxLen: type(unsigned long = 1024), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/maxlen.php");

qualifier MaxValue: type(signed long = 1024), alterable(TRUE),
```

```
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/maxvalue.php");

qualifier MinLen: type(long = 0), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/minlen.php");

qualifier MinValue: type(long = 0), alterable(TRUE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/minvalue.php");

qualifier Organization: type(string = NULL), alterable(TRUE),
    scope([module, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/organization.php");

qualifier Out: type(boolean = FALSE), alterable(FALSE),
    scope([parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/out.php");

qualifier Owner: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/owner.php");

qualifier Permissions: type(octet = 0755), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional], [attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/permissions.php");

qualifier Quality: type(string = NULL), alterable(TRUE),
    scope([object, optional], [interface, optional],
          [action, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/quality.php");

qualifier RegisteredAs: type(string = NULL), alterable(FALSE),
    scope([module, optional], [object, optional],
          [interface, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/registeredas.php");

qualifier Revision: type(short = 0), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/revision.php");

qualifier SpecStatus: type(string = "current"), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory], [action, mandatory],
          [attribute, mandatory], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/specstatus.php");

qualifier Status: type(string = "optional"), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory],  [action, mandatory],
          [attribute, mandatory] , [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/status.php");

qualifier StepIndex: type(long = 1), alterable(FALSE),
    scope([attribute, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/stepindex.php");

qualifier Units: type(string = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/units.php");
```

```
qualifier Usage: type(string = NULL), alterable(FALSE),
    scope([object, mandatory], [interface, mandatory],
          [action, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/usage.php");

qualifier ValueMap: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/valuemap.php");

qualifier Values: type(string[] = NULL), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/values.php");

qualifier Version: type(short = 0), alterable(FALSE),
    scope([module, mandatory], [object, mandatory],
          [interface, mandatory]),
    descr("http://www.vandermeer.de/mama/doc/q/version.php");

qualifier Wildcards: type(boolean = FALSE), alterable(FALSE),
    scope([attribute, optional], [parameter, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/wildcards.php");

qualifier xmlDTD: type(string = NULL), alterable(FALSE),
    scope([interface, optional], [attribute, optional],
          [action, optional]),
    descr("http://www.vandermeer.de/mama/doc/q/xmlDTD.php");
```

C.1.2 Recommended Values for the Qualifier Units

Recommended Values	Recommended Values
Bits, KiloBits, MegaBits, GigaBits	Bits per Second
Bytes, KiloBytes, MegaBytes, GigaBytes, Words, DoubleWords, QuadWords	Cycles, Revolutions, Revolutions per Minute, Revolutions per Second
Degrees C, Tenths of Degrees C, Hundredths of Degrees C, (the same with Degrees F and Degrees K), Color Temperature	Gauss, Gilberts, Henrys, MilliHenrys, Farads, MilliFarads, MicroFarads, PicoFarads
Volts, MilliVolts, Tenths of MilliVolts, Amps, MilliAmps, Tenths of MilliAmps, Watts, Milli-WattHours	Minutes, Seconds, Tenths of Seconds, Hundredths of Seconds, MicroSeconds, MilliSeconds, Nano-Seconds
Joules, Coulombs, Newtons	Lumen, Lux, Candelas
Pounds, Pounds per Square Inch	Hours, Days, Weeks
Hertz, MegaHertz	Pixels, Pixels per Inch
Counts per Inch	Percent, Tenths of Percent, Hundredths of Percent
Meters, Centimeters, Millimeters, Cubic Meters, Cubic Centimeters, Cubic Millimeters	Inches, Feet, Cubic Inches, Cubic Feet Ounces, Liters, Fluid Ounces
Radians, Steradians, Degrees	Gravities, Pounds, Foot-Pounds
Ohms, Siemens	Moles, Becquerels, Parts per Million
Decibels, Tenths of Decibels	Grays, Sieverts

Table C-2: Qualifiers – Recommended Values for Units [DMTF-CIM]

C.1.3 Type Definitions

```
[ValueMap("0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11",
          "12", "13", "18", "19"),
 Values("inconsistent", "char", "string", "boolean", "octet", "short",
        "ushort", "long", "ulong", "longlong", "ulonglong", "float",
        "double", "longdouble", "array", "struct")]
typedef unsigned short tDataType;

[ValueMap("0", "1", "2", "4"),
 Values("none", "read", "write", "exec")]
typedef unsigned short tAccessFlag;

struct sNamedValue{
  string name;
  string value;
  tDataType nvDataType;
  tAccessFlag nvAccessFlag;
};
typedef sNamedValue[] tNameValueList;

[Description("URL according to RFC 1738]
typedef string tURL;

[ValueMap("0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
          "100", "101", "102"),
 Values("Specification", "Module", "Object", "Interface", "Attribute",
        "Action", "Parameter", "Qualifier", "Type Definition",
        "Qualifier Declaration", "DirectoryEntry",
        "DirectoryEntryInstance", , "DirectoryEntryAlias")]
typedef short tElementType;

[ValueMap("0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11",
          "12"),
 Values("standard", "manager", "sub-manager", "agent", "sub-agent",
        "dynamic-mo", "static-mo", "gui", "nels", "nels-manager", "dnss"]
typedef short tEntityType;

typedef string tPath;

[ValueMap("0", "100", "200", "300", "400"),
 Values("standard", "corba", "java", "jini", "upnp")]
typedef short tMiddleware;

[ValueMap("0", "200", "201", "300", "400", "500", "600", "700", "800"),
 Values("unspecified", "ADL", "xADL", "CORBA-IDL", "DCOM-IDL", "TINA-ODL",
        "JAVA", "SNMP-SMI", "OSI-GDMO")]
typedef short tSpecLanguage;

[ValueMap("0", "100", "200", "300"),
 Values("standard", "corba", "jini", "upnp")]
typedef short tMiddlewareReference;

[ValueMap("0", "100", "200", "300", "305", "400", "500", "600", "700",
          "706", "712"),
 Values("unknown", "FREEBSD", "IRIX", "LINUX", "LINUX-2.4", "MACOS",
        "NETBSD", "OPENBSD", "WIN32", "WINDOWS-NT", "WINDOWS-NT-5")]
typedef short tOperatingSystem;

[Status("required"),
 ValueMap("0", "1", "2", "3", "4", "5", "6", "7", "8", "9"),
 Values("other", "unknown", "idle", "standby", "active", "busy", "powerUp",
```

```
        "powerDown", "maintenance", "jam")]
typedef short tEntityStatus;

struct sTime{
  [MinValue(0), MaxValue(23), StepIndex(1)]
  unsigned short hour;

  [MinValue(0), MaxValue(59), StepIndex(1)]
  unsigned short minute;

  [MinValue(0), MaxValue(60), StepIndex(1)]
  unsigned short second;

  [MinValue(0), MaxValue(9), StepIndex(1)]
  unsigned short secFrac;

  signed short numOffset;
  signed short offset;
  signed short partialTime;
};

struct sDate{
  [MinValue(1), MaxValue(12), StepIndex(1)]
  unsigned short month;

  [MinValue(1), MaxValue(31), StepIndex(1)]
  unsigned short day;

  [StepIndex(1)]
  signed short fullyear;
};

typedef string tTime;
typedef string tDate;
typedef string tTimeDate;

[ValueMap("0", "1", "2", "3"),
 Values("unspecified", "no-security", "object-authentication",
        "data-encryption")]
typedef short tSecurityLevel;

typedef string tUUID;

[ValueMap("0", "1", "2", "3", "4", "5", "6"),
 Values("unknown", "Information", "Warning", "Error", "Exception",
        "Accounting", "Notification")]
typedef unsigned short ticketCategory;

struct sTicket {
  ticketCategory category;
  tTimeDate time;
  unsigned long ticketPriority;
  string ticketType;
  string ticketOriginator;
  string ticketDescription;
  tNameValueList optionalHeaderFields;
  tNameValueList filterableBody;
  tNameValueList anyThingElse;
};
```

C.1.4 Generic Objects

```
[Description("a general purpose object with an ADL interface to"
             "alter qualifiers that every other object with"
             "alterable qualifiers can inherite from"),
 Abstract, Version(1), Revision(1)]
object oMamaCore{

  [Description("this interface MUST be inherited by all MAMA objects that"
               "are not specified 'abstract'"),
   Version(1), Revision(1)]
  interface iMamaCore{
    [Description("action to alter qualifiers on the object")]
    boolean changeQualifier([In] string name, [In] string value);

    [Description("action to get the complete specification of the object")]
    string getSpecification([Out] MAMA::tSpecLanguage language);
  };
};
```

C.1.5 Entity Management

```
object oEntityMgmt{
  struct sCompileTime{
    unsigned short NumberOfInterfaces;
    MAMA::tTime CompileTime;
    unsigned short Version;
    unsigned short Revision;
    string Cvs;
    MAMA::tEntityType Type;
  };

  struct sInstallation{
    MAMA::tPath PkgLocation;
    MAMA::tDate PkgDate;
    string PkgSerialNumber;
    string PkgProductName;
    string PkgVersion;
    string PkgManufacturer;
  };

  struct sLaunch{
    string launchUser;
    string launchParameters;
    string launchTime;
  };

  struct sRuntimeGeneral{
    string SupportContact;
    string PhysicalLocation;
    string ID;
    unsigned short Boots;
    MAMA::tTime Time;
    MAMA::tTime Uptime;
    MAMA::tTime LocalTime;
    MAMA::tOperatingSystem OS;
    MAMA::tMiddleware Middleware;
    string Host;
    MAMA::tEntityStatus Status;
```

```
};

struct sRuntimeConfigUrls{
  MAMA::tURL ConfigUrl;
  MAMA::tURL PersistentUrl;
  MAMA::tURL LogUrl;
  MAMA::tURL EventServerUrl;
  MAMA::tURL EntityUrl;
};

struct sRuntimeConfigFixed{
  unsigned short LogLevel;
  unsigned short DebugLevel;
  unsigned short MonitoringLevel;
  string SerialNumber;
  string Vendor;
  string Manufacturer;
  string ModelName;
  string LanguageEdition;
};

struct sRuntimeConfigVariable{
  unsigned short TransactionTimeout;
  string SecurityModel;
  MAMA::tSecurityLevel SecurityLevel;
  string OperationStatus;
  MAMA::tTime LastChange;
};

struct sRuntimeLog{
  MAMA::tTime LastRequestIn;
  MAMA::tTime LastRequestOut;
  MAMA::tTime LastResultIn;
  MAMA::tTime LastResultOut;
  MAMA::tTime RejectedRequestsIn;
  MAMA::tTime RejectedRequestsOut;

  [MinLen(0), MaxLen(1000000000), Counter, StepIndex(1)]
  unsigned long RequestInCount;

  [MinLen(0), MaxLen(1000000000), Counter, StepIndex(1)]
  unsigned long RequestOutCount;

  [MinLen(0), MaxLen(1000000000), Counter, StepIndex(1)]
  unsigned long ResultInCount;

  [MinLen(0), MaxLen(1000000000), Counter, StepIndex(1)]
  unsigned long ResultOutCount;

  [MinLen(0), MaxLen(1000000000), Counter, StepIndex(1)]
  unsigned long RejectedRequestsInCount;

  [MinLen(0), MaxLen(1000000000), Counter, StepIndex(1)]
  unsigned long RejectedRequestsOutCount;
};

interface iEntityMgmt{
  [Permissions(0444)]
  attribute MAMA::oEntityMgmt::sCompileTime compileInformation;

  [Permissions(0444)]
  attribute MAMA::oEntityMgmt::sInstallation installInformation;
```

```
      [Permissions(0444)]
      attribute MAMA::oEntityMgmt::sLaunch launchInformation;

      [Permissions(0644)]
      attribute MAMA::oEntityMgmt::sRuntimeGeneral runtimeGeneralInformation;

      [Permissions(0644)]
      attribute MAMA::oEntityMgmt::sRuntimeConfigUrls configUrls;

      [Permissions(0444)]
      attribute MAMA::oEntityMgmt::sRuntimeConfigFixed configInformationFixed;

      [Permissions(0644)]
      attribute MAMA::oEntityMgmt::sRuntimeConfigVariable
                                   configInformationVariable;

      [Permissions(0444)]
      attribute MAMA::oEntityMgmt::sRuntimeLog logInformation;
    };
  };
```

C.2 Application Protocol

```
  [Abstract, Version(1), Revision(1)]
  object swProtocol{
    interface Protocol{
      typedef string tOperation;
      typedef MAMA::tPath[] tSeqObjectPath;

      MAMA::tNameValueList swAction([In] tOperation operation,
                                    [In] tSeqObjectPath addresses,
                                    [In] MAMA::tNameValueList parameters,
                                    [In] MAMA::tNameValueList options);
    };
  };
```

C.2.1 OMG IDL Specification

```
typedef unsigned short IDLDataType;
const IDLDataType IDL_DT_inconsistent = 0;
const IDLDataType IDL_DT_char         = 1;
const IDLDataType IDL_DT_string       = 2;
const IDLDataType IDL_DT_boolean      = 3;
const IDLDataType IDL_DT_octet        = 4;
const IDLDataType IDL_DT_short        = 5;
const IDLDataType IDL_DT_ushort       = 6;
const IDLDataType IDL_DT_long         = 7;
const IDLDataType IDL_DT_ulong        = 8;
const IDLDataType IDL_DT_longlong     = 9;
const IDLDataType IDL_DT_ulonglong    = 10;
const IDLDataType IDL_DT_float        = 11;
const IDLDataType IDL_DT_double       = 12;
const IDLDataType IDL_DT_longdouble   = 13;
const IDLDataType IDL_DT_array        = 18;
const IDLDataType IDL_DT_struct       = 19;

typedef unsigned short IDLDataFlag;
```

```
const IDLDataFlag IDL_DF_none  = 0;
const IDLDataFlag IDL_DF_read  = 1;
const IDLDataFlag IDL_DF_write = 2;
const IDLDataFlag IDL_DF_exec  = 4;

struct IDLNamedValue{
  string       name;
  string       value;
  IDLDataType  vdatatype;
  IDLDataFlag  vdataflag;
};
typedef sequence<IDLNamedValue> IDLSeqNamedValue;

typedef string IDLObjectPath;
typedef sequence<IDLObjectPath> IDLSeqObjectPath;

interface Management{
  IDLSeqNamedValue action(in IDLOperation operation,
                          in IDLSeqObjectPath addresses,
                          in IDLSeqNamedValue parameters,
                          in IDLSeqNamedValue options);
};
```

C.3 Application Programming Interface

```
  [Abstract, Version(1), Revision(0)]
  object swAPI{

    [MinValue(1), MaxValue(9999), StepIndex(1)]
    typedef unsigned long swMaxErrors;

    typedef boolean swTransmiterror;
    typedef string swServerName;
    typedef string swObjectPtr;
    typedef string swOpName;
    typedef string swOpPtr;
    typedef string swOpDescr;
    typedef string swEventDescr;
    typedef string swEventNumber;

    [ValueMap("0", "1", "2", "3", "4", "5"),
     Values("other", "publisher", "publisherAndSubscriber", "subscriber",
            "subscriberAndPop", "SubscriberAndPush")]
    typedef short swEventServerFlags;

    typedef string[] tObjectPath;

    struct swArgStruct{
      swOpName opName;
      MAMA::tNameValueList addressList;
      MAMA::tNameValueList parametersList;
      MAMA::tNameValueList optionsList;
    };

    interface swAPI{
      short initEntity([In] MAMA::tMiddleware mwtype,
                       [In] MAMA::swAPI::swMaxErrors maxerrors,
                       [In] MAMA::swAPI::swTransmiterror transmiterror,
                       [In] MAMA::swAPI::swServerName servername);
```

```
    short configureMiddleware([In] MAMA::swAPI::swObjectPtr ptr);

    short registerEvSrv([In] MAMA::swAPI::swEventServerFlags evflags);

    short deregisterEvSrv();

    short changeRegistrationEvSrv(
            [In] MAMA::swAPI::swEventServerFlags evflags);

    short performAction([In]  MAMA::swAPI::swArgStruct arglist,
                        [Out] MAMA::tNameValueList retarglist);

    short sendEvent([In] MAMA::swAPI::swEventDescr evdescr,
                    [In] MAMA::swAPI::swEventNumber number);

    short addNewOperation([In] MAMA::swAPI::swOpName opname,
                          [In] MAMA::swAPI::swOpPtr opptr,
                          [In] MAMA::swAPI::swOpDescr opdescr);
};
```

C.3.1 Standard Library

```
interface swNamedValue{
  string Name();
  void changeName([In] string name);
  string Value();
};

interface swOptionsList{
  void changeNV([In] MAMA::sNamedValue nv);
  boolean concat([In] MAMA::tNameValueList oplist);
  boolean contains([In] string str);
  MAMA::tNameValueList current();
  unsigned long entries();
  boolean getNext([In] MAMA::sNamedValue nv);
  boolean getNextKey([In] string str);
  boolean insert([In] MAMA::sNamedValue nv);
  boolean next();
  boolean remove([In] string str);
  void reset();
  void showAll();
};

interface swOperationMap{
  boolean contains([In] MAMA::swAPI::swOpName strOP);
  string getNext();
  string getCurrent();
  string getCurrentDescription();
  void reset();
  void insert([In] string key, [In] MAMA::swAPI::swOpName strOp,
              [In] string description);
  unsigned long entries();
  void showAll();
  MAMA::sNamedValue list([In] MAMA::swAPI::swOpName operation);
  MAMA::tNameValueList listAll();
};

interface swObjectPath{
  string decrementPath();
};
```

```
    interface swAddressList{
      unsigned long entries();
      MAMA::swAPI::tObjectPath getFirst();
      MAMA::swAPI::tObjectPath removeFirst();
    };

    interface swError{
      void setServerName([In] string ServerName);
      boolean setTransmitMode([In] boolean transmit);
      void newError([In] MAMA::sNamedValue nv);
      MAMA::sNamedValue lastError();
      MAMA::tNameValueList listErrors();
      void showErrors();
      void showLastError();
      boolean sendLastError([In] boolean transmit);
    };
```

C.3.2 Middleware Library

```
    interface  CORBAServer{
      void shutdown();

      void visibile();
      boolean checkLocalExecution([In] MAMA::swAPI::swArgStruct actionargs);
      boolean checkForwardExecution([In] MAMA::swAPI::swArgStruct actionargs);
      MAMA::tNameValueList ForwardExecution(
                           [In] MAMA::swAPI::swArgStruct actionargs);
    };

    interface CORBA{
      short Initiate([In] MAMA::swAPI::swObjectPtr ptr);
      short Action([In] MAMA::swAPI::swArgStruct ArgStruct,
                   [In] MAMA::tNameValueList ReturnList);
    };
  };
```

C.4 Application Services

C.4.1 Directory Naming and Specification Service

C.4.1.1 eXchange Data Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT collection (directoryEntryInstance*,
                     directoryEntryAlias*,
                     directoryEntry*)>
<!ELEMENT directoryEntry (directoryEntryInstance*,
                          directoryEntryAlias*,
                          directoryEntry*)>
<!ATTLIST directoryEntry
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED>
<!ELEMENT directoryEntryInstance EMPTY>
<!ATTLIST directoryEntryInstance
```

```
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    reference CDATA #REQUIRED
    reference_type CDATA #REQUIRED
    object_distinguished_name CDATA #REQUIRED
    uuid CDATA #IMPLIED
    state CDATA #REQUIRED>
<!ELEMENT directoryEntryAlias EMPTY>
<!ATTLIST directoryEntryAlias
    name CDATA #REQUIRED
    distinguished_name CDATA #IMPLIED
    entry_instance_reference CDATA #REQUIRED
    uuid CDATA #IMPLIED>
```

C.4.1.2 DNSS Specifications

```
  [Version(1), Revision(0)]
  object oDNSS{
    typedef string tDN;
    typedef string tParentDN;
    typedef string tInstanceDN;
    typedef string tObjectDN;

    interface iDNSS{
      /* Operations for both models */
      MAMA::tUUID getUUID();

      string getAttributeValue([In] MAMA::oDNSS::tDN elementDN,
                               [In] MAMA::tElementType entryType,
                               [In] string attributeName);

      boolean setAttributeValue([In] MAMA::oDNSS::tDN entryDN,
                                [In] MAMA::tElementType entryType,
                                [In] string attributeName,
                                [In] string[] attributeValue,
                                [In] MAMA::tUUID uuid);

      /* Directory Model Operations */
      boolean deregister([In] MAMA::oDNSS::tDN instanceDN,
                         [In] MAMA::tElementType entryType,
                         [In] MAMA::tUUID uuid);

      string getAll();

      short getCount([In] MAMA::oDNSS::tParentDN parentDN,
                     [In] MAMA::tElementType entryType,
                     [In] boolean isRecursive);

      string getEntries([In] MAMA::oDNSS::tParentDN parentDN,
                        [In] MAMA::tElementType entryType,
                        [In] boolean isRecursive,
                        [In] short fromIndex, [In] short toIndex);

      string getEntry([In] MAMA::oDNSS::tDN entryDN,
                      [In] MAMA::tElementType entryType);

      string getObjectSpec([In] MAMA::oDNSS::tInstanceDN idname,
                           [In] MAMA::tSpecLanguage formatType);

      string getInstanceIOR([In] MAMA::oDNSS::tInstanceDN idname);
```

```
      boolean modifyEntryName([In] MAMA::oDNSS::tInstanceDN oldEntryDN,
                              [In] MAMA::tElementType entryType,
                              [In] MAMA::oDNSS::tInstanceDN newEntryDN,
                              [In] MAMA::tUUID uuid);

      boolean register([In] MAMA::oDNSS::tDN entryDN, [In] string ior,
                       [In] MAMA::tMiddlewareReference referenceKind,
                       [In] MAMA::oDNSS::tObjectDN odname,
                       [In] MAMA::tUUID uuid);

      boolean registerAlias([In] MAMA::oDNSS::tDN aliasDN,
                            [In] MAMA::oDNSS::tDN instanceDN,
                            [In] MAMA::tUUID uuid);

      /* Specification Model Operations */
      boolean addSpecification([In] string specificationName,
                               [In] string specification,
                               [In] MAMA::tUUID uuid);

      boolean addToSpecification([In] string newElement,
                                 [In] MAMA::tElementType elementType,
                                 [In] MAMA::oDNSS::tParentDN parentDN,
                                 [In] MAMA::tUUID uuid);

      string getElement([In] MAMA::oDNSS::tDN elementDN,
                        [In] MAMA::tElementType elementType,
                        [In] MAMA::tSpecLanguage formatType,
                        [In] boolean compact);

      string getElements([In] MAMA::oDNSS::tParentDN parentDN,
                         [In] MAMA::tElementType elementType,
                         [In] boolean isRecursive, [In] short fromIndex,
                         [In] short toIndex,
                         [In] MAMA::tSpecLanguage formatType,
                         [In] boolean compact);

      string getElementsByValue([In] string attributeName,
                                [In] string attributeValue,
                                [In] MAMA::tElementType elementType,
                                [In] short fromIndex, [In] short toIndex,
                                [In] MAMA::tSpecLanguage formatType,
                                [In] boolean compact);

      string getInstances([In] MAMA::oDNSS::tObjectDN objectDN);

      boolean remove([In] MAMA::oDNSS::tDN elementDN,
                     [In] MAMA::tElementType elementType,
                     [In] MAMA::tUUID uuid);
    };
  };
```

C.4.2 XAMAV CSS Specification

```
li.f.object, li.f.module, li.f.interface { color:red; font:18pt Helvetica;}
.f.object.name, .f.module.name, .f.interface.name, { color:black; font:18pt
Helvetica;}
.f.object.ext { color:red; font:12pt Helvetica;}
.f.object.ext.name { color:black; font:12pt Helvetica;}
.f.object.qual, .f.interface.qual, .f.module.qual { color:black; font:10pt
```

```
Helvetica;}
.f.object.qual.name, .f.interface.qual.name, .f.module.qual.name {
color:#990000; font:bold 12pt Helvetica;}
.f.object.cv, .f.module.cv, .f.interface.cv, { color:blue; font:italic 10pt
Helvetica;}
.f.object.cv.text, .f.module.cv.text, .f.interface.cv.text { color:black;
font:11pt Helvetica;}
li.f.typedef, li.f.attribute, li.f.action { col or:blue; font:18pt
Helvetica;}

/* typedef styles (filter)*/
.f.typedef { color:blue; font:italic 10pt Helvetica;}
.f.typedef.name { color:black; font:11pt Helvetica;}
.f.typedef.basetype, .f.typedef.type, .f.typedef.signed,
.f.typedef.array_dim { color:black; font:11pt Helvetica;}
.f.typedef.basetype.name, .f.typedef.type.name, .f.typedef.signed.name,
.f.typedef.array_dim.name { color:black; font:11pt Helvetica;}
.f.typedef.qual { color:black; font:10pt Helvetica;}
.f.typedef.qual.name { color:#990000; font:bold 12pt Helvetica;}
.f.typedef.cv { color:blue; font:italic 10pt Helvetica;}
.f.typedef.cv.text { color:black; font:11pt Helvetica;}

/* member styles (filter) */
.f.member { color:blue; font:italic 10pt Helvetica;}
.f.member.name { color:black; font:11pt Helvetica;}
.f.member.basetype, .f.member.type, .f.member.signed, .f.member.array_dim {
color:black; font:11pt Helvetica;}
.f.member.basetype.name, .f.member.type.name, .f.member.signed.name,
.f.member.array_dim.name { color:black; font:11pt Helv etica;}

/* parameter styles (filter) */
.f.parameter { color:blue; font:italic 10pt Helvetica;}
.f.parameter.name { color:black; font:11pt Helvetica;}
.f.parameter.basetype, .f.parameter.type, .f.parameter.signed,
.f.parameter.array_dim { color:black; f ont:11pt Helvetica;}
.f.parameter.basetype.name, .f.parameter.type.name,
.f.parameter.signed.name, .f.parameter.array_dim.name { color:black;
font:11pt Helvetica;}

/* action styles (filter) */
.f.action { color:blue; font:italic 10pt Helvetica;}
.f.action.name { color:black; font:11pt Helvetica;}
.f.action.basetype, .f.action.type, .f.action.signed, .f.action.array_dim {
color:black; font:11pt Helvetica;}
.f.action.basetype.name, .f.action.type.name, .f.action.signed.name,
.f.action.array_dim.name { color:black; font:11pt Helvetica;}
.f.action.qual { color:black; font:10pt Helvetica;}
.f.action.qual.name { color:#990000; font:bold 12pt Helvetica;}
.f.action.cv { color:blue; font:italic 10pt Helvetica;}
.f.action.cv.text { color:black; font:11pt Helvetica ;}

/* attribute styles (filter) */
.f.attribute { color:blue; font:italic 10pt Helvetica;}
.f.attribute.name { color:black; font:11pt Helvetica;}
.f.attribute.basetype, .f.attribute.type, .f.attribute.signed,
.f.attribute.array_dim { color:black; font:1 1pt Helvetica;}
.f.attribute.basetype.name, .f.attribute.type.name,
.f.attribute.signed.name, .f.attribute.array_dim.name { color:black;
font:11pt Helvetica;}
.f.attribute.qual { color:black; font:10pt Helvetica;}
.f.attribute.qual.name { color:#990000; f ont:bold 12pt Helvetica;}
.f.attribute.cv { color:blue; font:italic 10pt Helvetica;}
.f.attribute.cv.text { color:black; font:11pt Helvetica;}
```

```
/* general styles (core model)*/
li.cm.object, li.cm.module, li.cm.interface { color:green; font:18pt
Helvetica;}
li.cm.typedef, li.cm.attribute, li.cm.action, li.cm.qualifierdef {
color:green; font:18pt Helvetica;}
.cm.object.name, .cm.module.name, .cm.interface.name, { color:black;
font:18pt Helvetica;}
.cm.object.ext { color:green; font:12pt Helvetica;}
.cm.object.ext.name { color:black; font:12pt Helvetica;}
.cm.qual { color:black; font:10pt Helvetica;}
.cm.qual.name { color:green; font:bold 12pt Helvetica;}

/* qualifierdef styles (core model) */
.cm.qualifierdef { color:blue; font:italic 10pt Helvetica;}
.cm.qualifierdef.name { color:black; font:11pt Helvetica;}
.cm.qualifierdef.basetype, .cm.qualifierdef.signed,
.cm.qualifierdef.array_dim { color:green; font:11pt Helvetica;}
.cm.qualifierdef.basetype.name, .cm.qualifierdef.signed.name,
.cm.qualifierdef.array_dim.name { color:black; font:11pt Helvetica;}
.cm.qualifierdef.default_value, .cm.qualifierdef.alterable { color:green;
font:11pt Helvetica;}
.cm.qualifierdef.default_value.name, .cm.qualifierdef.alterable.name {
color:black; font:11pt Helvetica;}

/* typedef styles (core model)*/
.cm.typedef { color:green; font:italic 10pt Helvetica;}
.cm.typedef.name { color:black; font:11pt Helvetica;}
.cm.typedef.basetype, .cm.typedef.type, .cm.typedef.signed,
.cm.typedef.array_dim { color:green; font:11pt Helvetica; }
.cm.typedef.basetype.name, .cm.typedef.type.name, .cm.typedef.signed.name,
.cm.typedef.array_dim.name { color:black; font:11pt Helvetica;}

/* parameter styles (core model) */
.cm.parameter { color:green; font:italic 10pt Helvetica;}
.cm.parameter.name { color:black; font:11pt Helvetica;}
.cm.parameter.basetype, .cm.parameter.type, .cm.parameter.signed,
.cm.parameter.array_dim { color:green; font:11pt Helvetica;}
.cm.parameter.basetype.name, .cm.parameter.type.name,
.cm.parameter.signed.name, .cm.paramete r.array_dim.name { color:black;
font:11pt Helvetica;}

/* member styles (core model) */
.cm.member { color:green; font:italic 10pt Helvetica;}
.cm.member.name { color:black; font:11pt Helvetica;}
.cm.member.basetype, .cm.member.type, .cm.member.signed,
.cm.member.array_dim { color:green; font:11pt Helvetica;}
.cm.member.basetype.name, .cm.member.type.name, .cm.member.signed.name,
.cm.member.array_dim.name { color:black; font:11pt Helvetica;}

/* scope styles (core model) */
.cm.scope { color:green; font:italic 10pt Helvetica;}
.cm.scope.name { color:black; font:11pt Helvetica;}
.cm.scope.element { color:green; font:11pt Helvetica;}
.cm.scope.rank { color:black; font:11pt Helvetica;}

/* description styles (core model) */
.cm.description { color:green; font: 14pt Helvetica;}
.cm.description.text { color:black; font:11pt Helvetica;}
```

C.4.3 Notification Event and Log Service

```
object oNELS{
  interface iNELS{
    struct nelsSubscription{
      [ValueMap("1", "2", "3", "4"),
       Values("Consumer", "Producer", "Subscriber", "Publisher")]
      short role;

      [ValueMap("1", "2"),
       Values("push", "pop")]
      short method;

      string channel;
      string objectClass;
      string objectInstance;
      MAMA::ticketCategory category;
    };

    boolean subscribe([In] nelsSubscription subscription,
                      [In] MAMA::oDNSS::tInstanceDN name);

    boolean unSubscribe([In] nelsSubscription subscription,
                        [In] MAMA::oDNSS::tInstanceDN name);

    string[] getChannels();
    boolean submitTicket([In] MAMA::sTicket ticket);
    void showTicket();
  };
};
```

C.4.4 Lifecycle and Configuration Management Service

```
module mLCMS{
  struct sInterface{
    MAMA::oDNSS::tInstanceDN instanceDN;
    string intReference;
    MAMA::tMiddlewareReference intReferenceType;
    MAMA::oDNSS::tDN interfaceSpec;
  };
  typedef sInterface[] interfaceList;

  struct sObject{
    MAMA::oDNSS::tInstanceDN objectDN;
    MAMA::oDNSS::tDN objectSpec;
    sInterface[] interfaces;
  };
  typedef sObject[] objectList;
```

C.4.4.1 Object Interface

```
  object oMamaObjectInit:MAMA::oMamaCore{
    interface iMamaObjectInit{
      boolean create();
      void init([In] MAMA::tNameValueList initParams);
      void terminate();
      void checkpoint();
    };
```

```
    };
```

C.4.4.2 Cluster Manager

```
    object oClusterManager{
      interface iClusterManagement{
        [Permissions("0444")]
        attribute MAMA::mLCMS::objectList objects;

        MAMA::oDNSS::tInstanceDN create(
            [In] MAMA::oDNSS::tInstanceDN instanceDN);

        void init([In] MAMA::oDNSS::tInstanceDN instanceDN,
                  [In] MAMA::tNameValueList initParams);

        void terminate([In] MAMA::oDNSS::tInstanceDN instanceDN);

        MAMA::mLCMS::interfaceList getIntRefs(
            [In] MAMA::oDNSS::tInstanceDN instanceDN);

        MAMA::mLCMS::interfaceList getAllIntRefs();

        MAMA::oDNSS::tDN selectIntRef(
            [In] MAMA::oDNSS::tInstanceDN instanceDN,
            [In] MAMA::oDNSS::tDN interfaceType);
      };
    };
```

C.4.4.3 Capsule Manager

```
    object oCapsuleManager{
      interface iCapsuleManagement{
        boolean setType([In] MAMA::oDNSS::tDN elementDN);
        boolean delType([In] MAMA::oDNSS::tDN elementDN);
        MAMA::oDNSS::tDN getTypes();

        MAMA::mLCMS::sObject create([In] MAMA::oDNSS::tDN co_type,
                                    [In] MAMA::oDNSS::tInstanceDN instanceDN);

        void init([In] MAMA::oDNSS::tDN co_type,
                  [In] MAMA::oDNSS::tInstanceDN instanceDN,
                  [In] MAMA::tNameValueList initParams);

        void terminate([In] MAMA::oDNSS::tInstanceDN instanceDN);

        MAMA::mLCMS::interfaceList getIntRefs(
            [In] MAMA::oDNSS::tDN coType,
            [In] MAMA::oDNSS::tInstanceDN instanceDN);

        MAMA::mLCMS::objectList getAllIntRefs(
            [In] MAMA::oDNSS::tDN coType,
            [In] MAMA::oDNSS::tInstanceDN instanceDN);

        MAMA::mLCMS::sInterface selectIntRef(
            [In] MAMA::oDNSS::tDN co_type,
            [In] MAMA::oDNSS::tInstanceDN instanceDN,
            [In] MAMA::oDNSS::tDN interfaceType);
      };
    };
  }; // LCMS
```