A Viable Architecture for Autonomic Management of Distributed Software Components

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Emil Stoyanov

aus Ruse, Bulgarien

Hauptberichter:	Prof. Dr. Dieter Roller
Mitberichter:	Prof. Dr. rer. nat. habil. Paul Levi
Tag der mündlichen Prüfung:	27.04.2010

Institut für Rechnergestützte Ingenieursysteme
der Universität Stuttgart

2010

Acknowledgement

This dissertation was developed as a cooperation between the Institut für Rechnergestützte Ingenieursysteme (IRIS) at Universität Stuttgart and Siemens Corporate Technology.

I would like to express my gratitude to everyone who influenced me, helped me and supported me during the years of work dedicated to this dissertation.

Thank you, Professor Dr. Dieter Roller, for giving me the chance to experience the real research spirit at IRIS and to work in a team of real professionals, for the provided support, the understanding, your valuable advices and feedback. Thank you, all IRIS colleagues for the kind attitude, support and help. A special "thank you" to Stavros and his whole family for the unconditional help, guidance and personal attitude.

Thank you, all colleagues from Siemens Corporate Technology. Markus, thank you for being the one to support me and to introduce me to this great working environment, for the comments, ideas, advices and for your professional spirit. Thank you, Asa, for the feedback, for sharing your wisdom and for showing me attitude, in both professional and personal aspects. Thank you Andreas, Marcos, Ivo, Anton, Charek, Xun and Matthias for being the best office-mates and for your support during the innevitable difficult moments.

Thank you Velislav and Alexander, for being great friends, always available and ready to share valuable thoughts and feedback.

I would like to thank my parents for their faith and moral support. Special gratitude to my wife, Miglena, for her love, help and endless patience.

Contents

List of Abbreviations

ABLE - Agent-building and Learning Environment, 57
AI - Artificial Intelligence, 23, 71
API - Application Programming Interface, 67, 89, 90, 116, 118, 120, 121, 130, 135, 149, 167

CAD - Computer-Aided Design, 26
CDL - Component Description Language, 60, 70
CDS - Channel Dynamics Status, 129
CIM - Common Information Model, 66, 88, 89, 140–149, 171
CIM-OM - CIM Operation Manager, 88, 89, 159
CLR - Common Language Runtime, 39, 51, 65
COM - Component Object Model, 65, 67, 108, 121
CORBA - Common Request Object Architecture, 60, 69, 70, 88–90, 139–142, 145, 146, 149, 158, 171

DBMS - Database Mangement System, 31, 67
DMTF - Distributed Management Task Force, 40, 66, 171
DPD - Distributed Product Development, 26, 151, 155

EJB - Enterprise Java Beans, 25, 26, 49, 65, 70, 76, 82, 83, 85, 87–89, 107, 120, 121, 138, 142, 150–152, 154, 155, 157, 158

IT - Information Technology, 23–25, 30, 33, 41, 54, 58, 88, 89, 91, 100, 101, 141, 162

J2EE - Java 2 Enterprise Edition, 39, 49, 50, 65, 67, 68, 70, 76, 120, 145
JCP - Java Community Process, 51
JMX - Java Management Extensions, 39, 49, 65, 66, 82, 85, 89, 90, 121, 138, 145, 158, 163, 167
JNDI - Java Naming and Directory Interface, 65
JVM - Java Virtual Machine, 39, 74, 76

LRK - Law of Requisite Knowledge, 46, 126, 129, 132
LRV - Law of Requisite Variety, 46, 47, 123, 124, 126

MAS - Multi-Agent System, 85, 87, 154, 155
MCC - Managed Communication Channel, 74, 76, 77, 97, 99, 156

RCP - Rich Client Platform, 26, 85, 87, 154, 155
RMI - Remote Method Invocation, 65, 173
RPC - Remote Procedure Call, 52, 68–70, 83, 141, 154
RPD - Rapid Product Development, 35, 38, 85, 154, 155

SLEE - Service Logic Execution Environment, 39, 121, 138, 145
SMC - Self-managed Cell, 55, 56

UPnP - Universal Plug and Play, 35, 172

VSM - Viable System Model, 19, 46–48, 77, 89, 90, 92–95, 97, 100–102, 121–123, 127–130, 132–134, 136,

List of Figures

List of Tables

Abstract

Autonomic Computing is a brand of system design approaches which enable IT systems with self-management capabilities such as self-configuration, self-healing, self-protection and self-optimization. Although the field of distributed system management has achieved considerable advances, building autonomic management solutions for heterogeneous component-based systems presents five major challenges. First, component deployment and its management gets difficult with the growth of the system, because of the variety of component models with their own specifics. Second, each component framework provides its own way and interface for management creating redundancy and variety of management routines. Third, software components evolve separately which introduces problems with compatibility upon system upgrade. Forth, there are remote dependencies which are difficult to tack and this may cause unpredicted inconsistency of the system after component update. Finally, the integration of a management sub-system influences the overall system complexity by making it dependent on interfaces and functionality of the management module.

This thesis introduces an architectural approach which addresses these challenges. An organizational meta-model represents the architectural constraints for encapsulation of software components and defines requirements for feedback loops adapted from the Viable System Model (VSM) for software components. It enables modeling of viable organization and communication management on the levels of component deployment and runtime operations.

The autonomic management architecture consists of modules that facilitate monitoring component states, an operation manager that allows inspection of distributed dependencies by utilizing the notion of the managed communication channel. Its design conforms with the recommendations of the proposed organizational model. A channel management middleware implements the necessary functionality for establishing communication channels and provides interfaces for integration of autonomic managers which follow the requirements of the organizational and communication model.

A prototype of the middleware has been developed to implement the architectural approach for real-world scenarios in two separate domains - Smart Home and Distributed Product Development Support Systems. It has demonstrated the usability of the architecture by satisfying the management requirements of these domains and addressing the management challenges.

Zusammenfassung

Autonomic Computing ist ein spezieller Ansatz des System-Designs, der die Entwicklung von IT-Systemen mit Selbst-Management-Fähigkeiten ermöglicht, die Eigenschaften wie Selbst- Konfiguration, Selbst-Heilung, Selbst-Schutz und Selbst-Optimierung integrieren. Obwohl der Bereich der verteilten System-Verwaltung erhebliche Fortschritte erzielt hat, sind die Lösungen für heterogene Komponenten-basierte Management-Systeme von fünf großen Herausforderungen gekennzeichnet. Zunächst wird der Komponenten-Einsatz und die Verwaltung schwierig mit dem Wachstum des Systems; ein Grund dafür ist die Vielzahl von Komponenten- Modellen, die ihre eigenen Besonderheiten haben. Ein zweiter Punkt ist, dass jedes Komponenten- basierte Framework über seine eigenen Abläufe und seine eigenen Schnittstellen verfügt, was zu zusätzlicher Redundanz und Vielfalt von Management-Routinen führt. Drittens werden Software-Komponenten weiterentwickelt, und erzeugen so Probleme mit der Kompatibilität beim System-Update. Besonders verteilte Abhängigkeiten, die schwer zu beheben sind können nach einem System-Update zu unvorhersehbaren Inkonsistenzen führen. Schließlich beeinflusst die Integration eines Management-Sub-Systems die Gesamt-Komplexität des Systems, so dass eine starke Abhängigkeit von den Schnittstellen und von den Funktionen des Management-Moduls erzeugt wird.

Diese Dissertation stellt einen Architektur-Ansatz vor, um diesen Herausforderungen zu begegnen. Ein organisatorisches Meta-Modell stellt die Architektur -Anforderungen für die Kapselung von Software-Komponenten bereit und definiert angepasste Feedback-Schleifen, die das Viable Systems Model für Software-Komponenten realisieren. Es ermöglicht die Modellierung von komplexen Strukturen und Kommunikationswegen auf der Ebene der Komponenten-Implementierung und der Runtime-Operationen.

Die Management Architektur besteht aus Modulen, die die Analyse von Komponenten erleichtern und aus einem Operation-Manager, der verteilte Abhängigkeiten durch den Einsatz der verwalteten Kommunikationskanäle (Managed Communcation Channels) kontrolliert. Eine Channel-Management-Middleware implementiert die notwendige Funktionalität für die Einrichtung von Kommunikationskanälen und bietet Schnittstellen für die Integration von Autonomen Managern, die die Anforderungen des Organisations- und des Kommunikations-Modells erfüllen.

Ein Prototyp der Middleware wurde entwickelt, um das Konzept für die Real-World-Szenarien in zwei getrennten Bereichen zu demonstrieren - Smart Home und verteilte Produkt-Entwicklung. Hierdurch wurde die Nutzbarkeit der entwickelten Architektur demonstriert, wobei die speziellen Herausforderungen und Management-Anforderungen dieser Bereiche erfüllt werden.

Chapter 1

Introduction

Autonomic Computing is a brand of system design approaches that enable Information Technology (IT) systems with self-management capabilities such as self-configuration, self-healing, self-protection and self-optimization. During the last several years the need of automated management and system self-diagnosis has grown steadily as a consequence of the increasing system complexity, heterogeneity and degree of system distribution. Although the field of distributed system management has achieved considerable advances, autonomic computing raises the requirements to a higher level of security and return of investments. This aim however, introduces new challenges, part of which this dissertation addresses.

This dissertation proposes an architectural approach, the cybernetically viable architecture for heterogeneous component-based software management, which supports integration of self-management capabilities in distributed software component systems focusing on the stability and adaptability throughout their evolution in production environments.

This dissertation describes the field of autonomic computing and distributed component management together with the existing challenges demanding attention. It shows how the cybernetically viable autonomic architecture addresses these challenges by usage of cybernetic principles and extending already developed architectural approaches in the field. To achieve this it presents a meta-model, architecture and a middle-ware using real-world examples and deployment scenarios. The thesis concludes with results from experimental work using a developed prototype.

1.1 Autonomic Computing

Autonomic Computing [47, 53] is a trend in the development of system management approaches which focuses on enabling systems with self-management capabilities. It combines elements from the fields of distributed management, adaptive systems, artificial intelligence (AI) and reliable computing. It emerged as a reaction to the

Term Evolution	1982	2003+
System Service	Local standalone application on a single host	Composite services running on a network of serving hosts
System Components	Microprocessors, Disk drives, TCP-IP Stack, Operating System	Servers, Network accessed storage arrays, Networks, Network control platform
System Management	Management of discrete component configuration	Management of clustered and networked resources

Table 1.1: The meaning of system management changes with the development of IT.

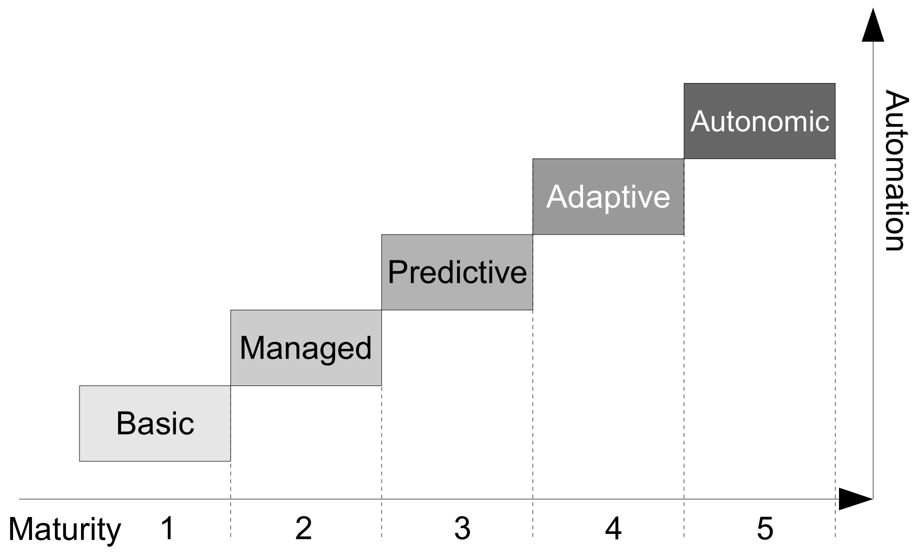


Figure 1.1: Levels of Management Maturity

growing complexity of the IT systems, the increasing human-error factor and the higher requirements for security and system availability.

The variety of standards, the dynamically changing requirements and distribution for system functionality resulted in a mix of technological solutions that hardly communicate and even worse, became hardly manageable. A single organization deploys in its infrastructure a number of different solutions that are either not designed to be managed or had limited and proprietary interfaces for management. The developments of Internet and the recognition of IT value inside organizations of any size has additionally boosted the growth and the transformation of locally run systems with local programs to distributed clusters of serving equipment with network services processing enormous amount of data.

Table 1.1 [140] shows how the meaning of the word “system” has changed over the time along with the development of IT.

The IT research community recognized the need of self-management capabilities with dedicated international conferences and workshops that specifically investigate issues like architectures and algorithms for self-configuration, self-protection, self-optimization and self-healing. Figure 1.1 illustrates the levels of management maturity [46] in which autonomic management takes highest place. Most of the systems today still use systems that belong to level 1 (basic) and level 2 (managed) while a very small percentage utilizes management systems from level 3 (predictive). The autonomic computing initiative aims to deliver the needed tools and the necessary bridge for successful management support on the autonomic level.

In a typical autonomic environment the system administrator defines a policy for automated self-management. The system tracks the values of its own parameters like health condition, system load, security intrusion symptoms, component errors and executes the defined policy which may activate actions demanding human assistance or trigger completely automated chain of procedures acting on the performance, security, configuration or its overall "health" condition reflecting the stability and reliability of the system.

The automated sequence of monitoring, analysis, planning and execution is known as *autonomic control loop* [5]. For a dynamic and adaptive self-management a single control loop for a whole system is not sufficient because of the distributed nature of the modern systems. That is why an autonomic management environment has to foresee and support the management *interoperability* and *cooperation* in distributed parts of the system. Autonomic management consolidates different approaches of monitoring and reaction. For example, *passive monitoring* activates the control loop upon a generated event. In the case of *pro-active monitoring* [134], the system parameters are inspected *autonomously* and independently from the rest of the system. Both passive and pro-active monitoring contribute to discovery and prediction of problems before it is too late and critical for the operational health of the system. Examples for autonomic self-management are web-server load-balancing, dynamic database and storage configuration [36], network routing optimization, automated restart (self-healing) in case of component failure, reliable and redundant component communication and many others wherever an autonomic loop controls important parameter values and system behavior.

In a typical industrial IT deployment setup multiple versions of different technologies have to be integrated for a system functional completeness. One reason for this is the need for communication across the boundaries of a particular organization that adopts a technology solution [117]. This technological heterogeneity is accompanied by the problem of evolution in the technology itself resulting in conflicting components or system wide communication problems. The distributed component technologies [69] that are currently being used in production environments do not have intrinsic methods of handling such kind of problems. These include well-established software frameworks such as Enterprise Java Beans (EJB), OSGi Framework, Windows components, assemblies and Web Services. An environment that integrates these technologies exhibits difficulties in the general management of the system as a whole and specifically problem tracking, dependency in communication and version management.

An example for deployment of component-based systems include service provision for *smart home* [116] where a component framework, for example OSGi, is deployed on a home gateway that communicates with a service provider via web service interfaces. The service provider on his own has a separate component based service provision back-end based, for example, on EJB. Although these are two separate systems, there are critical dependencies between the home gateway and the service provision back-end.

Another example for heterogeneous environment is a *distributed product development* (DPD) *system* [118, 123]. Such a system typically deploys a variety of hardware devices and software for both management and development purposes. The adoption of new functionality is important for the efficiency, for example in prototyping and testing phases, but at the same time software stability during project development is a critical requirement for successful project completion. The distributed nature of the technical documentation bases, for example using EJB, and the Computer-Aided Design (CAD) environments, based on custom plug-in architectures, or *Rich Client Platforms* (RCP) [25, 28] have functional dependencies. Careful maintenance has to bring the system to a consistent state and assure healthy component communication.

These common difficulties can be addressed by an autonomic architecture that facilitates self-inspection, heterogeneity management and communication. It would contribute to a more reliable and cost-efficient system control. However, the development of such system exhibits a number of architectural challenges and requirements that have to be addressed.

1.2 Challenges in Autonomic Management of Software Components

Development of autonomic systems has seen a significant progress during the last several years [1, 20, 85, 95, 99] despite the difficulties it faces in the field of automated management, reasoning, control design and interfaces. Previously manual management of system elements has been partially replaced with the help of autonomic design. Among the notable advances are self-configuration for databases, auto-discovery of network devices, self-optimization of grid resources. However few of the projects address some of the significant obstacles for autonomic management in distributed environment. In the field of component based management nothing meaningful is yet developed to face these issues. Currently there is no architectural approach that enables component management in heterogeneous environment with a *common model, common architecture* and using a *common middle-ware.* This thesis addresses five challenges that are related to the development and integration of autonomic support for such systems.

First, different frameworks define their component models with specific life-cycles, access methods, levels of encapsulation and meta-data descriptions. In a distributed environment deployment and continuous human support of several different component frameworks is an error-prone process that puts the system stability in risk. This risk increases with the growth of the system as more components are being deployed. This creates the problem of *overhead in component deployment management.*

Second, component frameworks usually provide management interfaces for monitoring of and access to resources but the management standards [73] and interfaces that framework vendors develop are different. This introduces *management heterogeneity* inside a single business domain, thus additional management overhead and higher risk of error.

Third, depending on the degree of system distribution and the methods of development, components that are loosely coupled may evolve with different paces. The coordination of their integration is conditioned to be difficult and the risk of incompatibility or lack of communication interoperability is increasing with time [117]. This creates the problem of *parallel component evolution.*

Forth, software components may have local dependencies within a single container, including dependencies on local framework services, but in distributed environment there are remote dependencies that may be mutually important for the antecedent and depending components, including dependencies on remote services [90]. This creates the problem of *deep component dependencies.*

Fifth, the integration of management facilities such as autonomic managers, policy engines, sensors and pro-active monitoring managers requires adaptation of the existing infrastructure, communication interfaces or component logic. This creates the problem of *management complexity overhead.*

1.3 Proposed Solution

To address the five problems of overhead in component deployment management, management granularity, parallel component evolution, deep component dependencies and management complexity overhead the use of an *organizational software management model* for component description is proposed; a *cybernetically viable architecture for autonomic component management* and *supporting middle-ware for dynamic communication channel management.*

The model defines the organizational layout and information flow that suits the description of component encapsulation to address the problems in deployment and multi-version component communication. The architecture implements a traditional autonomic loop adapted to both fit and to integrate the organizational model without management complexity overhead, but capable of handling management interface adaptation. The middle-ware facilitates dynamic selection of communication interfaces to address inter- and intra-component dependencies and allow communication brokering between parallel evolving components.

1.4 Research Hypothesis

The research hypothesis is as follows:

> The cybernetically viable software architecture, coordinated by a channel management middle-ware and integrated organizational system model for software, facilitates integration of autonomic management capabilities by addressing the problems of overhead in component deployment management, management granularity, parallel component evolution, deep component dependencies and management complexity overhead. The architecture can be used with a number of popular software frameworks for distributed computing to increase system reliability by supporting consistency in component communication between evolving components.

1.5 Approach

To express the validity of this thesis an overview of the currently applied approaches in this field is given and a new approach is proposed based on the reviewed methods with extensions and modifications which target problems and requirements that were previously not met. Then, the feasibility of the approach is explored in concrete scenarios for distributed component deployment and its usefulness evaluated with the help of a developed prototype.

Chapter 2 overviews existing techniques for development of autonomic management systems and particularly management of distributed component systems. It discusses several example applications of those approaches and defines existing problems, requirements and challenges.

Chapter 3 reviews existing research projects that developed architectures for integration and implementation of autonomic management in software systems. It then illustrates common development approaches that are helpful in addressing the challenges.

Chapter 4 presents the solution and shows its application in scenarios from two different domains - Smart Home and Distributed Product Development.

Chapter 5 and Chapter 6 look into the details of the model, the architecture and the middle-ware of the solution.

Chapter 7 presents results and impressions from a built prototype that implements the solution.

Chapter 8 concludes the thesis with a summary of the contribution, presents the limitations of the approach and shows the directions for future developments.

1.6 Contribution

This dissertation contributes to the development and research of autonomic computing architectures with a model of component systems organization, an architecture for integration of autonomic management capabilities and a middle-ware for communication-based coordination of evolving software components. The system design approach proposed in this thesis can be used as an example for integration and organization of management facilities in other systems where components exhibit hierarchical and recursive type of containment.

Chapter 2

Autonomic Systems Engineering

Design of autonomic systems requires specific knowledge about the nature of the systems for which self-management capabilities are provided. The specific architecture which an IT system implements influences how the management sub-system receives events and management information or how it affects the system state. However, there are basic principles of organization, automation and control that autonomic systems exhibit. This chapter presents general methods for building autonomic systems abstracted from details about concrete implementations of interfaces for monitoring and management.

2.1 Definition and vision

Autonomic is often misunderstood or misinterpreted as "automatic or "autonomous". Although autonomic systems might exhibit characteristic of automation and autonomous behavior the meaning of autonomic clearly defines the context of *self-management.*

Webster's Revised Unabridged Dictionary defines the meaning of autonomic as follows:

> *Autonomic \Au'to*nom"ic\, a.*
> *Having the power of self-government; autonomous*

Wordnet Dictionary defines autonomic as follows:

> *autonomic adj : relating to or controlled by the autonomic nervous system; "autonomic reflexes"*

The main aim of the autonomic computing as a research field is to bring know-how for building IT systems that allow people to focus on the big picture and let the low level of the system to monitor and manage itself. In a typical corporate network, there is a wide range of clients, servers and multiple databases connected via network crossing internal and external firewalls. To manage this network one has to consider dozens of systems and applications, hundreds of components, and thousands of tuning parameters. Autonomic computing aims for engineering of such complex systems that can accept business process-related high-level rules as input and manage their own infrastructure, configuration and functionality.

2.2 Examples for Application of Autonomic Solutions

This section provides an overview of scenarios that apply to the domain of autonomic management. The examples 2.2.1 - 2.2.6 demonstrate concrete cases of need for autonomic support in order to improve the reliability, availability or security of system services. Sections 2.2.7 - 2.2.9 show how these standalone scenarios are used in a real-world setup of on-line service provision where the elements are dependent on each other in networked environment.

2.2.1 Dynamic Database Tuning

Database Management Systems (DBMS) are notorious for the large set of parameters [110] that govern the behavior and performance and the optimal performance configuration of the system is usually dependent on the specific scenario of database access and data organization. An autonomic management system contributes to the optimization and system health management by an automated configuration of the database in order to save time and decrease the risk of human error. The parameters that an autonomic manager monitors are adjusted according to the dependencies on concrete setup specifics and scenario of utilization, for example, present system resources and expected database load. In the case of load management the database needs to be dynamically tuned by adjusting parameters, such as cache sizes, type of indexing, process forking strategy and others. Additionally an autonomic manager can monitor irregularities and oddities, such as corrupted database indexes that can be restored automatically by the system. The reaction of the manager and its ability to detect problems on-time is critical for the functional health of the system.

2.2.2 Dynamic Web Server Tuning

Web server tuning is usually a static procedure of common configuration that suites most cases of web server utilization [36]. However a dynamically tuned server utilizes system resources in a better way and increases system responsiveness. An example is the Apache Web server. The desired and feasible CPU and memory utilization can be achieved by properly selecting the tuning parameters "MaxClients" and "KeepAlive". A higher value of the "MaxClients" parameter allows the Apache server to process more client requests, and increases both CPU and memory utilization. Decreasing the value of "KeepAlive" parameter potentially allows worker processes to be more active, which directly results in higher CPU utilization and indirectly increases memory utilization, since more clients can connect to the server. In practice, adjusting these parameters is time-consuming, error-prone, and skills-intensive. For an automated approach the meaning of a "better system resource utilization" can be defined by means of management policy that is executed by an autonomic manager according to the load of the server. The manager monitors the values of the load metrics and adjusts the "MaxClients" and "KeepAlive" parameter values.

2.2.3 Component Self-healing

The component-based systems often exhibit problems with unpredicted component behavior. Partially this is due to loosely coupling in both runtime and development phases. The logic behind the implementation of a component is not always prepared for the input from other components in the system and in the case of untested or fault design of the component's variable states, different events may result in a complete termination of the process (often called "crash"), or a continuous loop (often described as "hanging"). In such situations the fastest and practically effective measurement is to set the component in initial state, or with other words to restart it. In autonomic computing terms this is the primitive form of component *healing*. In a distributed system this kind of system repairing is error-prone and time-consuming procedure [80]. An autonomic management system can address this problem by using monitoring of type "watchdog" [52]. The system monitors the state of the component by expecting *service pulse* within a set period of time-out. If the component does not report its status on time, the manager does the necessary to bring the component in normal operation. This approach of self-healing reduces the time to reaction to bring the system in a healthy state and contributes to problem reports, such as crash conditions and faulty component localization.

2.2.4 Dynamic network routing

The general meaning of network implies connected and communicating nodes. Whether it is a network of servers, network of clients and servers or just supporting network of routing devices, the concrete path of information between nodes is known as a route and the process of message delivery through selected set of nodes is called routing. The way routing is done has impact on the performance and the operation of the network as medium. For the different architectures and system communication flow different routes are used to fully utilize system resources. The topic of dynamic routing in communication networks has been researched in detail [85] and there are existing approaches to auto-configuration of the routing algorithms, but the general notion of routing optimization with the accent of general management and dependencies with other system assets still remains in the field of research. An autonomic approach to dynamic networking routing configuration implies a common monitoring and management interface on the different levels of routing (packet routing, message routing, etc) and managers integrated into a common autonomic architecture [135]. This integration of the network-management with a database, application server or web servers provides flexibility and opportunity to optimize the information flow dynamically, on-demand and according to the current system configuration and the activity of the environment that it serves, for example high-volume traffic to specific nodes, unexpected node failure or scheduled off-line maintenance of a serving equipment.

2.2.5 Autonomic System Upgrade Support in Production Environments

IT systems are dynamically changing according to the demands of its users and the environment in which they operate. Agility and adaptability using software architectures has been a long-time research process that produced significant results [94]. Even in everyday life we see this progress by using programs based on modular approaches for adding functionality, we can see our operating system which updates itself automatically with the necessary security fixes. However, desktop system updates have presumably lower requirements when it comes for keeping the functionality of the system intact. The systems used in production and corporate environments, on the opposite, are sensitive to the updates of the system because small changes in either interfaces of the used components or their functional behavior may have unexpected influence on the behavior of the whole system. This is partially due to the complex infrastructure of such environments and partially because of the lack of coordination in development of the different frameworks [54, 63]. Before deployment, the updated components are tested in an isolated from live environment conditions. This does not give the necessary assurance that the deep dependencies between the updated components and other parts of the system do not conflict and will not damage the rest of the system operation. An autonomic management system for upgrade of software components in production environment has to facilitate the procedures of step-wise, gradual deployment by allowing deployment of new components in the live environment for purposes of real-world tests. This can be achieved by autonomic support for dynamic selection of interfaces for communication between non-conflicting components. In this way, new versions of components may reside together with the old ones and can be tested against the real environment. The bigger part of the prototype development that supports this thesis is dedicated to this scenario. The cybernetically viable architecture and the communication channel management environment allow for dynamic selection of component interfaces.

2.2.6 Pro-active Intrusion Detection Systems

System security has a high priority in open operating environments. Although investigation in methods for reaching high-levels of system security has advanced significantly, the majority of production systems still exhibit only passive security measures, such as restriction of physical, network or user access. The vision for an autonomic security support implies self-defense by means of passive and pro-active management [134]. A pro-active protection defines how a system reacts to events coming from the operating environment in order to predict potential security risks, such as intrusion or "Denial of Service" attacks. Approaches may include the ability of service migration, and network reconfiguration governed by decisions taken by autonomic managers that sense the events and plan according to a dynamic base of symptoms. The role of the managers is to support the decisions that potentially would absorb undesired condition and perturbations coming from open system interfaces. Reasoning for such kinds of purposes requires methods from the field of machine learning [127, 136] and stochastic models for prediction.

2.2.7 Integrated on-line service management

The mentioned examples of management (web server, database, network and software components) were analyzed only as stand-alone operating elements. However a scenario of on-line service provision requires integration of these components into a single domain of cooperation or communication. Such an example is when users access an on-line service through a web server which on its behalf activates service component for process of content served by a database over the network. This defines a clear set of dependencies between the elements and for the successful completion of the service they have to be satisfied. In a real-world scenario the elements are clustered to facilitate load balancing and network traffic-routing optimization. Then, the management procedures such as configuration and update have to be applied on the level of clusters and with regard to the existing dependencies inside the clusters [103] and outside them. The necessary actions have to be executed to adapt the elements in the remaining part of the system. On a higher level, the management system has to be able to follow the defined policy and execute it across the different elements and groups in a seamless manner. Figure 2.1 illustrates the dependencies in such administration domain and the management elements in its heterogeneous environment. The illustrated management interfaces are of different nature and must be integrated in a common management unit, noted on the picture as "Global System Management". The cybernetically viable architecture would facilitate this scenario with its organizational model of recursive and nested levels reflecting the way components are grouped and managed.

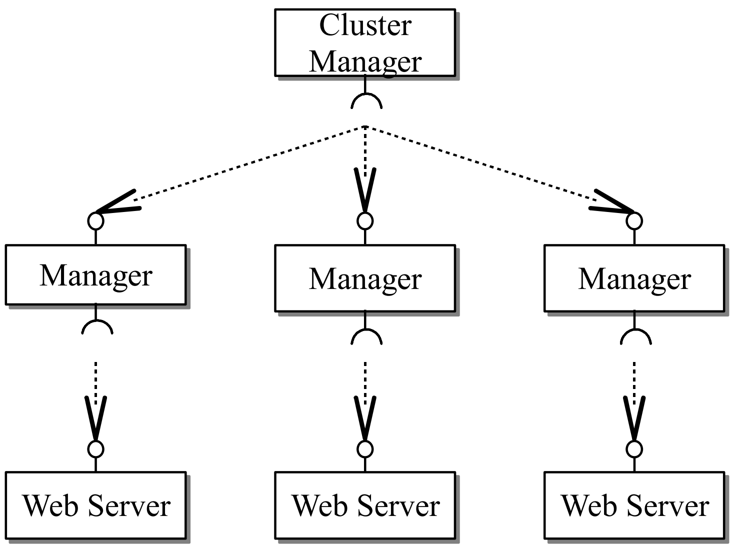
2.2.8 Distributed Service Delivery Solution Management

The penetration of IP connectivity in the recent years has allowed mobile devices to connect to local networks or to the Internet and serve content directly in the palm of their users [21, 99]. The possibilities of this development are expanding and practically difficult to measure. There are many research projects in the area of pervasive and ubiquitous computing that investigate scenarios and architectures for location and context-based services [108], however, there are many problems to be addressed in important aspects of device communication, such as common policy and dependency management. While there are already standards, such as Universal Plug and Play (UPnP) [98] that facilitate partially these two aspects, the problem of common management persists. An environment that hosts the devices has to be able to define policies for access and interaction between devices, devices and users, and users and services. Figure 2.2 illustrates the existing communication functional and management dependencies and the need of management integration in the scenario of a Smart Home Solution. The end-user side on the left of the picture consists of communicating devices which access services through a deployed service gateway. The devices are dependent on it and on the network equipment, either wired or wireless. The solution provider integrates security services along with an architecture for delivery of services from third party service providers. This setup demonstrates the usage of heterogeneous management and functional dependencies and has similar characteristics as the scenario from Section 2.2.7 - distributed communication that demands common management framework. An autonomic architecture may help in this situation with a middle-ware for adaptation of communication channels and remote dependency management.

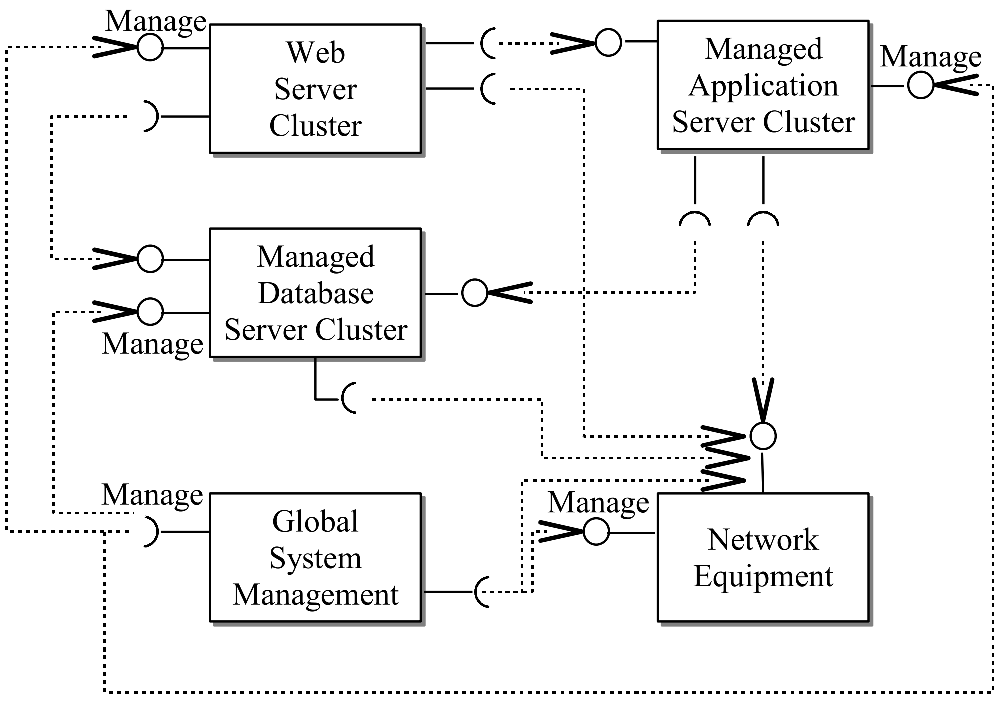
As seen from Figure 2.2, there are multiple points of management, where there is a need for co-ordination, policy distribution and a common information base. An autonomic solution provides these missing features to enable automated management and reduce the costs.

2.2.9 Rapid Prototyping Systems Support

The systems that support the process of *Rapid Product Development* (RPD) [118] have high requirements for adaptability and functional richness. Modern RPD systems are by intent designed for distributed deployment and include a number of technologies that demand integration and maintenance [123]. An example scenario for management of a system that supports the process of RPD is illustrated on Figure 2.3. A typical RPD system consists of a large variety of components, such as image acquisition systems, simulation software, modeling software, documentation



(a)



(b)

Figure 2.1: Dependencies in distributed environment for on-line-service provision. Dependencies in a cluster of managed web servers (a) and dependencies between managed clusters as elements of a system for on-line service provision (b).

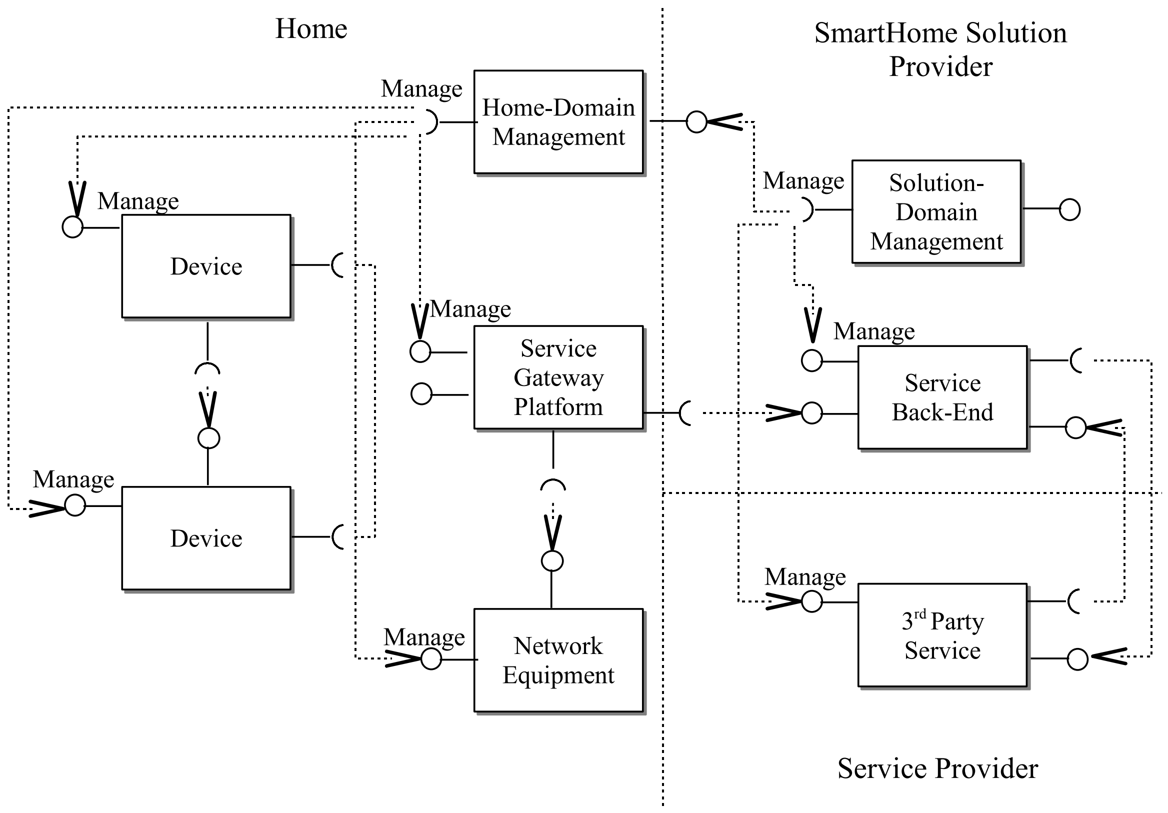


Figure 2.2: Communication dependencies of service delivery in Smart Home scenario. On the left side home-wide dependencies between devices, network equipment and service gateway platform, on the right bottom third-party service delivered through an integration with a back-end on right top.

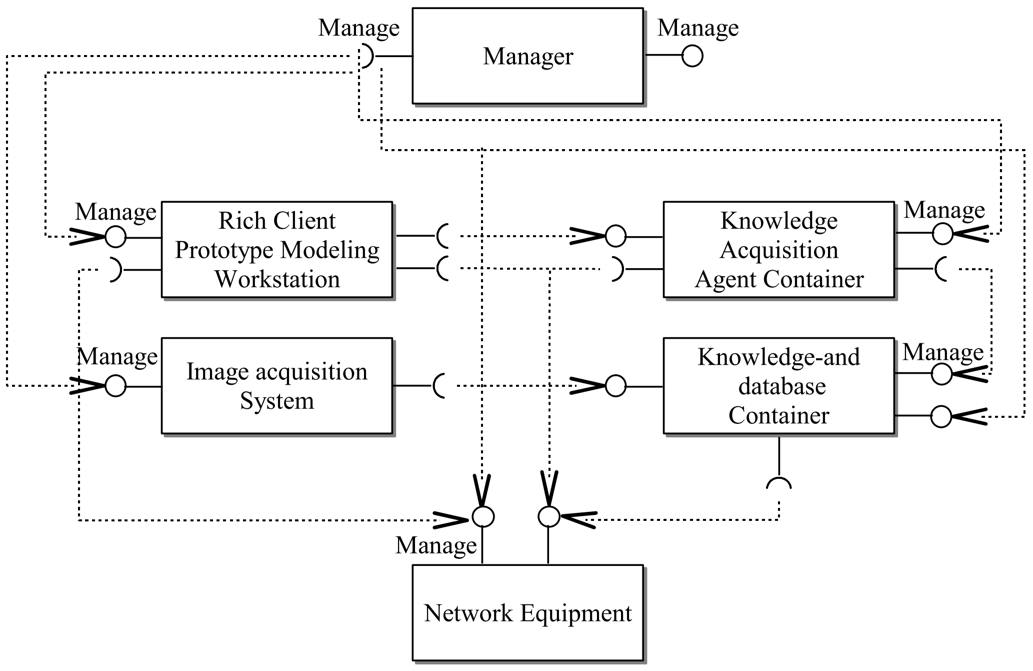

Manager
Manage
Manage
Rich Client Prototype Modeling Workstation
Manage
Knowledge Acquisition Agent Container
Manage
Image acquisition System
Manage
Knowledge-and database Container
Manage
Manage
Network Equipment

Figure 2.3: Components in a managed distributed RPD support system

and model knowledge base, pro-active data retrieval, etc. A specific characteristic of RPD is that the requirements and the specifications of the modeled product change dynamically and that requires functionality of the supporting systems (hardware, image processing software) to be updated with new features while the communication process is still active. At the same time, the requirements for stability are high because a small glitch in the intensive communication between development teams and respectively between the components of the supporting system can cause undesired delays in the progress of the project. That is why all these technologies need management infrastructure and coordinated software deployment. The management component has to integrate the various interfaces in order to coordinate the managed resources and facilitate comfortable management of the heterogeneous dependencies in such system.

An architecture that supports integrated communication management would play an important role in stability control in the process of system adaptation.

2.2.10 Distributed Component Systems Management

This section gives an overview of required software elements for management of component-based systems that enable a system's state to be monitored and manipulated. They form the basis of a management infrastructure with the help of which autonomic support assets can be built, such as policy management, self-configuration and adaptive communication.

2.2.10.1 Management Consoles

Regardless of the efforts for fully-automated system management, the end consumer on the side of the management system remains the human being with assigned role *"administrator"*. The administration tools of the modern frameworks for distributed computing provide means for visual and graphic representation of the management data. These tools and applications are called management consoles. Their single purpose is to make the data easy to handle and to give the administrator a comfortable way for interacting with the system configuration, system tuning and profiling. The degree of integrated management of a management system and the environment depends highly on the heterogeneity of the deployed systems.

2.2.10.2 Runtime Management

Because the components are hosted in an environment that manages their life-cycle it is a common practice the monitoring and the management of the components and their containers to be performed through interfaces provided by the environment itself. An example of such integrated management is the *Java Management Extensions* (JMX)-based management of Java 2 Enterprise Edition (J2EE) platform [27, 82]. Per specification it is built on top of the management framework that creates a flexible and manageable environment. Additionally, the latest specification of the *Java Virtual Machine* (JVM) that virtually hosts the platform uses the same framework to expose its management data. That way management of components of a distributed system can be managed through a single console remotely, on different levels of encapsulation. Similar example is the management of Service Logic Execution Environment (SLEE) containers and its building blocks the implementations of which take exactly the same approach towards integrated management. *.NET Common Language Runtime* (CLR) provides management interfaces for monitoring of the hosted components and similarly .NET-based applications can be monitored remotely and using single integrated management console.

2.2.10.3 Management Adapters

Although the software vendors are trying to deliver integrated solutions for their technologies the heterogeneous nature of the distributed systems in production and corporate environments requires specific management interfaces to be adapted for unified management access in order to reduce the problem of management granularity. This is achieved by using adapters that translate the events and management operations in both directions - from the management application to the concrete management interface in case of operations and from the interface to the application in case of events or indications. The place where adapters are used depends on the approach and the architecture of the unifying management framework. For example, an adapter that is used on the side of the end-user management application or console translates the target management interface operations and events into unified graphical representation. The other way for design of common access to management interfaces is to use middle-ware that abstracts the management operations into a common access interface and exploits a normalized information base. Then the adapters are used as pluggable building blocks [22] for reflection of environment's real state. This approach can be seen in the way DMTF framework specifies the architecture for building CIM-based management instrumentation systems.

2.3 Architectural Elements for Autonomic System Design

This section makes an overview of the used in system design elements and defines the nature of autonomic systems in abstract terms. These include definitions of building blocks such as autonomic managers, elements and control loop flow that constitute the autonomic character of a management system.

The autonomic systems are integrative collection of *autonomic elements* [46] - individual system constituents that contain resources and deliver services to humans and other autonomic elements. System self-management arises as the result of interaction of the autonomic elements according to a predefined management policy and shared knowledge.

An autonomic element consists of:

- ***Managed element(s)*** - equivalent to the elements in non-autonomous systems but can be adapted to enable to be controlled and monitored (by the autonomic manager)
- ***Autonomic manager*** - monitors the managed element and its external environment, constructs and executes plans based on an analysis of this information.

An autonomic manager is a component that implements a particular *control loop* over a managed element of the following form.

2.3.1 Managed Element

A managed element in an autonomic system may be any of the elements that participate in the system's functional or resource inventory. A managed element have clearly defined interfaces for management which serve as access points for monitoring and control. Autonomic computing defines the access points for monitoring as *sensors* and those for control are defined as *effectors* [131]. Sensors and effectors are both formally defined and the actual implementation of the autonomic system treats them accordingly to achieve parameter monitoring and management operations.

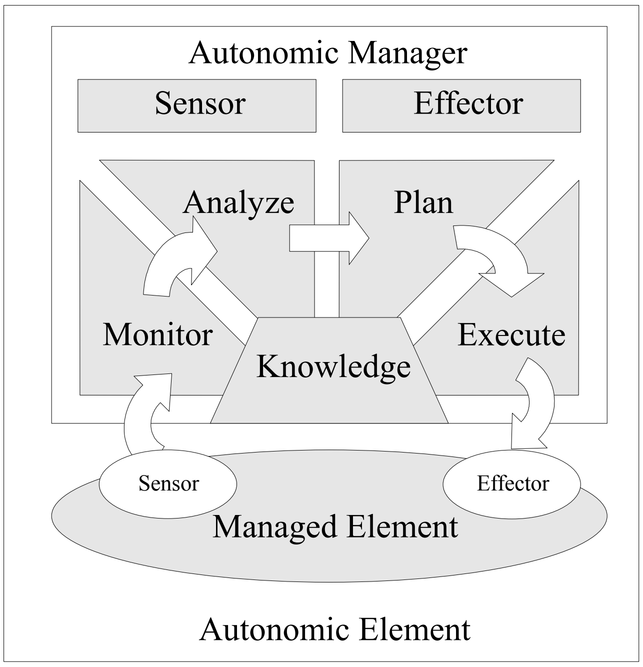


Figure 2.4: Autonomic Element with MAPE Loop-Control

The manager reads elements' state from its sensor (S) and executes management operations through its effector (E).

2.3.2 Autonomic Manager

An autonomic manager has the responsibility to monitor, analyze, plan and execute management operations over the sensors and effectors of a managed element [29, 56]. A main characteristic of the autonomic manager is that it is a standalone, autonomous and communicative element. Autonomic managers can on their side provide sensor and effector interfaces to allow higher-levels of management to have access to management information. The numerous autonomic managers in a complex IT system can work together to deliver autonomic computing to achieve common goals. For example, a database system needs to work with the server, storage subsystem, storage management software, Web server and other elements of the system in order for the IT infrastructure as a whole to become a self-managing system [4].

2.3.3 Control Loop

The flow of information and the steps of management interaction between manager and elements form an *autonomic control loop*. The control loop is conducted by the autonomic manager either by signals coming from the managed element's sensors, or by its autonomously initiated actions for monitoring and control. The process of management through autonomic control loop is divided into four major phases: ***m****onitoring,* ***a****nalysis,* ***p****lanning* and ***e****xecution*, known in the autonomic research community as ***MAPE***. Figure 2.4 illustrates coupled managed element and an autonomic manager with its belonging elements responsible for handling the four phases of the autonomic control loop.

Monitoring is either passive, event-driven or proactive, initiated by the manager, the monitoring phase processes incoming data from the managed element and adapts it to understandable for the analyzer format.

Analysis is the process in which the value of collected data is analyzed for particular patterns and constraints. This is the phase where the manager applies its policy and evaluates the state of the managed element.

Planning is a phase in which depending on the results of the analysis phase, the manager creates a plan for execution of actions on the managed element. The created plan should affect the state of the element in desired by the element manner and according to the system policy.

During the ***execution*** phase the manager performs management operations using the effector interface of the element.

2.3.4 Shared Knowledge

An important aspect of autonomic system management is the ability of the managers to reason with the help of a shared knowledge base. The knowledge base contains necessary policy instructions, history data or rules that all together help the manager in the analysis and planning phases of the control loop. Depending on the way the knowledge-base is organized in terms of knowledge representation and storage, it may vary from system to system [91]. A common understanding is that interoperability between autonomic systems is dependent on the degree of adoption of standards-based knowledge representation models, such as ontologies and usage of standard information models that can be understood and implemented inside every management system.

2.3.5 Autonomic Manager Communication and Cooperation

The sensors and effectors provided by the autonomic manager facilitate collaborative interaction with other autonomic managers. The managers cooperate in order to bring the system to a desired state communicating either by using direct peer-to-peer access (Figure 2.5) to each-others resources or by using communication services designed for the purpose. This way of cooperative control has been investigated in the research domain of multi-agent systems with efforts spent on protocols for communication, negotiation and brokering.

Managers may form hierarchies of management levels that determines the distribution of tasks and policies. Figure 2.6 shows how a manager may become managed

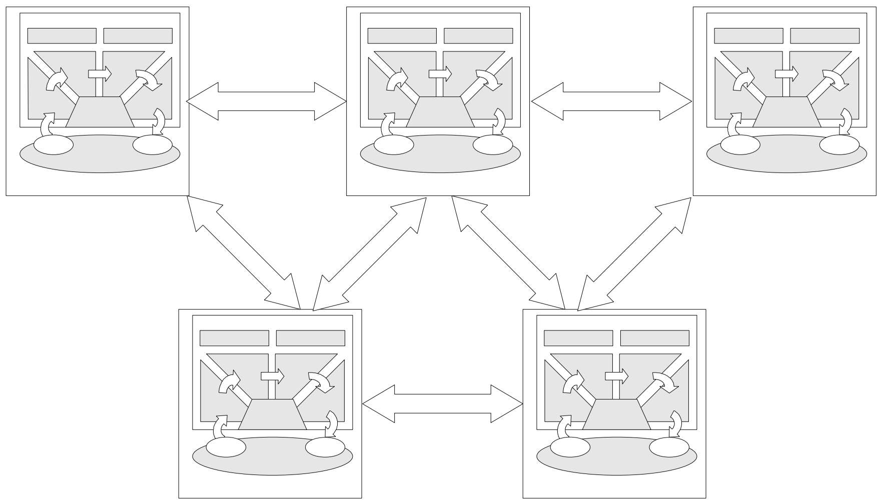


Figure 2.5: Communication between autonomic elements
It is achieved using exchange of information directly between peers.

element for another manager from a higher level of the system management hierarchy. This approach simplifies the management and defines clear way of policy and task distribution.

2.4 Principles of System Control

An important aspect of system management is the understanding of core principles of control. These include types of loops, balancing and models for system organization and control [5, 86]. This section makes an overview of the theory basics in the field of cybernetics related directly to system control. Section 2.4.3 reviews a model for complexity management based on cybernetic principles used as a reference and starting point in the design of a cybernetically viable software architecture.

2.4.1 Cybernetic Control Mechanisms

As seen in Section 2.3 the autonomic managers operate using control loops to regulate the state of the software elements. The field of cybernetics is covering the topic of system control and investigates practical and formal definition of *regulation* and of *control loops* operation. There are three fundamental methods to achieve regulation [70]:

Buffering is the passive absorption or damping of perturbations. For example, the wall of a thermostatically controlled room is a buffer: the thicker or the better insulated it is, the less effect fluctuations in outside temperature will have on the

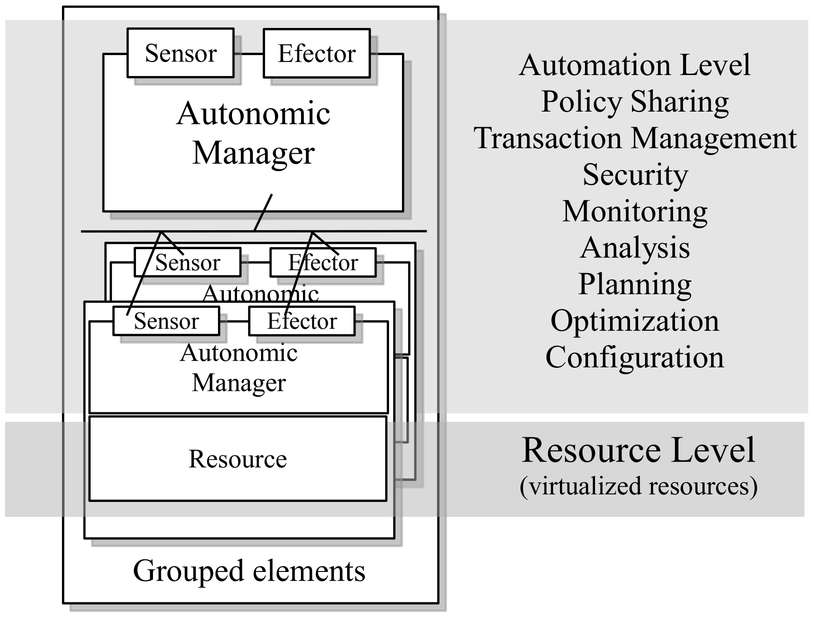


Figure 2.6: Hierarchy of managers.
Managers can be hierarchically grouped and controlled by higher level of management.

inside temperature. Feedback and feed-forward both require activity on the side of the system, to suppress or compensate the effect of the fluctuation.

Feed-forward control will suppress the disturbance before it has had the chance to affect the system's essential variables. This requires the capacity to anticipate the effect of perturbations on the system's goal. Otherwise the system would not know which external fluctuations to consider as perturbations, or how to effectively compensate their influence before it affects the system.

Feedback is a compensation of an error or deviation from the goal after it has happened. Thus, feedback control is also called error-controlled regulation, since the error is used to determine the control action.

The methods feedback and feed-forward are embedded in control loops. Feedback control loops have several characteristics that are related to system stabilization and growth:

Deviation in Time Performing a control loop means to measure the difference of the current state of a system compared to a state in different moment of time.

$$D_y = y - y_0 \tag{2.1}$$

where y is the current observed state, and $y_0$is a comparative (e.g. equilibrium) state. If we introduce change of time as D_t it can be presented in this way:

$$D_y(t + D_t) = kD_y(t) \tag{2.2}$$

Depending on the sign of k there are two types of feedbacks:

Negative Feedback Control loop which has a positive deviation at given moment t (increase with respect to given state y_0) leads to a negative deviation (decrease with respect to y_0) at the following time step $t+D_t$. Negative feedback is ubiquitous as a control mechanism in machines of all sorts, in organisms (for example in homeostasis and the insulin cycle), in ecosystems, and in the supply/demand balance in economics.

Positive Feedback The opposite situation, where an increase in the deviation produces further increases, is called *positive* feedback. For example, more people infected with the cold virus will lead to more viruses being spread in the air by sneezing, which will in turn lead to more infections. An equilibrium state surrounded by positive feedback is necessarily unstable. For example, the state where no one is infected is an unstable equilibrium, since it suffices that one person become infected for the epidemic to spread. Positive feedbacks produce an explosive growth, which will only come to a halt when the necessary resources have been completely exhausted. For example, the virus epidemic will only stop spreading after all people that could be infected have been infected. While negative feedback is the essential condition for stability, positive feedbacks are responsible for growth, self-organization, and the amplification of weak signals.

2.4.2 Variety and Knowledge Management

An important notion in system control is the absorption of *variety.* The controller has to be able to recognize perturbations coming from the environment and the element under control in order to take appropriate measures concerning the goal of regulation. This requirement is known as the *law of requisite variety* and is a fundamental principle in construction of management systems. It has the following formal definition: *a model system or controller can only model or control something to the extent that it has sufficient internal variety to represent it* [86].

For example, in order to make a choice between two alternatives, a controller must be able to represent at least two possibilities, and thus one distinction. If we take into account the constant reduction of variety K due to buffering, the principle can be stated more precisely as:

$$V_E \geq V_D - V_R - K \tag{2.3}$$

where:

V_E represents the variety of the essential variables of the system.

V_D represents the variety of the disturbances acting to the system

V_R represents the variety of the regulator of the system

K is a buffering constant, aimed to reduce the effect of the disturbances (reduce the variety of disturbances)

A trivial implication of this law is the operation of a thermostat. The thermostat percepts the room's temperature and has the goal to keep it in a predefined value. In order to do this, the limits of temperature deviation it is able to sense has to be at least equal to the extreme values of temperature change outsides the room. The buffer in this situation are the walls which compensate the temperature deviation to a certain degree.

An implication of the law within a scenario of software security management can involve a controller in the form of a port monitoring firewall bound to the network and analyzing the network activity which represent the disturbing factor. The goal of the controller then would be to keep the functional parameters of the network service (the essential variables) in range that satisfies the operation of the system. The controller has to be able to recognize variety of events that relate either to the normal operation of the system or those belonging to potential attacks and either mediate them or reject them. The functionality of the buffer in this implication can be represented by cascaded firewalls or other filtering facility in position to mediate the events coming from the network environment and directed to the service.

While variety absorption is important for the controller, it has to be able to decide and reason depending on the knowledge it has about the absorbed variety. That means, the controller has to have the *requisite knowledge* [5] in order to perform management operations. This knowledge can be in the form of rules and chains of rules that evaluate the conditions and map them onto actions that the controller has to execute to bring the system to a desired state.

The Law of Requisite Variety (LRV) and the law of requisite knowledge (LRK) are fundamental principles for verification of the ability of a controller to perform correctly and as expected. They are heavily used and required for a construction of a management model.

2.4.3 The Viable Systems Model

This section presents the basic characteristics of the *Viable System Model* (VSM), a system model organizational and communication principles for management of complex structures [9, 10, 20, 55].

Systems that adapt and sustain their operation in a changing environment are known as *viable*. The VSM Model developed by Stanford Beer focuses on management of organization, knowledge and communication flow in viable systems. Figure

2.7 illustrates graphically the organization, the relations and the encapsulation of elements according to VSM. The following conceptual properties characterize the model:

- The **law of requisite variety** (LRV) is the critical and most important law which has to be fulfilled for guaranteed system viability. The communication requirement is applied to the communication channels between the elements and and the elements themselves.

- **Self-reference** - each part makes sense in terms of the other parts. The system defines or produces itself based on the parts and their arrangement. This property is also called logical closure and is related to identity, self-awareness, self-repair and recursion itself.

- **Homeostatic**, **morphostatic**, **morphogenetic capabilities** - maintenance of critical variables within certain limits to ensure stability of a system in response to changes in the environment for both on the level of element management and structural adaptation.

The first thing to note about the cybernetic theory of organizations encapsulated in the VSM is that viable systems are recursive - *viable systems contain viable systems* that can be modeled using an identical cybernetic description as the higher (and lower) level systems in the containment hierarchy. Beer defines this property of viable systems as *cybernetic isomorphism.* Several principles and axioms of organization form define how a system using the VSM can keep its adaptability and stability in changing environment.

The ellipses on Figure 2.7 represent the external environment in which the system is embedded. The environment is interpreted as a set of current and future state of external parameters that influence the operation of the system. On the right side of the diagram there is a management system that controls plants and balances its communication and organizational structure to reflect current and future changes of the environment and keep the controlling system in balance. The set of rectangles labeled with System 3, System 4 and System 5 represents a higher level of management that performs the executive, planning and operations functions. The rectangle labeled as System 2 defines a *regulator* with main function scheduling and synchronization. These functions provide needed feedback loops to ensure smooth operation of the overall system. The *regulator* is responsible for maintenance of the operation schedules and for coordination with other controllers. An "algedonic" signal representing positive and negative (pain and pleasure) values of estimation for operational health of the controlling units is shown going from the controllers at the lower level directly to the executive function at the next level. This can be thought of as a "panic override" alert that bypasses all filtering in the System 3 and System 4.

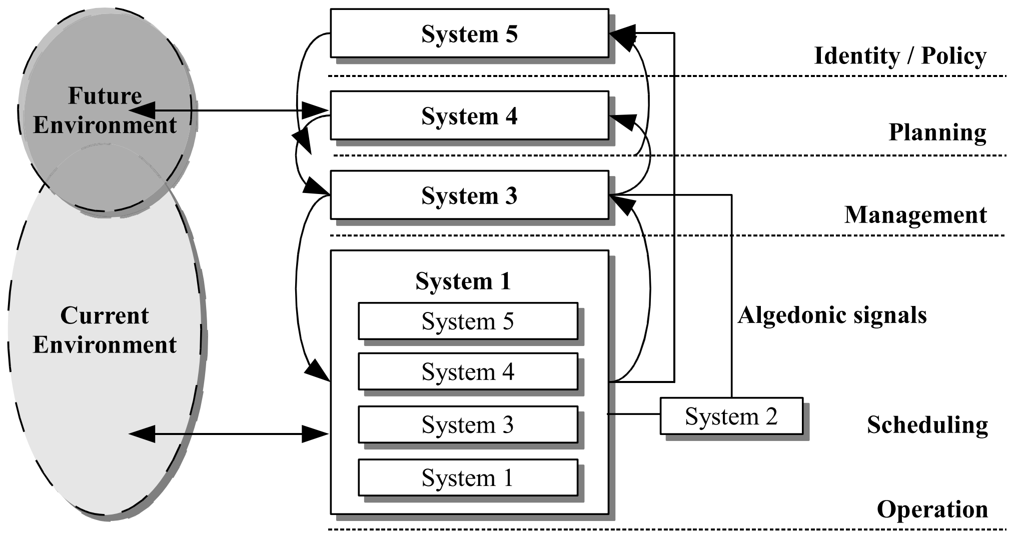


Figure 2.7: The Viable System Model

VSM has *homeostatic*, *morphostatic and morphogenetic capabilities* of adaptation. These correspond to System 3 and the combined Systems 4-3 and 5-4-3 (see Figure 2.7).

Homeostasis is the maintenance of critical variables within certain limits to ensure stability of a system in response to changes in the environment. This is normal control system behavior. *Morphostatic* and *morphogenetic* adaptation relate to the *structural adaptation* of the system. Morphostatic behavior is "simple adaptation" such as changing internal control algorithms. This corresponds to the 4-3 function. Morphogenetic systems maintain meta-properties of the system (“identity”) through evolution of the structure and/or components that make up the system itself. That is, the ability to acquire new components and discard others. This is the full 5-4-3 controller function. This combination forms a “supervisory-adaptive” controller for a management system that implements homeostatic, morphostatic and morphogenetic policies to manage and maintain the stability of a controlled plant. The viability of a system according to VSM is a measure of how well these policies are realized in a particular environment.

2.5 Autonomic Component Management Challenges

The task of enabling autonomic support in existing systems requires resolution of several problems related to the heterogeneous nature and the distributed communication in component-based software. The problems addressed by this dissertation are only a part of a larger set of challenges in this field. More specifically, it addresses problems that relate to system organization and integration providing a basis for experimenting with and development of approaches that may address challenges beyond the scope of this work.

2.5.1 Overhead in component deployment management

The variety of applications that can be deployed in a production environment introduces difficulties in the support of the system during its lifetime. The support may include installation of new components, frameworks and services, upgrade, or de-installation. The necessary carefulness with which every step of modification in the system increases with system growth. In a distributed environment these difficulties are even bigger due to the decoupling of the elements and mediation of their communication over a network [23].

Variety of component models: The administrative human force that is responsible for the support of the system has to consider the differences in the deployment life-cycle of the component and adjust its configuration according to the requirements of the hosting container. Sometimes, although by specification the component models of different component-based framework implementation are the same, there may be specific configuration settings that have to be kept in mind in the process of system modification. An example for such allowed deviations are the specific container configurations of J2EE implementations with services defined out of the framework specification.

Intra-model variety: The same way the functionality of components may evolve, the component models that facilitate deployment of the functionality evolve too. A component container may support several versions of the same technology specifications depending on the concrete pace of evolution, but in the case of a major re-design of the model it may be required to deploy of a separate container supporting the new version of the component model specification [39]. This introduces additional need of attention and carefulness in the support process of the component systems.

2.5.2 Management Granularity

The variety of platforms and frameworks that a heterogeneous environment may deploy introduces variety of management solutions accompanying these frameworks. For example a J2EE EJB container can be managed by means of JMX beans [43] and enabled consoles, while the management of a OSGi framework may be independently selected by the implementation vendor. The management of system assets, such as databases or operating system services is achieved using different sets of management standards, agents and consoles.

Proprietary management solutions: Include proprietary management back-end and closed specifications of the interfaces and protocols. These solutions limit themselves to a set of interfaces and extensions on the side of the management console. Deployment of proprietary management solutions is so far used in a specialized cases of high-end or advanced systems, such as virtual machine appliances, corporate search appliances and device management interfaces. A main disadvantage of proprietary management is the limitations of flexibility in bridging, generalizing and integrating the management into a single model. That is why autonomic computing requires the elements of the system to support open and standards-based solutions for access to their management information.

Standards-based management: Current software component frameworks allow management using management interfaces and protocols that are open and publicly available. However, depending on the type of managed elements, the policy of the vendor and its preferred management protocol, the frameworks are shipped with management back-ends and consoles based on a variety of standards differing in their architecture, implementation or version of the specification. This creates a large variety of management channels that are difficult to integrate. The problems of integration can be related to interaction, architectural or semantic. The interaction problems are related to the way a managed element communicates with a manager, including protocols and data types. The architectural problem arises in the case where the management interfaces are provided by components on different level of containment. For example management of J2EE components happens through the management interfaces of the runtime, while OSGi environment is accessed through a regularly deployed bundle. A management architecture has to consider this type of arrangement in order not to break any dependencies. Semantic difficulties in management integration are related to the way the different management frameworks are mapped onto a common terminology. For example the information bases of SNMP and CIM define different deepness of relations in the hierarchies of classes. Mapping onto a common model by definition will miss semantics that may be important for the management of certain components.

2.5.3 Parallel component evolution

Distributed systems are not only separately maintained but often their development happens independently [117]. A component in one part of the system may be subject of continuous development and upgrade from one software vendor, while others are developed by different vendor. This may introduce inconsistency in the functional and communication aspects of component's evolution and cause severe damages, regression and influence negatively the overall system health, thus regression testing and verification [2, 40, 64, 107, 114] becomes extremely difficult.

Evolution in component communication: The interfaces that a component provides for communication with other components have direct dependency with its functional function. Often components serve as a common functional element in systems of different nature that were created for different purposes, thus changes in component interfaces reflect not only the functional or architectural evolution of its internal implementation but also the goals and the needs of a larger set of systems or subsystems, known or unknown in the domain of component deployment. The aspect of interface compatibility is not the only requirement. The act of communication involves interpretation of the semantics of the interfaces and while the interface vocabulary or way of access remains the same the meanings of the concrete messages in the interface may differ. The differing semantics are often related to the concrete implementation and functional evolution of the component that often remains uncoordinated between parties dependent on it.

Lack of coordination in development: Broken functional or semantic dependencies between components in separate parts of the system may cause inconsistency in the evolution of the whole system. An example is deployment of components that were developed by third parties having multiple customers. Without oversight on the direction of development the way a component will evolve is uncertain and not in the domain of control of the deploying party. Community based development solves this kind of problems but this happens usually on the level of system interfaces or on the level of abstract component model definition. Concrete examples for such kind of development are the Java Community Process (JCP) [26] responsible for the evolution of Java core interfaces and specifications, OGSA Consortium developing the OSGi standard or the working groups for development of the CLR standard. However, on the level of component communication evolution is strictly bound to the concrete business needs of the consumer and the problem remains unsolved.

2.5.4 Deep component dependencies

Components are used as building blocks for functional re-usage. A system contains multiple components that are dependent in a different ways. A distributed component system has the characteristic of remote component dependency where components from a local container depends on functionality that is provided by a component located in remote container. Additionally component functionality may be dependent on its container, both for because of its life-cycle management function and concretely on services provided by the container.

Local dependencies: A component-based application is usually delivered as a set of bundled components that are deployed in a hosting component container. These components may depend on each other's functionality in local manner, by using the interfaces directly or with the help of services provided by the container. The communication is not dependent on network operations, such as marshaling and unmarshalling of data parameters and can be monitored by means of interception of local events. The closed nature of such communication allows for faster problem discovery such as inconsistencies in communication interfaces, configuration and deployment parameters. In the category of local dependencies belong those between components and local container services, such as naming directories, timers, access to data sources and others.

Remote dependencies: Distributed applications are deployed on multiple hosting containers in a networked environment. The components may perform both local and remote interactions with other components, thus there are local and remote dependencies. A remote dependency adds a degree of complexity with the need for networked communication in the form of remote procedure call, interception or event handling. These require network availability and successful marshaling and unmarshalling of data types and values upon sending and receiving of data. This way, a remote dependency is not determined only by the component interface semantics but also by the capabilities of the elements that build the communication medium, such as Remote Procedure Call (RPC) proxies, network service access points, etc. Remote dependencies are hard to handle if both communication points do not provide information about their provided and used interfaces. A workaround for managing remote dependencies is the usage of a central deployment tool that keeps track of the deployed units and analyzes the dependencies prior to deployment. However in a distributed environment the need of a common component distribution and deployment infrastructure complicates the integration and introduces heavy dependencies between the components and the central deployment tool.

2.5.5 Management Complexity Overhead

If a system was not planned with management in mind the efforts of integration may vary and are expressed by modifications in the internal logic or larger re-design of the system. Similarly, integration of autonomic support for such systems may introduce problems with making the architecture more complex.

Automated management: Automation requires careful coordination of management flow with component life-cycle. To achieve this, a framework has to be able to expose its internal state to the management unit. Depending on the event system and internal organization of a component framework the control flow may be itself a complex system and as such influence the stability of the whole system. That is why an architecture for automated management integration has to be able to integrate with the rest of the system in an as simple as possible way using open interfaces for monitoring and management, with as little as possible required modifications on the side of the managed element. Ideally, the management sensing elements would have access to management interfaces and system state when they are deployed as components and co-exist with the rest of the deployed components.

Policy migration problems: While autonomic management promises more flexible and better control over the previously human-managed devices, the migration to a new system is accompanied by problems with adaptation and policy transfer.

2.6 Participating and interested parties in autonomic systems

Autonomic systems operate in an environment with many participators, such as users of the system, administrative staff, developers and service providers. This section shows how these groups benefit from autonomic management support and the problems that this thesis addresses.

2.6.1 Users

An autonomic system behavior serves the purpose of automated management, some aspects of which are accessible to the end user. Autonomic support for personal devices can enable intelligent user assistance depending on the location of the device or on provided context. Concrete example is the Smart Doorplates project [128] where door displays guide a visitor to the visited person, depending on the current location of both.

Another useful aspect of autonomic support for end-user applications is the automated setup [48]. When a user starts a newly installed application it can help configure itself with a high-level interactive policy setup. The details of the configuration are hidden from the consumer. Further changes of the configuration may happen either through the high level policy or through direct access to the settings. An example for this kind of interaction is the installation procedure of the IBM Autonomic Toolkit [37].

The user experience can be enriched with trouble-shooting procedures where problems are handled by the application itself by means of autonomic self-healing. The application of the proposed in this thesis architecture can facilitate self-diagnosis of end-user application and troubleshooting related to software updates.

2.6.2 System Administrators

Systems administrators are benefiting from autonomic systems in various ways, but mostly from the automation of error-prone and time consuming routines that autonomic managers perform according to the high-level policies set by them for notification, automated reaction and control. The security aspect of system administration in recent years is assigned a first priority in many organizations. In current IT systems distributed security updates are handled by centralized security authorities that have an overview and monitoring capabilities on the existing in the inventory systems. While this is efficient way for analysis of system dependencies it is often limited to non-heterogeneous sub-domains, while differing deployments are managed either manually or by a separate deployment scheme. An autonomic architecture that facilitates heterogeneous dependency modeling and automated distributed deployment can increase the level of automation, thus the level of security.

2.6.3 Developers

A major problem in finding malfunctioning components in distributed environment is due to the quantity of dependencies and interactions between the distributed elements [41]. The developers who are involved in the development of the system will have a direct benefit from the ability of the proposed architecture to manage and monitor remote interactions that help in discovery of inconsistencies and wrong operation. For example, the developers can have the opportunity to easily find the place in the code for an intended communicative act.

2.6.4 Cooperating Service Providers

A distributed environment that spans several business domains may involve multiple service providers that share functionality and depend on each others' services [79]. An aspect of interest for them in deploying autonomic support is the ability to share common evolution policy or track parallel evolving components in a way to keep the operation of their dependent systems in healthy conditions. The architecture proposed in this thesis supports this kind of interoperability and policy distribution.

2.6.5 Manufacturers and suppliers

On a global scale, the deployment of autonomic support for IT systems helps organizations, such as manufacturers and service suppliers, to adapt their systems faster to the changing requirements of their customers, increase the security and availability of the system. Besides IT-related benefits companies can achieve savings on system support with automation of routines and optimize their human resource power.

Chapter 3

Related work

This chapter presents the state of the art in research relevant for the subject of this work. The list of projects is separated in the following categories - projects directly dedicated to autonomic self-management, projects related to automated component management, research in software evolution, reliable systems research and related techniques and methods in development of distributed component systems.

3.1 Autonomic Computing Research Projects

The projects presented in this section focus on development of architectures and tools that enable autonomic self-management. They target directly the specific autonomic assets self-configuration, self-healing, self-defense, self-optimization and are incepted with autonomic architectural style as a main approach for system design.

3.1.1 AMUSE - Autonomic Management of Ubiquitous Systems for e-Health

Group: Imperial College, London, University of Glasgow

Research goals: Investigation of models and architecture to facilitate efficient usage of mobile devices and physiological sensors for the health industry. Specific area of research is deployment of autonomic applications in ubiquitous e-health environments. Additional goals of the research include definition of architecture, interaction and implementation of a self-managed cell (SMC) as a basic architectural pattern for implementing self-management at both local and integrated system levels, peer-to-peer coordination for dynamic composition with the help of SMCs, identification and implementation of protocols for interactions and investigation of techniques and tools for management of small and and large scale networks and applications.

System description: The system is built on top of the concept of self-managed cells where cells are combination of hardware and software elements that form an administrative domain being able to function autonomously and are capable of self-management. Interaction between cells is achieved with the help of exchanging events over an event bus. The SMC includes management components that provide contextual information and service discovery.

Interesting aspects: The SMC is not dependent on the underlying hardware and in this way the operational details are transparent for the higher levels of management and communication.

Related Publications: [38, 124]

3.1.2 AutoMate

Group: Applied Software Systems Laboratory (TASSL), Rutgers University, State University of New Jersey

Research goals: Investigation of key technologies, including programming models, frameworks, and middle-ware services, to enable the development of autonomic Grid applications that can address the challenges of complexity, dynamism, heterogeneity and uncertainty in Grid environments. Its overall goal is to develop conceptual models and implementation architectures that can enable the development and execution of such self-managing Grid applications. Specific issues that the group is investigating is design, development and deployment of autonomic middleware services that support policy and context driven execution and management of autonomic applications. Other research effort includes dynamic composition of autonomic applications for dynamic deployment and execution in run-time.

System description: The developed system integrates a component framework, deductive engine, dynamic role-based access engine, a peer-to-peer messaging substrate and a decentralized discovery service. These components govern the communication and composition on the different layers of the system - system layer, component layer and application layer.

Related Publications: [1]

3.1.3 Autonomia

Group: High Performance Distributed Computing Laboratory (HPDC), University of Arizona

Research goals: Investigation of autonomic computing infrastructures. Main goal of the project is to develop an environment with a complete set of tools to specify the appropriate control and management schemes to maintain any quality of service requirement or application functionality.

System description: The system is composed of several modules: The Application Management Editor for specification of control and management policies, The Autonomic Middle-ware Services module for provision of common middle-ware services and tools needed by application and systems to automate operation, The Application Delegated Manager responsible for configuration of application execution environment and a runtime requirement maintenance and a Monitoring Services module for components and system resource monitoring. The main principle of operation is template processing according to application component and resources registered in a distributed repository.

Related Publications: [59, 60]

3.1.4 IBM Autonomic Toolkit

Group: IBM Almaden Research Center Research goals: research in architectures for autonomic systems, development of toolkit for enabling autonomic management, communication and interfacing, investigation of efficient algorithms for event correlation and problem determination in distributed environments.

System description: The ongoing development of the toolkit is directed in first place towards scenarios for autonomic management, such as installation and deployment. It provides several application programming interfaces for development of autonomic managers, managed resources, communication protocols, policy engine and a heterogeneous workload management solution that implement the guidelines of autonomic computing design with a MAPE management cycle. A key component of the framework is the Log and Trace analysis tool which enables log normalization and event-correlation from different sources, such as web servers, databases, application servers, etc. Reasoning is facilitated by an integrated agent toolkit ABLE (Agent-building and Learning Environment).

Interesting aspects: IBM coined the term “autonomic computing”

Related Publications: [37]

3.1.5 Organic Computing -AMUN (Autonomic Ubiquitous Middle-ware)

Organic Computing (SPP1183) is a program funded by the German Research Foundation.

Group: Embedded and Ubiquitous Computing Lab, University of Augsburg

Research Goals: Research goals: autonomic management in ubiquitous environment with accent on self-configuration, self-optimization and self-healing. The belonging projects are investigating biologically-inspired approaches for IT system management with accent on self-management capabilities and complexity management. Other research topics include reconfigurability, emergence and self-organization. The project "Autonomic Ubiquitous Middle-ware" investigates autonomic management in ubiquitous environments.

System description: The developed middle-ware is based on peer-to-peer technology. For realization of self-management the middle-ware is extended by an autonomic manager and interfaces to add monitoring capabilities. The manager itself is responsible for the configuration of a single node, where communication between managed nodes happens by exchanging messages containing information about the state of the monitored node. The manager is abstracted from actual control algorithms that govern the decisions for reconfiguration of the node.

Related Publications: [129, 130]

3.1.6 SARDES - Jade

Group: Institut National de Recherche en Informatique et Automatique (INRIA), France

Research goals: Investigation of the construction of distributed software infrastructures (operating system and middle-ware) to support global computing. Global computing is concerned with a projected environment in which processors will be everywhere and will be interconnected by a diverse array of networks, from ad-hoc pico networks to the global Internet.

System description: The system implements a middle-ware for management of distributed software components. It uses the traditional control loop and managed elements principles, but with the addition that it defines a common component model for specification of interfaces for different aspects of management such as configuration and reconfiguration. The architecture supports management of distributed elements with layered and nested type of composition. For the different types of management aspects the framework uses different classes of controllers such as attribute controller, life-cycle controller, content-controller and bindings controller. The architecture proposed in this work uses similar approach for abstraction of components on different levels of system organization.

Interesting aspects: The component model is described using ADL and additionally to the support for nested and layered managed components, it is able to represent the composition of the autonomic administration software itself the same way.

Related Publications: [13, 14, 24, 113]

3.1.7 PUSH - Policy-Based Update Management in Smart Home

Group: Mobile and Distributed Systems Group, LMU Munich Research goals: the project investigates the adaptive configuration and autonomic bug-fixing in embedded systems with focus on Smart Home scenarios.

System description: Device firmware updates are scheduled by a dependency manager and policy engine that takes in considerations the preferences of the device users in a way to bring the system to a consistent state. Dependencies are modeled as a non-cyclic graph and are traversed according to version compatibility tables. The dependency manager detects conflicts in update phases and reschedules the update accordingly.

Related Publications: [33]

3.2 Software Component Management Research

Research initiatives in the domains of system and software management, software evolution and reliable computing accumulated valuable knowledge that can be directly applied in the development of approaches for autonomic system design. The following sections make an overview of the existing projects from these domains for their relevance to the subject of this work. The projects presented in this section are related to software components and distributed system management. Important and relevant to autonomic management aspect of this research is the automation of component life-cycle and framework integration.

3.2.1 SOFA - Software Appliances

Group: Distributed Systems Research Group, Department of Software Engineering, Charsles University, Czech Republic

Research goals: to design and implement a platform for dynamically updatable software components and integration with Common Request Broker Architecture (CORBA) middle-ware

System description: the project defines a Component Description Language (CDL) with the help of which components are defined and their description is stored into a common repository. The connectors provide support for software systems where all application components contain application logic only, while the connectors implement the necessary interaction semantics and cover deployment dependent details. The connectors solve the deployment anomaly problem.

Interesting aspects: The system provides a set of templates for connectors of different types, for example procedure calls, event delivery and streaming. An environment for development of SOFA applications consists of set of nodes distributed and bound to hosts in a network. The architecture allows update of the components in runtime where updated parts of the components are associated to their versions. The system defines control objects for the purpose of component management.

Related Publications: [66, 115]

3.2.2 IRISA "PARIS" - Adaptive Software Components

Group: Institut National de Recherche en Informatique et en Automatique (INRIA), Institut National des Sciences Appliquees (INSA)

Research goals: Project PARIS aims at improving to the programming of large scale parallel and distributed systems. The projects has several sub-projects dedicated to research in different software development domains, one of which is adaptive components. The Adaptive Components project targets component run-time adaptation and optimization with accent on component deployment in Grid computing.

System description: The basic approach taken for implementation of the system is very similar to the architectural guidelines for autonomic computing. Management of components is separated from the components themselves and by means of event monitoring the management unit executes actions according to a predefined policy.

Related Publications: [3, 34]

3.2.3 ADAPT - Middle-ware Technologies for Adaptive and Composable Distributed Components

Group: cooperative work of several universities with coordinator Technical University of Madrid

Research Goals: to provide open-source middleware support for the creation of adaptive and composable web services. Concrete subjects of research are development of self-descriptive basic services, composite service definition and enactment of business processes, adaptable both basic and complex services.

System description: The core principle of system operation is abstraction of work-flow and interfaces for composition of web services. The developed middle-ware is capable to perform replication and dynamic implementation execution defined by selective, optimization and information policies. The service composition characteristics can be described by means of a service specification language. Non-functional compositional service properties govern the decisions of the work-flow engine according to the specified policies.

Related Publications: [132, 133]

3.3 Research Projects on Software Evolution

The research projects in this section had as a goal to abstract a system of processes and rules that govern the development of complex software systems. This field of research is relevant for the topic of the thesis for giving a higher and common view on the processes that are involved in software engineering and are accompanying part of the system development.

3.3.1 FEAST Projects

Group: Department of Computing, Imperial College

Research Goals: Studying of presence in and impact of feedback on the global software process of software evolution. Additional subjects of research are analysis of metrics for software evolution and its dynamics in small and large systems

Method description: The project examines a set of metrics and behaviors that govern the feedback loops in the process of software evolution in E-type systems [75, 76]. Main target of the project was to explore the FEAST hypothesis: *"As complex feedback systems, E-Type software processes evolve strong system dynamics and a tendency towards feedback based global stability"*. The approach of analysis includes tracking of the metrics throughout the development of several large projects. Among the investigated characteristics that influence system stability are system growth and system dynamics. FEAST/1 concluded respective system growth and dynamics models on base of which predictions can be made for how software management influences system development. FEAST/2 extended the research to examine the concluded laws of evolution with deep analysis by more evidences found in development of large projects. The results of the FEAST projects show an side of software development unexplored until that time with its own laws that have to be taken into consideration when developing the system. The laws of software evolution are valuable guide for planning software process with no regard to the type or the size of the system. The relevance of this research to autonomic system management is expressed in the Eighth Law of Software Evolution: *"E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems"*. It shows that solutions based on feedback systems can be applied not only inside the system, but on the level of software management and development processes.

Related Publications: [75, 76, 77, 78]

3.3.2 ESPRIT RENAISSANCE Project

Group: University of Lancaster and numerous companies as partners, project funded by the European Union Technologies RTD program.

Research Goals: To develop a systematic method for system evolution and re-engineering which is geared to the requirements of the commercial systems domain. The research involves development abstract model for software evolution and investigates evolution strategies.

Method description: The developed abstract model consists of layer role categories: strategic, operational and service categories. There several view points for studying of legacy systems: technical, economic, managerial views that filter respectively the different sets of factors driving the evolution. While the abstract model represent a static assignment of roles and views, the process model describes activities that involved individuals have to perform. The research dedicated to evolution strategies defines the concrete methods for modification of the system classified in three groups: continued maintenance, re-engineering, replacement.

Interesting aspects: the orchestration of roles, their assignment in the process of system evolution and the evolution strategies identifies the forces for concrete development steps.

3.3.3 COSE - Controlling Software Evolution Group: Software Evolution and Architecture Lab, University of Zurich

Research goals: To investigate means to analyze and control the evolution of a software system at various levels. Particularly interesting for the project are the relations of architecture and evolution, hidden dependencies in system's evolution, filtering and visualization of software data, including source code, bug history and release data.

Method description: The method is a combination of visual representation of data collected and techniques of code inspection or reverse-engineering. The analysis is separated in three aspects: Architectural Control, Change Dependency Control and Evolution Visualization. Structural dependencies in systems are analyzed and visualized in a way to demonstrate the influence and effect of code change in different parts of the system. The development of software architecture is tracked throughout the development of the system in order to see the driving forces for its changes and respectively the negative effect on maintainability. History reconstruction helps in identifying the concrete reasons for architectural decaying.

Related Publications: [42, 97]

3.4 Research Projects on Reliable Systems Design

This section describes research projects in the field of reliable computing. Methods and software architectures developed in these project influence design of autonomic systems in aspect self-protection and service availability.

3.4.1 ROC, Recovery Oriented Computing

Group: Reliable Oriented Computing Group, Berkeley and Stanford Research goals: system wide undo, integrated diagnostics support, survivability architecture

System description: Recovery Oriented Computing is a method of system design where error is accepted as an inevitable occurring event. The system tries to support discovery of errors and recovery of system services using mechanisms like monitoring, redundancy and automated restart, undo. System wide undo is achieved using an “undo proxy” and “rewindable storage” that wrap the application service in order to track events and user actions. The user service remains virtually unchanged. The proxy and the storage are managed by an “undo manager” that tracks the system time-line log. The rewindable storage is a system layer that allows roll-backing of system state to a prior point in time.

Interesting aspects: The system provides a model that is capable of defining domains of undo without modification of the user applications.

Related Publications: [17, 18, 96]

3.4.2 SABER - Survivability Architecture: Block, Evade, React

Group: Department of Computer Science and Electrical Engineering, Columbia University

Research Goals: Development of software architecture for system self-protection and service survivability under active attacks. Intrusion detection and reaction, DoS attacks.

System description: The architecture integrates Secure Overlay Services component, a set of intrusion detection systems, process migration, autonomic software patching and event-based command and control organization. The autonomic software patching system takes care of isolating patched and previously vulnerable software elements in parallel with the operating system, tests it against the known issues and deploys them into back into the environment.

Related Publications: [67, 71]

3.5 Common Approaches

This chapter reviews commonly used methods for interaction with and management of distributed components. Understanding of these methods is essential for developing a management architecture for heterogeneous component environment.

3.5.1 Architectures for run-time management

Modern component systems run on platforms and containers that provide management interfaces for introspection of internal parameters. These interfaces can be accessed either locally or remotely. Examples are the JMX management framework and the .NET Common Language Runtime (CLR) [84] management interface. With their help system analysis can be performed in runtime, on demand and independently from the application logic of the deployed in the container applications. In the process of testing of different aspects of system performance and resource allocation the frameworks and the runtime itself can be profiled for later analysis and optimization. Among metrics of interest for monitoring are memory allocation, thread states, network traffic and database access speed. The management back-end of such platforms allow virtually full transparency of the system resources and framework specific parameters. These facilities make the monitoring of component system simpler and allow system observation without interference in with the internals of the framework [44]. The operations performed by a console connected to the back-end may be setting and getting parameter values, remote or local execution of management procedures. The communication is bi-directional where the communication from the back-end to the console happens by sending indications or events for which the management console or client has been subscribed. Some implementations of J2EE use the management framework JMX as a platform for micro-kernel design. A specific representative of this approach is the JBOSS EJB micro-kernel implementation.

3.5.2 Directory Services

When distributed component communicate with each other or with services provided by their container they often need as a first step to locate component or service implementations. As a common practice for this purpose component frameworks provide *directory services* [83, 89]. They are an important element in distributed component design and management for their central role in deployment and distributed access. Using a directory service components are able to announce their presence to other components and the way to find them. When there is no need to access the component any more it is unregistered from the directory. Implementations of such services are the Java Naming and Directory Interface (JNDI) used in EJB and programs communicating using Remote Method Invocation (RMI) [104]. OSGi has its own implementation of service registry to support the blackboard-pattern for service-oriented component interactions. Windows Common Object Model (COM) component framework links dynamically communicating components that registered themselves with the framework's registry.

3.5.3 Information Models

System management would be impossible if there was no common way for the management system to describe the managed elements, their relations, properties and operations. This description is known as *information model* or *information base* [62, 81]. The information model is an abstraction of elements' management properties from the concrete implementation of the interface for access to the properties. Additionally, the model serves as a common vocabulary for the way a management console or agent communicate regarding the existing types of managed elements. *Distributed Management Task Force* (DMTF) defines as information model for their architecture the Common Information Model (CIM) [81]. CIM is an object oriented approach for describing of managed systems. CIM consists of a meta-model and a set of predefined models that define the relations between commonly used managed elements, such as policy models, software and hardware elements, configuration and deployment. The Tele Management Forum (TMF) has defined an advanced model for the Telecommunication Service domain - the Shared Information/Data model(SID). SID provides the common language for communicating the concerns of the four major groups of constituents represented by the four Views: Business, System, Implementation and Deployment defined in the NGOSS Life-cycle [51]. The model allows to create a bridge between the business and Information Technology groups within an organization, providing definitions that are understandable by the business, but are also rigorous enough to be used for software development. Both DMTF CIM and TMF SID use the UML notation to express their models.

3.5.4 Management Agents

A common approach in system management is based on the client-server model with a *management console* as client and *management agents* performing the server role. The management consoles are applications that visualize monitored system parameters and send management requests. The agents are usually located on the system under control and have access to its resources. They can perform the task of management request translation if the monitored device has its own management interface. This behavior is similar to a management adapter. Another purpose of management agents can be aggregation of system parameter aggregation or management request proxy operations. In the case of Simple Network Management Protocol (SNMP) [61], for example, the management architecture utilizes this model with the help of management daemons running in background and serving as an interface for the management console. In JMX the approach is very similar. DMTF does not define clearly the role of an agent, instead access to system resources is done through an *operation manager* that handles system resources with the help of specific adapters.

3.5.5 Connectors

Distributed components communicate with each other and with services provided by their environment. A common approach to abstract this communication is to use *connectors* [87]. They serve as medium and provide common interface for communication without the need of any specific information about the way communication is achieved. For example a component may communicate with a database using a common connector or driver Application Programming Interface (API) while the underlying mechanism selects the appropriate communication protocol implementation. This adds flexibility and simplicity to distributed component architectures. For example, in the case of a database connector, an application may be enabled to operate with a set of Database Management System (DBMS) according to the distribution available for the user. For example, in SOFA connectors are explicitly used as a mean of component communication.

3.5.6 Component Descriptors

When deploying a component a framework has to be able to recognize it and extract information about it in order to know how to deploy and configure it. *component descriptors* [15] contain meta-data which describe component properties, such as interface definitions, version, used packages, vendor information, dependencies. Component meta-data definition is an important aspect of software management because it gives a basis for deployment reasoning and dependency management. For example the OSGi framework uses bundle meta-data to manage the version of services and select appropriate packages to be used with a particular package. The J2EE heavily uses component meta-data to map remote and home interfaces, data sources, configuration and security realms. COM uses component version meta-data to link dynamically server parts and clients. A necessary requirement for properly functioning management is a well defined meta-data that supports the desired management functionality.

3.5.7 Component Repositories

A widely accepted approach for configuration and management of component-based systems is to use a common *component repository* [74] where component information is kept and optionally components are stored for the purpose of downloading prior to their deployment. The repositories keep meta-data such as component versions, formal names of components, vendor information and dependency data. Dependency data is used to achieve consistent state of the system where components are deployed. Usually repositories contain complete history of a system's evolution by keeping multiple versions of evolving components throughout the lifetime of the system. When a deployment tool needs to install a component, the dependency chains defines a set of components that need to be deployed or undeployed in appropriate sequence before the concrete component. Repository meta-data may contain special information about branches that represent milestones in system development as whole and thus simplifies dependency management and decreases the chance of wrongly specified dependencies which is a possible case when the number of interrelated packages is large. Examples for component repository are the OSGi Bundle Repository (OBR) [19] and the existing Linux package repositories repositories supporting variety of package management formats [12].

3.5.8 Aspect-Oriented Component Engineering

Complex software requires modularization and component based programming solves partially this problem by functionally separating the program into units called components. However, in a relatively complex systems components duplicate part of their operation code for access to framework facilities and services. The integration of this communication remains granulated and difficult to maintain when implementation of either components or frameworks changes. Aspect-oriented design [53] addresses this problem by providing encapsulation of crosscutting parts between software elements and component frameworks. This encapsulation is called aspect. Examples for aspects are *logging, security, persistence* and *configuration.* It is important to note that component and aspect design are complementing each other and not conflicting. Aspects fill the gap of common component facilities by modularizing it in aspects, while components are functionally separated by business logic. A relatively new trend in software development is *aspect-oriented component engineering* that has the goal to provide methods and frameworks that facilitate aspect-oriented approaches in component architectures. An example for implementation of component framework using aspect-oriented approach is the JBOSS J2EE implementation. The approach of aspect component support is an interesting opportunity for flexible run-time component management and is relevant to the subject of this thesis as a possible way of integration with the proposed management architecture.

3.5.9 Web Services

Communication between distributed components and frameworks often occurs across the borders of business domains. To support the interoperability in communication between software systems a bundle of standards known as *web services* was developed to utilize the established web standards and provide comfortable communication over the Internet. Web services support several aspects of system design most notably Remote Procedure Call (RPC) and Service Oriented Architectures (SOA). There is a set of specifications that forms the core of the *web services standards.* The communication protocol *Simple Object Access Protocol* (SOAP) defines the standard way of message transport between communicating peers. A Web Service Description Language (WSDL) is designed for specification of the web service bindings to specific protocols. The *Universal Description Discovery and Integration* (UDDI) [35] specification defines a standard way of registering and look-up of services in public or private service registries. Additionally, the WS-Security specification defines the usage of the Extensible Mark-up Language (XML) [126, 138] for Encryption in SOAP messages for secure peer communication. The web service approach for service access gains popularity in modern business applications and is considered to be a milestone of the efforts for global inter-operable communication. Modern component frameworks are able to provide access to component functionality through interfaces exported as web services. Systems implementing this architectural approach usually exhibit multiple layers of functional separation which requires further management effort, including demand for self-management [50]. The open nature of web-service-enabled systems provide new possibilities for communication, but at the same time introduce new challenges and requirements for component management. Among the issues concerning management of web service enabled components is SOAP and WSDL evolution and incomplete or improper implementations of the standards.

3.5.10 Log Adapters and Event Correlation

An old but simple and efficient practice for access and monitoring of system events is by creating logs of component activity. Logs are read and analyzed periodically or in the case of problem identification. The format of the log records and the log information in form of messages is of critical importance for debugging and problem handling. In distributed and heterogeneous environment correlation of events logged in different formats can be a complicated task. This is achieved with the help of *log adapters* the purpose of which is to monitor a target log base and transform its log format to a normalized system form for the management. Such framework that was designed to solve the problem of log normalization across distributed systems is the Common Base Event (CBE) [8] specification. It provides means for description of event flow, communicating components and levels of severity. Once there is a normalized log base with the help of the references to communicating components and the timestamps of the records, the sequence of calls leading the problem occurrence can be traversed and the problematic component located [30]. Sometimes log analysis is the only available method for system inspection and has to be taken into consideration when managing heterogeneous environments.

3.5.11 CORBA

The *Common Object Request Broker Architecture* (CORBA) is a standard developed by OMG with the similar to web services purpose to facilitate interoperability in communication between distributed applications. The communication protocol used for transport of calls and parameters for the purpose of RPC is the *Internet Inter-ORB Protocol* (IIOP) [65]. It uses compact binary message format opposed to SOAP which is based on XML. Similarly, objects access functionality of remote or local objects via interfaces which are defined by *Interface Description Language* (IDL). IDL serves as a source input for automated generation of stubs and skeletons for different programing languages. Objects use facilities of Object Request Broker (ORB) which automates the routines for marshaling and un-marshaling of calls and parameters to other objects. For the purpose of locating objects CORBA specifies naming services which provide yellow-pages functionality for object location. CORBA is a well developed middle-ware specification with a track of stable implementations, heavily used in critical applications for transactional purposes. Newer versions of the specification define a component model which describes components as well as interfaces. Similarly to IDL the Component Description Language (CDL) is used for component interfaces description. The J2EE specification defines CORBA's IIOP as standard transport protocol for RPC between distributed EJBs. Programing of standard Java applications that need to connect to remote CORBA applications are supported by an ORB implementation and IDL compiler integrated in the Standard Edition of Java2 (J2SE).

Because of its mature specification and lightweight communication protocol CORBA is used in this work as an interoperability framework for integration of management interfaces.

3.5.12 Policy-Based Management

Automation in system management in general is dependent on concrete decisions about the state of system at a given point in time. Changes in the state of the system trigger management loops with decisions and actions according to a predefined behavior called *management policy. Policy-based Management* [88] refers to an architectural style of automated system management with a certain accent on decision making specified by the system administrator in a way to keep the system in certain conditions. A policy may define a large variety of conditions and actions for different situations or states for concrete part of the system. This policy has to represent a valid set of constraints, corresponing to the characteristics of the managed system [68]. An example for a simple constraint-based policy is the specification of minimal and maximal values of system parameters and corresponding actions for the cases when these values exceed the specified limitations. *Policy languages* are developed to make the definition of concrete policy flexible and easier for system administrators. A common approach for definition of policies is to use *Event-Condition-Action* (ECA) rules which consist of three elements: a concrete event emitted by the managed environment, a condition that evaluates the event and relates it to other parameters, and an action that is fired depending on the condition output. This is a simple but very flexible approach of policy definition that can be implemented with a number of reasoning approaches including rule engines with forward chaining algorithms and ontology-based reasoners. The presented architecture includes a reasoning module that implements a simple ECA policy.

3.5.13 Pro-active Monitoring and Management

For the purpose of on-time problem discovery and failure prediction the management part of system has to be *pro-active.* Pro-active design involves autonomous management units that independently from the passive event-oriented management inspect the state of the system and use event history in order to foresee possible effect on system stability. SABER [2] and ROC apply such design for prediction of intrusion detection or error prediction respectively. The proposed architecture foresees similar functionality by active monitoring of external for system access point interfaces.

3.6 Summary

The techniques and projects described in this chapter exhibit different aspects of autonomic management: distributed deployment, automated monitoring, policy-based behavior. Certainly, each of them contributes to the common aim of reducing management effort by combining existing software and hardware management concepts and applying reasoning methods from the field of *Artificial Intelligence* (). However, in the current state of the art in this field there is no significant contribution which investigates the close relationship between existing and verified abstract organizational management models and concrete software architectures to address heterogeneity and software evolution.

This dissertation outlines an approach which fills this gap by exploiting basic organizational and communication principles adapted from an existing and already investigated management model (The Viable Systems Model). The design of the proposed architecture and the communication mechanics reflect these principles, thus contributing to a new way of component relationship mapping and definition of requirements for communication in evolving distributed systems.

Chapter 4

Overview of the Proposed Solution

This chapter gives an overview of the principles and the ideas which the proposed architecture includes in order to address the challenges in autonomic management of component-based software systems. An organizational meta-model represents the architectural constraints for encapsulation of software components and requirements for feedback loops implemented by autonomic management support. The meta-model allows for formal modeling of viable organization and communication management on the levels of component deployment and runtime operations. The autonomic management architecture consists of modules that facilitate monitoring component states, operation manager that allows inspection of distributed dependencies. The architecture is modeled following the guidelines of the meta-model and provides interfaces for control according to its requirements. A middle-ware establishes managed communication channels between components and frameworks to support the architectural requirements.

4.1 Feedback Control Loops in Autonomic Component Management

There are three feedback control loops that a software management system may implement to support autonomic behavior. The first loop implements *runtime control* of components that are already deployed and functioning. In this loop managers are monitoring their respective managed elements and react according to pre-defined management policy to support control of system functional and operational health and assist with on-time problem discovery, dynamic re-configuration and intelligent runtime tuning for optimization purposes.

The second loop is the *deployment* feedback loop operating at the moment when the component is introduced to the system and becomes a functional part of it. A control loop examines the component meta-data and information it can acquire in order to resolve dependencies and adapt the rest of the system before the actual

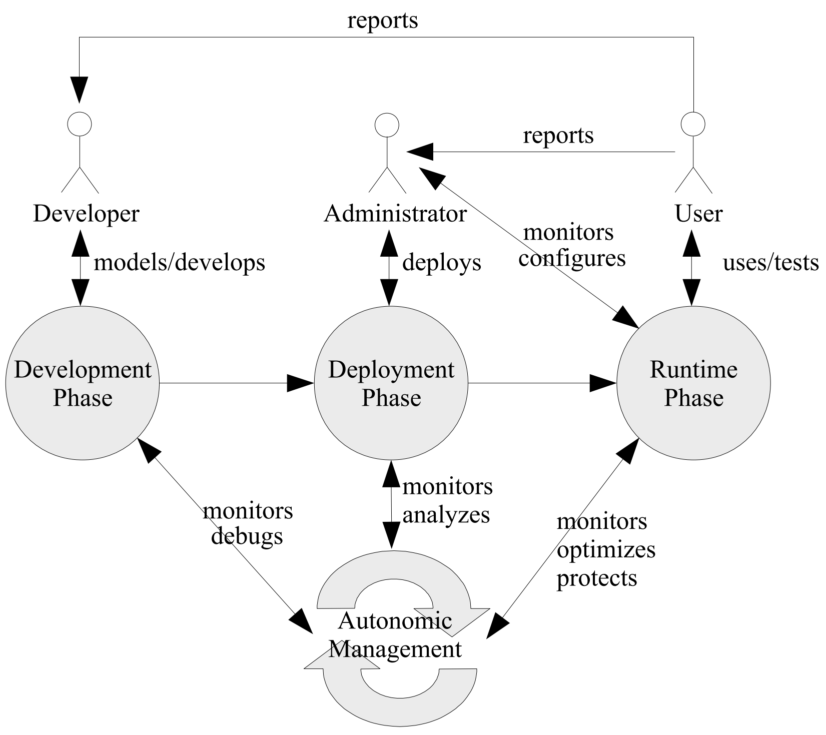


Figure 4.1: Autonomic Management Feedback Loops supporting the three phases of component life-cycle

functionality to become available in the system. This loop is important for dependency management and adjustment of communication channels to support the flow of data between components.

With the third feedback loop an autonomic system may support components in their development phase with relevant information about the runtime, conflicts and notification for possible problems in the overall system health.

Figure 4.1 illustrates the feedback loops operated by participating parties in component life-cycle phases of development, deployment and runtime operation. The development phase is influenced by the actions of developers, the deployment is governed by the system administrator while in the runtime phase users are interacting with the functionality of the component and report problems either to the administrative body or the development team. The autonomic management system supports the three phases of a software component and helps directly with the management facilities in debugging in development phase, dependency analysis in deployment phase and with optimization and security in the runtime phase. The picture does not illustrate details in the organization of the management system, it focuses only on its purpose of control and analysis support.

The proposed meta-model for component encapsulation and management facilitates autonomic support for the three phases by taking into account the role of the participators in component life-cycle and organizing the management levels accordingly. Every management level is responsible for particular sub-system with particular type of management set of communication channels and management policy.

4.2 Component Communication Management

Software components communicate using interfaces, events or messages and depending on the purpose of communication it may be with the management framework, with other components or with traditional (non-component) software entities, such as external functions and interfaces of classes outside the components. Figure 4.2 illustrates these aspect of communication. Communication with the framework is usually performed by the components in the case where they need resources or information about certain states of the system. Communication between components happens on two levels: semantic (functional) and sequential (executive). Semantic communication is related to the way client interacts functionally with the component on the level of their functional interfaces. The executive communication is usually handled with the help of the framework and is related to the concrete steps with which the actual information and variety transfer is achieved between the components. This includes marshaling/unmarshaling of data-types, data transfer protocol implementation and type mapping.

To achieve semantic communication the successful executive operation has to be present. Another important aspect of component communication is that the component itself is dependent on the underlying framework and essential resources, such as network and database access, are utilized through communication with the upper management levels. The vertical arrows coming out of a component (a) represent queries for resource or communication adapter to achieve the actual communication with other components or the component framework itself (b). That is why an essential point of heterogeneous management is the understanding of the hierarchical grouping and encapsulation of elements.

The approach to component communication management proposed in this dissertation introduces the notion of *Managed Communication Channel* (MCC). MCC represents both semantic and executive communication pathway that imply management unit to ensure proper communication between components the level on both levels. A MCC is an abstract representation of a medium for variety transfer, similar to the general notion of communication channel from the field of *Information Theory*, but simplified to accommodate management and analysis of communication between components which are based on management strategy.

Managed communication channels model communication dependencies between components and components and containers. The notion of component container has the broader meaning of a software entity that has degree of control over operation of elements inside it. For example, the Java Virtual Machine (JVM) is a container

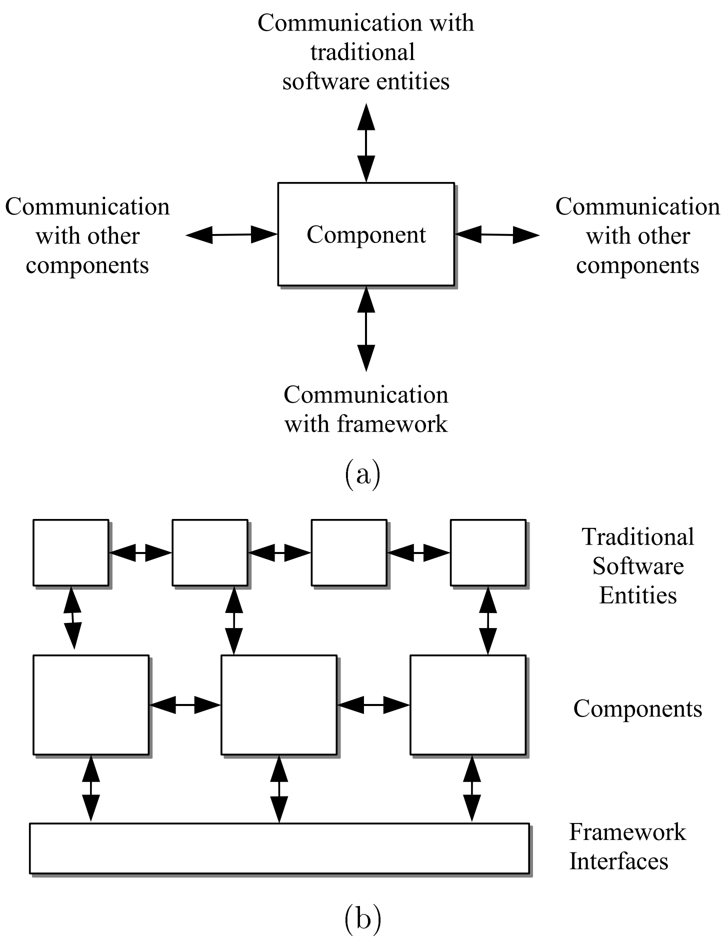

Communication with traditional software entities
Communication with other components
Component
Communication with other components
Communication with framework
(a)
Traditional Software Entities
Components
Framework Interfaces
(b)

Figure 4.2: Component Communication

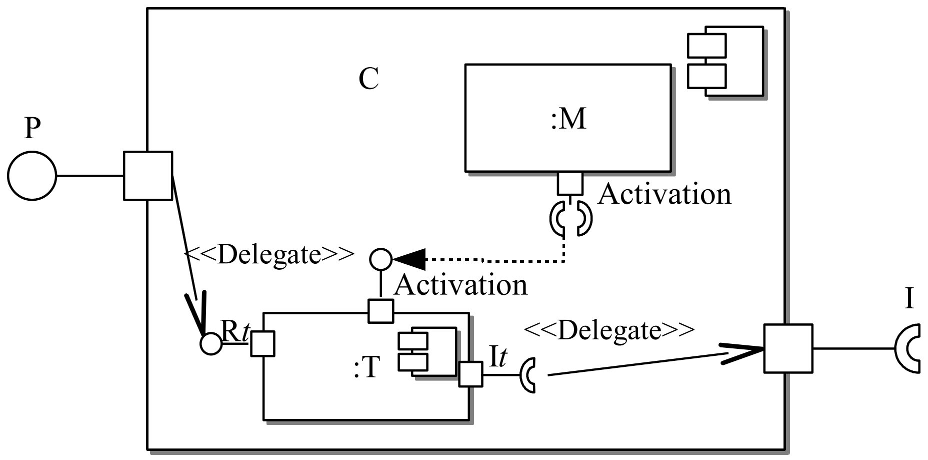


Figure 4.3: Components of Managed Communication Channel:
M - Channel Manager, T - Transducer, P - Source Communication Port, I - Destination Communication Interface

handling the operation Java programs hosted in it. A J2EE Application server is a component container that manages Enterprise Java Beans (EJB). On their turn, components that reside inside containers may also serve the purpose of management of other components. For example the J2EE application server is a component managed by the JVM, but at the same time container for EJB components. The location of a managed element in the hierarchy of containment determines the type of dependencies it may have, thus the type of communication data bound to the communication channel. There are two types of communication within a component system according to the hierarchy of component containment: component-component and container-component communication. In the deployment phase this kind of separation allows for monitoring of dependencies on the level of component deployment and in run-time component communication. The abstraction of MCC provides a flexible way of static and dynamic dependency inspection with the option for per-channel autonomous distributed management. Figure 4.3 shows the dependencies of elements that form a MCC.

A manager is responsible for the activation and setting up the delegates inside the channel that connects a components that demands communication with a component that provides communication interface - remote or local. The actions of the manager are determined by the variety of target and source interfaces in a way to satisfy the *law of requisite variety.* The actual transformation of variety is handled by a transducing unit, which may be interpreted with terms more commonly known in software engineering as adapter or bridge.

4.2.1 Meta-Model For Description of Management Organization

Another aspect of modeling component semantics is the role they have in the system containment and control hierarchy. The proposed organizational meta-model enables modeling of systems with characteristics like self-reference and recurring nested management and utilizes the concept of MCC.

The concrete management solution is able to describe its own architecture, thus make analysis of its own components and discover inconsistencies or communication difficulties to protect the system from management faults.

A key element in using the meta-model is the understanding of the two aspects of variety and knowledge management adaptation.

The first one is the *coupling of management entities* with other elements to form feedback control loops. Every management unit is assigned a responsibility role that corresponds to a level of the VSM model and is coupled to a managed element using managed communication channels.

The second aspect of adaptation is the *mapping of VSM's principles*, axioms and requirements to a management policy which represents the constraints defining "healthy" communication inside the management system.

The meta-model can be used to model management architectures and at the same time serve as augmenting meta-data for existing component-based systems.

4.2.2 Middle-ware for component communication management

The developed middle-ware for component communication management assists in setting up transparently communication channels and handles the provision of management information from and to other components of the management system. A component may need little or no change in its implementation code in order to be able to utilize the channel communication framework. The services that the middleware provides are made available through an interface for registration of component communication requests, adapter capabilities and available component services and remote interfaces.

The meta-model and the middle-ware for component communication management are the base on top of which managed applications can be built.

A management adaptation layer (Figure 4.4) provides a set of helper and base classes that can be sub-classed and re-used to facilitate fast adoption of the communication functionality. Such classes include activator classes, aspects and proxy classes that provide transparent registration of components and activation of the channel management middle-ware from components on the applications level. These classes include monitoring and management elements which serve respectively as sensors and effectors for the VSM Manager.

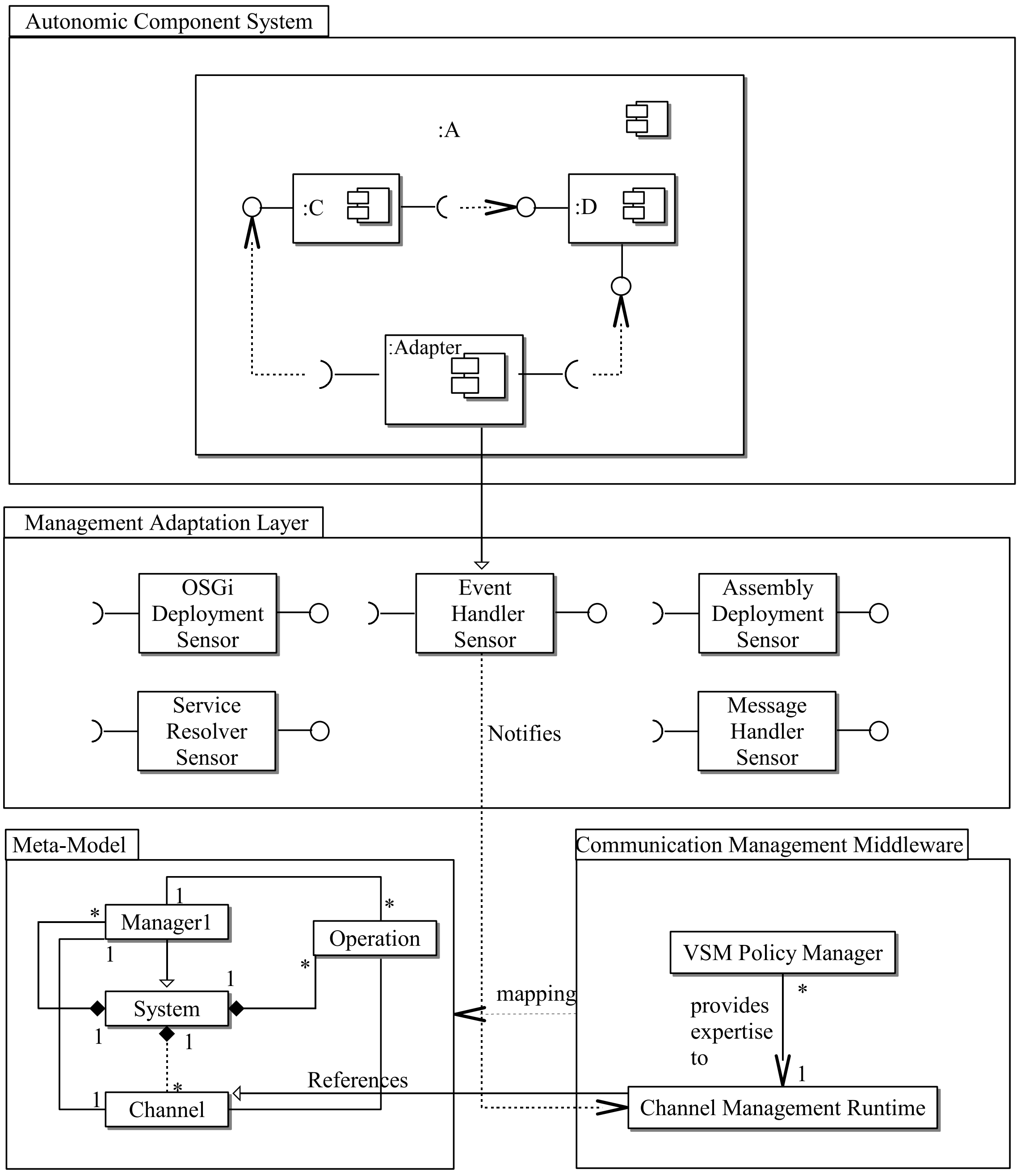


Figure 4.4: Autonomic Management Capabilities.
They are integrated as top-level elements on top of the cybernetically viable software architecture. A management adaptation layer provides management-ready abstract classes on basis of which autonomic components can be produced.

Additionally every component framework can provide management information with the help of a set of generic sensors and effectors implemented as native components for the platform. The specific choice of sensor integration depends on decisions related to the degree of coupling with the management system. Sub-classing base components implies that software development is in its initial phase. In-direct monitoring by generic sensor components is useful when applications do not allow additional modifications or any change in their design.

4.3 Effects on Autonomic Component Management Challenges

The proposed architecture, communication management middle-ware and the organizational meta-model address the challenges of component management described in Section 1.2.

4.3.1 Overhead in component deployment management

The problem of deployment management overhead is addressed by the management architecture which is designed to filter the reasons leading to those problems.

The architecture allows integration of deployment and communication models of various component systems within a single common model of interdependency and containment. The integration of additional models does not require explicit usage of the common management scheme as basis for development, rather they are indirectly associated to the meta-model and are modeled separately . This approach allows for technical separation and independent existence of the components but still relates them to the common domain of management using the same dependency model.

The communication middle-ware addresses the problem of *intra-model variety* by its capability to manage its own communication flow with the help of the communication management middle-ware. In this way a management system is able to monitor its own assets and discover inconsistencies that may lead to wrong operation or undesired influence on the system under control.

4.3.2 Management granularity

The problem of management granularity is addressed by the Communication Management Middle-ware which provides a common interface for monitoring of and access to management information on different levels of containment and distribution.

The middle-ware is capable to operate standards based and proprietary management interfaces by means of a standardized interfaces for integration of management adapters.

4.3.3 Parallel component evolution

The proposed architecture and communication middle-ware address the issues of parallel communication evolution and lack of coordination in software development by providing opportunity for multiple deployment of components and dynamic interface binding.

The possibility to deploy multiple versions of the same component and access their interfaces selectively provides the opportunity to test new or modified (optimized) component functionality inside the production environment and gradually extend the area of component's activity. Old component versions can be kept to provide stable functionality until the new version of the component full satisfies the tests or the requirements for healthy system operation. This approach indirectly helps overcoming of difficulties related to lack of coordination in development.

4.3.4 Deep component dependencies

The problem of deep component dependencies in a distributed environment is addressed by both the communication management middle-ware and the meta-model.

The meta-model is able to represent existing dependencies between components residing in separated containers as channels and associate them to concrete corresponding elements of the communication infrastructure supporting the distributed communication. Elements, such as communication adapters, connectors and interface resolving services have associated channel meta-data handled by the communication management middle-ware. Because the requirements verification is performed on the abstract level of channel, there is virtually no difference between local and remote communication dependencies. The variety of managed channels is determined by the observable set of communicating elements and the available meta-data for both components and container services.

4.3.5 Management complexity overhead

To address the problem of management complexity overhead, the architecture advises integration of management sensors and effectors as first-class manageable components deployed in the managed environment.

The phases of monitoring and control are separated logically from the operation of the managed element where the monitoring is achieved by either passively receiving notification by the sensors, actively probing the state without interrupting them, or using interception with blocking in the case of synchronized operations, such as channel activation or component deployment.

Component state synchronization is achieved by monitoring events using interception by either implicit wrapping of communication interfaces, or using aspect-oriented approach for interception of communication requests. The communication is blocked until actions from the management systems are undertaken, for example re-configuration or channel initialization. This approach may involve minor changes to the way component communicates but rather small to affect seriously its workflow.

4.4 Effects on Participating and Interested Parties

The proposed solution for autonomic management has effects on participating and interested parties operating in a managed by the proposed system environment.

As a consequence from the transparent communication management, the management approach affects the overall system availability throughout the process of system update and supports auto-configuration of application communication mechanisms on the *user side*. The approach may be applied for automated configuration of application dependencies without user intervention.

The architecture affects the experience of the *administrators* by reducing the overhead of management, especially in deployment and health monitoring, by using the common management model. The administrators will eventually tune only the parameters related to the strategy of component compatibility and communication settings. The rest of the management, such as adjusting concrete dependencies and selection of appropriate interfaces is automated by the system. Additionally the system is able to produce concrete problem reports to simplify the localization of problems relative to the role of the system administrator.

The system facilitates communication troubleshooting in the process of development. Software component *developers* are helped with reduced time to problem localization and discovery of communication inconsistencies related to initiation of communication, broken interfaces or network problems. The process of problem discovery can be escalated on the level of integrated development environment (IDE) by plugging monitoring functionality in the IDE itself to produce meaningful debugging messages.

Cooperating *service providers* will benefit from the dynamic communication management by allowing them to develop new versions of services independently from the consumers, deploy them separately and discover potential dependency problems in service composition.

The system management abilities for communication management provides advantage for manufacturers *and suppliers* for fast technology adoption and requirements satisfaction by extending of system functionality while the system retains reliable service consumption and provision.

4.5 Applying the architecture in Real World scenarios

To prove the usefulness of the approach, the architecture, the meta-model and the communication management middle-ware are applied to build a management system for two real world scenarios in separated domains of operation - *Smart Home* and *Product Development Support Systems.*

4.5.1 Smart Home

A typical Smart Home scenario involves at least three component technologies with interdependent components interacting in local and remote network environment: *Service Gateway* as a platform for integration of device communication and service access; *Service Back-end* for third party service integration, authentication and accounting, *User Interface* Control Software for user interaction and control remote and *Third Party Service Provision Software* as a third party service access point. The concrete scenario depicts OSGi as component technology on the service gateway, EJB on the back-end, .NET runtime on the user control device and SOAP as communication protocol for third party service access.

Architecture Elements: The management architecture defines sensors and effectors for the OSGi platform as first-class OSGi Bundles. In deployment phase management information is extracted with the help of OSGi-JMX adapter, which is also deployed as a bundle. EJB monitoring and management occurs through the integrated JMX architecture. On the .NET platform communication sensors are in the form of web service interface resolver wrappers.

Communication Management: Runtime channel initiation for component-to-component communication occurs with the help of service resolver wrapper, the purpose of which is to provide correct service implementation by using channel management middle-ware. The sensor bundle monitors deployment of OSGi bundles on the gateway and resolves the possible communication channels across the network with the Back-end server and the service provision. EJB and .NET communication channel establishment on the back-end, the user control device and the service provider is achieved with a common interface resolver wrapper communicating with the channel management middle-ware.

Meta-model bindings: The architecture is mapped on two levels of recursion. The OSGi Framework deployed on the gateway, the EJB server on the back-end-server and the .NET Framework on the remote control device are mapped to level "System 1" as containers. The channel management framework itself is mapped on

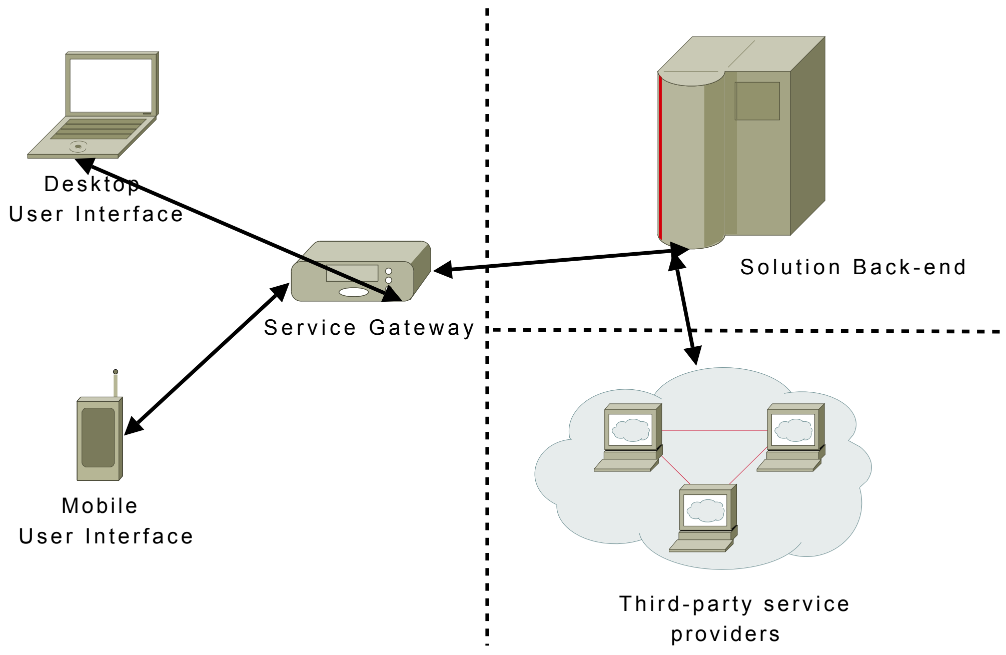


Figure 4.5: Elements of Smart Home environment.
User interface devices, service gateway are deployed in the home of the user running OSGi framework, solution back-end based on EJB and third-party service providers accessed remotely using Web Services or other RPC approach. Management of communication in such environment is critical for the quality of provided services, as well as for the end-user experience.

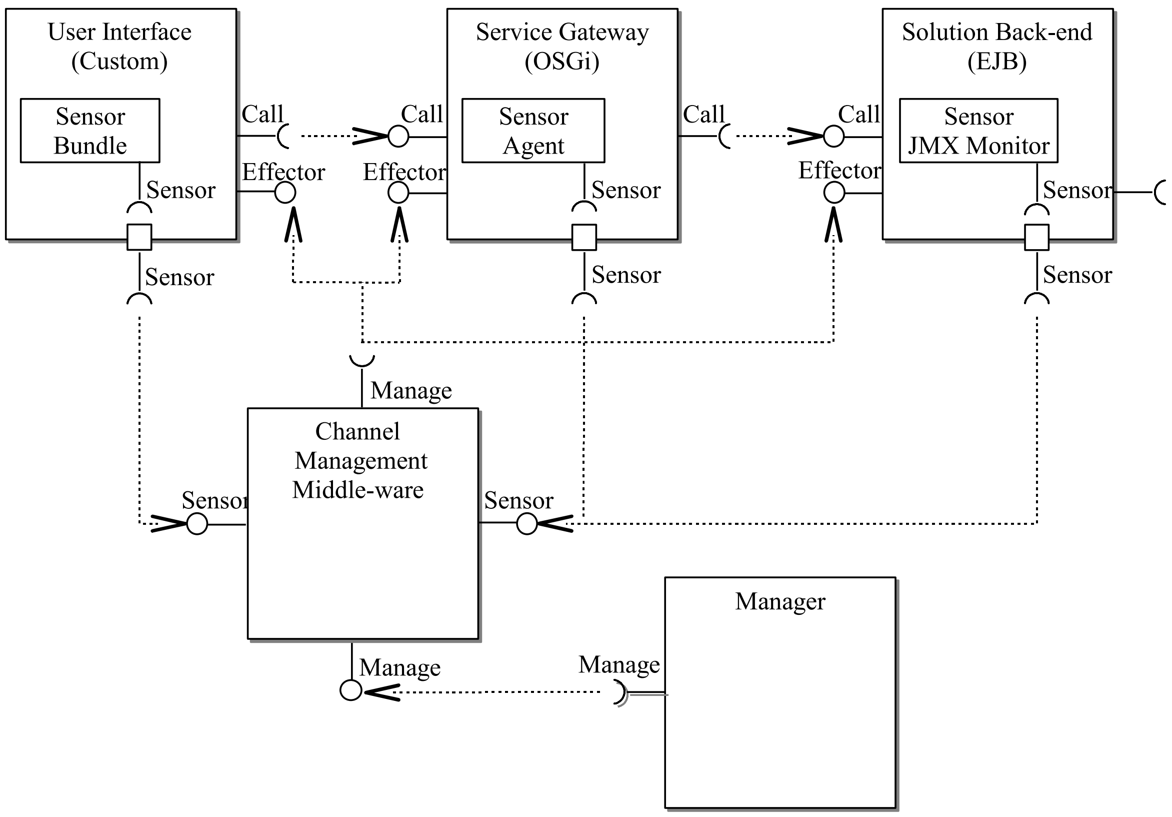


Figure 4.6: Communication management.
Channel management middle-ware handling the act of communication. Sensor components deliver management information for channel establishment. Concrete channels are managed by a channel manager. The figure notes a sample call interface to illustrate dependency on interface communication.

the level of “System 3” on the same level of recursion, while the bundles, beans and assemblies are mapped to management levels “System 1” as operation management elements on the next management level of recursion. The elements responsible for execution of EJB remote interfaces from the gateway, the web service proxies on the remote control .NET platform and the EJB back-end are mapped on the level of operation.

4.5.2 Product Development Support Systems

Product development support systems often involve a large variety of software technologies distributed across the environment serving the process of product development in its different phases - specification, design, prototyping and production. The specific scenario used as a demonstration of the proposed architecture references developed approach for integration of these four aspects with technologies for implementation of *rich client platforms (RCP), process knowledge-base back-end (KB), and knowledge-communication (KN) frameworks in RPD environment.* The rich client platform is a base on top of which design and document base applications are built. A clear representative of such framework is the *Eclipse Platform*. Its component design, based entirely on the OSGi component model determines the ability to extend the variety of the tools and provide fast integration of demanded functionality. The knowledge-base back-end provides services for knowledge-acquisition and integration and is implemented using an Active Semantic Network meta-model based on the EJB technology. The knowledge communication framework serves as middle-ware for automated retrieval from the knowledge-base, its aggregation and and coordination across the members of a design team according to the specified process of development or prototyping. It has been implemented with the help of a multi-agent framework, internally developed at IRIS [118] in the frame of Project SFB 374.

Architecture Elements: Similarly to the setup of the scenario in 4.5.1 the RCP integrate sensors in the form of first-class components residing inside the platform, while the EJB knowledge-base is managed over its JMX layer. Both platforms are dependent on the multi-agent system which leverages multicast architecture, thus allowing monitoring of messages of the communication between agents internal for the system. For this purpose a special agent connected to the same multicast group monitors the events between the elements. Sensors for provision of management information related to the external for the *Multi-Agent System* (MAS) components is tracked using agent client wrapper.

Communication Management: As in 4.5.1 the communication management middle-ware handles establishment and control of communication activities of the RCP components, the agents and the knowledge-base using the sensors and effectors

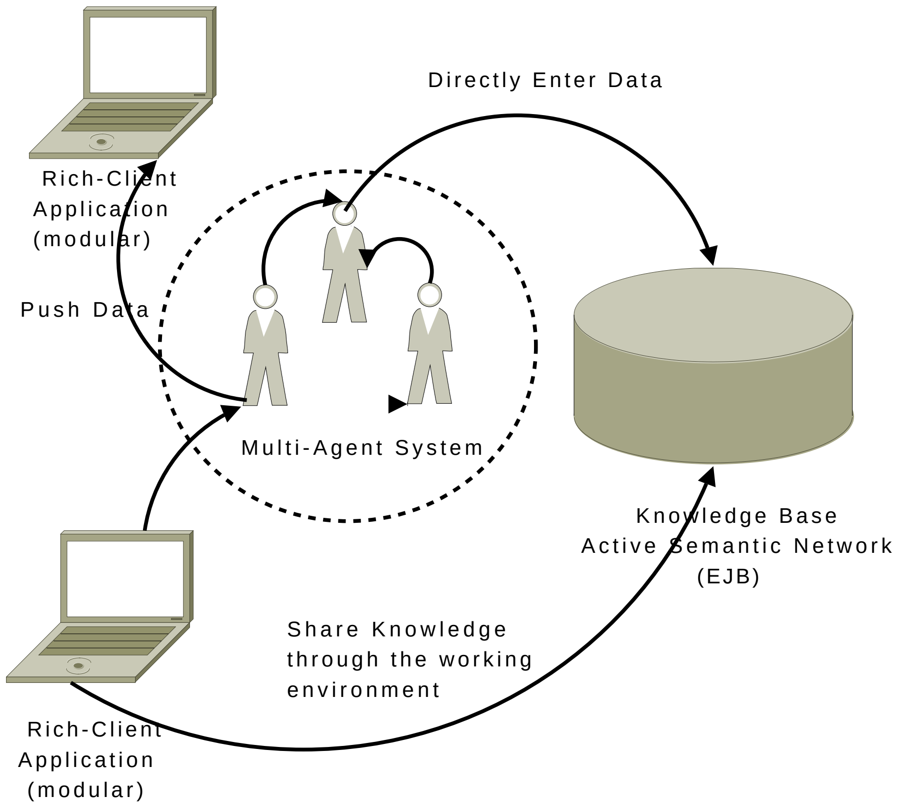


Figure 4.7: Architecture for Distributed Product Development with multi-user collaboration.
Multi-agent framework serves as communication middle-ware between the designers and between the designers and the knowledge-base. The knowledge-base can be accessed directly for input of information.

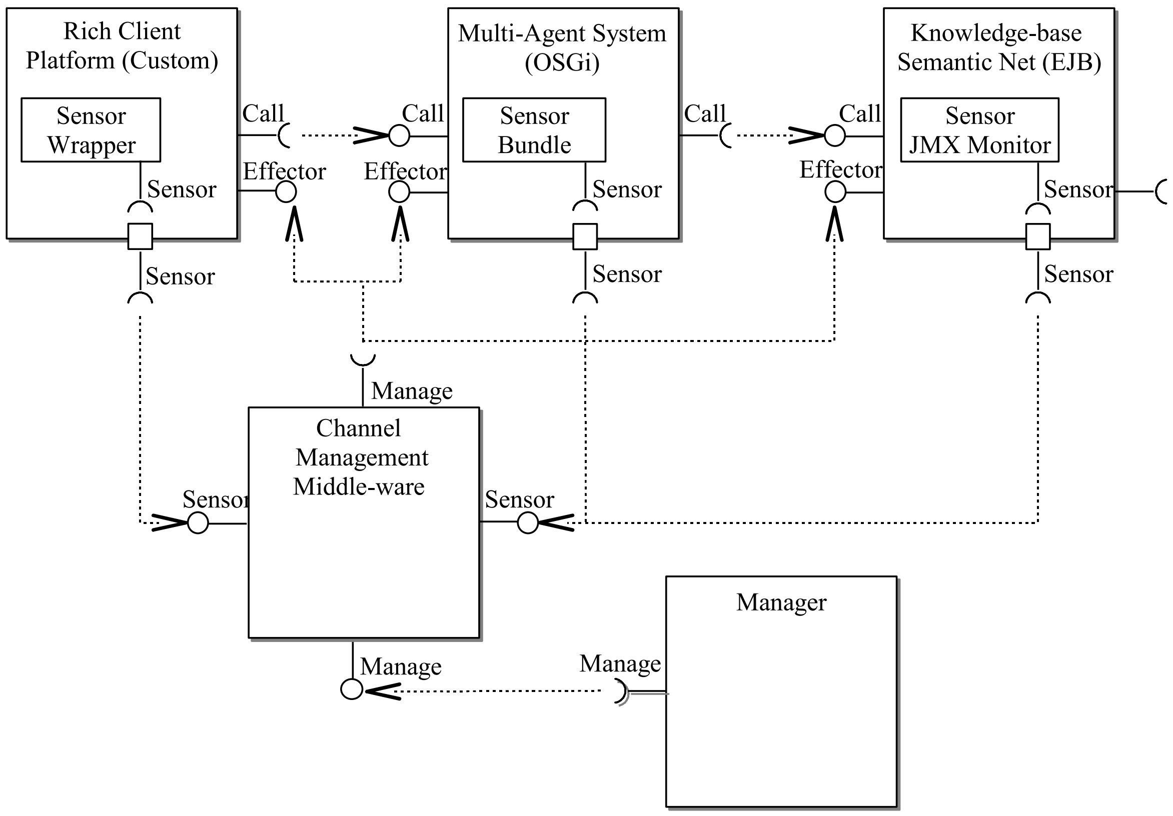


Figure 4.8: Communication management of a heterogeneous product development system with the help of channel management framework.
Sensors are first-class components delivering communication meta-data to the middle-ware for evaluation by a channel manager.

in the form of service invocation wrappers (OSGi), remote procedure call invocation proxies (EJB) and agent client wrappers (MAS).

Meta-model bindings: Bindings are very similar to those from 4.5.1. The RCP platform, the knowledge-base and the master agent from the MAS are associated with System 1, the management middle-ware with System 3, while the hosted components and the non-master agents belong to the next level of recursion and are associated with System 3. System 1 is represented by the service resolver, agent client and the remote interface invoker wrappers.

4.6 Prototype

To support the evidence of working management architecture for scenarios described in Section 4.5 A prototype was developed which implements the proposed meta-

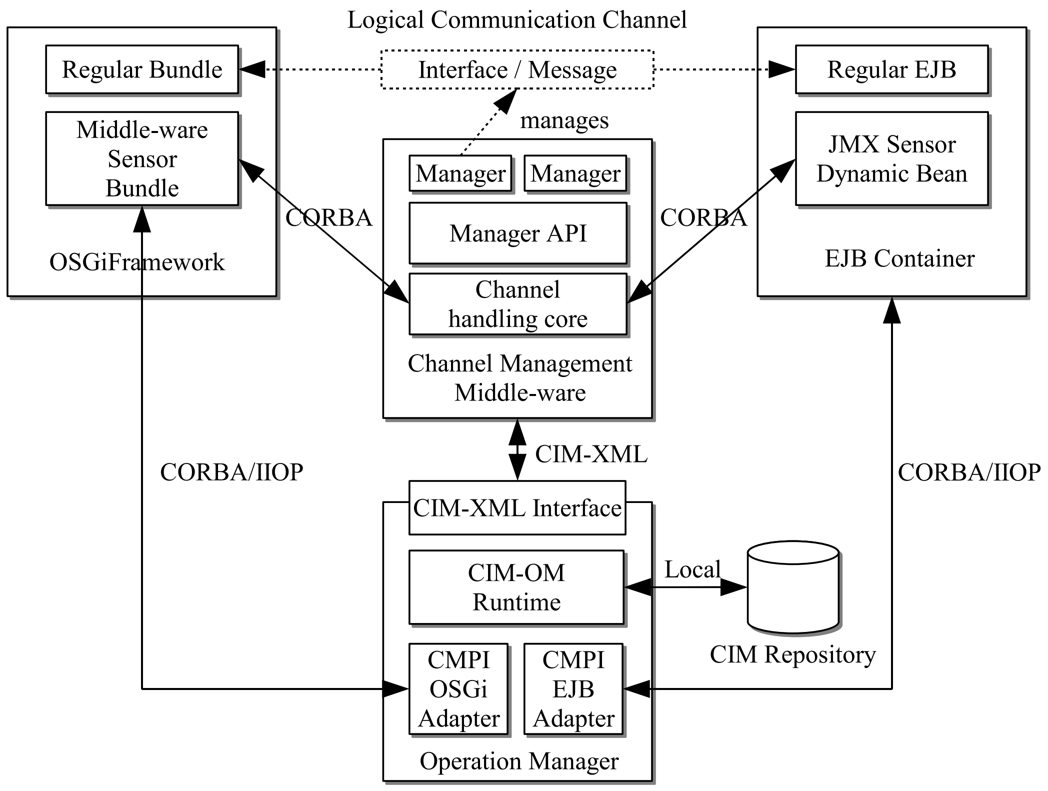


Figure 4.9: Implementation of the channel management middle-ware.
In this configuration an OSGi bundle and an EJB are communicating via communication channel of type Interface or Messages. The channel management middle-ware is monitoring communicative acts of both elements through integrated first-class sensor components. An Operation Manager handles model semantics and instantiation of management instances and creates mappings stored in Repository.

model, architecture and communication management middleware.

The objective was to develop a standards-based and platform-independent approach to IT assets modeling, monitoring and handling of management information in distributed environment. For this purpose CORBA was selected as a communication layer and WBEM [81] as management architecture with CIM as a modeling base. Figure 4.7 illustrates the concrete relations between the prototype elements below.

Meta-model: The developed meta-model is imported as a CIM extension-schema into the responsible for management CIM operation manager (CIM-OM) along with schema for the managed component frameworks, in the concrete case OSGi and EJB. The models are then mapped onto concrete management adapters that provide access to the managed instances - such as EJB Server, OSGi Framework and Bundles.

Management CMPI Providers: The role of management adapters in a WBEM management architecture is to extract necessary information from the environment and provide it to the CIM-OM within the constraints of the existing CIM model. For example, in the case of OSGi, the adapter creates CIM instances and initializes its property model with values corresponding to the information delivered from the OSGi Framework and the deployed bundles. Another task of the adapter is to create persistent associations with the meta-model. The associations are needed to reference the belonging of a component to a concrete element from the meta-model. Association can be traversed in both directions to express group relation or to serve as reflection path respectively. An adapter is needed for every managed component framework, however the access to the communication is unified by the CORBA communication layer.

Channel Management Middle-ware: The channel management middle-ware is a component delivering the essential framework interfaces for communication between sensors and VSM-adapted channel expertise. It is developed in Java with exported remote interfaces available through CORBA. This allowed platform-independent channel management with information delivered from applications programmed in languages supported by CORBA - Java, C++, Python, Perl, etc.

Management Expertise API: Evaluation of concrete situations of communication, component integration and management occurs in a component that implements a management strategy. The developed Expertise API provides a simple interface that allows integration of modules providing management expertise to provide flexibility in policy definition and distribution.

JMX-CORBA Adapter: JMX information is accessed traditionally through Java-specific interfaces, although the lower communication layer may involve the protocol used by CORBA - IIOP. To address this issue, a communication adapter has been developed which translates JMX interfaces to CORBA interfaces accessible through a remote CORBA Client. This enables a single interface for extraction of management information from both OSGi and EJB containers.

Re-used components: The work on the prototype has been supported by three components licensed with open source licenses that were needed to complete it.

One of them is the OSGi JMX- Agent developed within a project funded by Telefonica. It was needed for exporting internal OSGi framework functionality and interfaces as JMX manageable instances.

The second re-used component is the CIM-OM itself, an open source, enterprise-grade solution, successfully deployed in large IT environments. The third component is a CIM-Browser allowing to visualize and manipulate information in the CIM-OM.

Self-Management: The prototype itself implements a management architecture according to the adapted requirements of the VSM. It uses the meta-model to describe its own components and enforce management monitoring upon changes in the system the same way components are managed. In this way the management system is able to discover inconsistencies in its own components and issue a problem report, alarm and react by disabling its effective operations to prevent potential negative consequences from fault in management. The effector-components that are deployed as components in the managed component frameworks are mapped on level "Operation", the channel management middle-ware is related to "System 1" while the management expertise module (API) is on level "System 3". The corresponding channels connecting the elements - the CORBA interfacing, CMPI adapters and JMX adapter are monitored by the channel management middle-ware without interfering operations regarding channel adaptation. Instead, the middle-ware in this case only responds with either alarm, notification, or stop its own operation as a prevention measure.

Chapter 5

The Architecture for Autonomic Management of Distributed Software Components

This chapter describes the cybernetically viable architecture for autonomic management of heterogeneous component-based software systems. It can be used to add autonomic management support for systems composed of a variety of components deployed in a number of different component containers.

The viability aspect (survival in changing environment) of the architecture is achieved by adaptation of the Viable System Model - a cybernetic model for system organization, variety and knowledge communication designed to solve complexity in management of large-scale systems involving knowledge propagation and dynamic communication. A meta-model defines the way the system percieves managed data and its own organization and communication flow, thus remaining self-descriptive and self-monitoring. It can be used for the implementation of management instrumentation for autonomic computing, and applications that remain viable in changing IT environment.

The architecture consists of several elements that address the problems related to management of components, associated with the following key management capabilities:

- modeling of perception for the managed data - the management system has to be able to interpret management elements and its association with other components as a base for dependency reasoning and policy enforcement. This is facilitated by the meta-model providing means for mapping managed elements and relations.

- observable elements - managed elements have to be able to provide information about their status - facilitated by a management adaptation layer for state inspection, pro-active monitoring and management

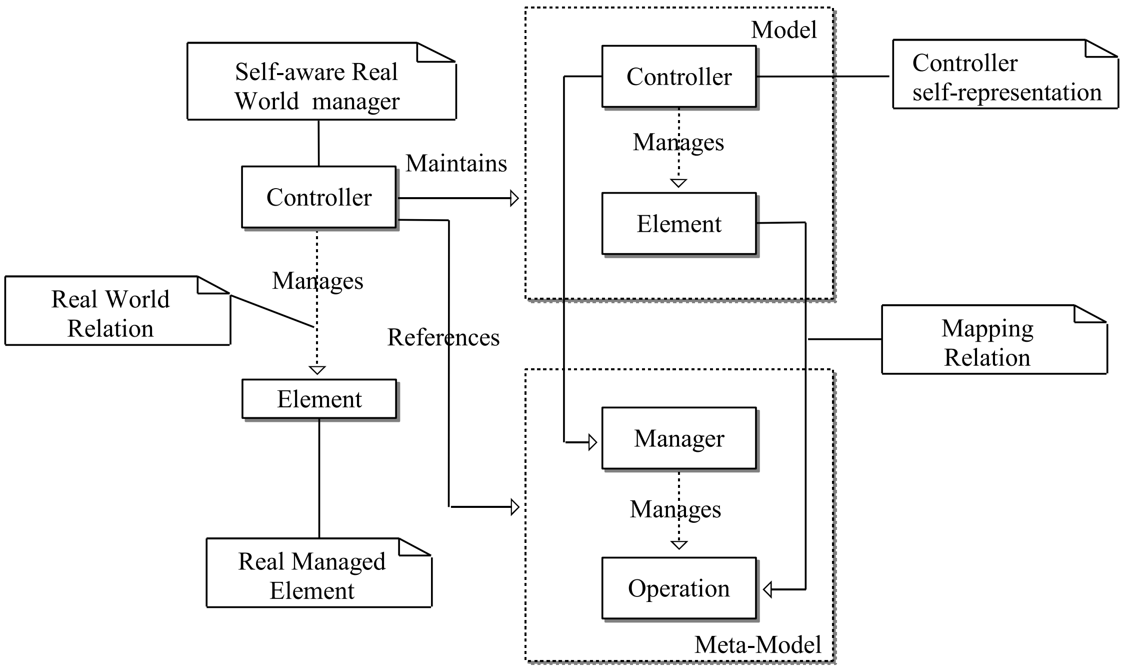


Figure 5.1: Management architecture with self-aware controller.

- assistance for adaptive communication - facilitated by the channel management middle-ware
- self-awareness and self-management - facilitated by the meta-model and reflection

5.1 Self-awareness

The meta-model and the abstract classes belonging to it serve as a descriptive mechanism for the management system to be able to deduce about the structure of the managed system and its own communication capabilities.

Figure 5.1 illustrates the principle of self-awareness using a meta-model. The management unit references a meta-model to maintain a model of the system. To be self-aware it represents itself in the model by mapping an instance of its management representation model to a meta-model along with the representation of the elements it manages. Because the management system architecture and the managed element share the same meta-model for representation of their structures, the requirements for system consistence has to cover both operational and management aspects of the system. The VSM includes principles related to communication flow between low level operations on the environment together with the distribution of variety and knowledge across the management sub-systems.

5.2 Meta-model Elements

The meta-model is shown on Figure 5.2 as a class diagram consisting of classes and associations between them separated visually in two distinctive parts and enabling description of both communication and structural organization in complex systems. On the left side are the elements representing the organizational aspect of the system while the right half of the diagram includes the classes for description of channel communication.

The organizational part reflects the VSM model by relating classes representing the management levels and component containment. The class *System* aggregates all other classes to represent a reduced version of the viable system model. The class *Operation Manager* is on its own a whole new system by deriving class System and inheriting all its characteristics, including the recursion. At the same time it is a first-level management unit. This implements the containment principle of VSM - viable systems consist of viable systems. Class *Manager* represents a higher level of management that is monitoring the actions of the system on the lower recursion level in the management system. The Operation class represents the elements that perform the acts that effect other elements or the environment in which the system resides.

The communication part on the right side of the picture includes classes that enable abstract representation of endpoints that communicate using communication channels. The channel class represents the communication medium assisting in transmitting of variety between elements by either filtering it, translating it or just passing it through if no of the former is applicable or needed.

The central class from which all other elements derive is the *Endpoint*. It represents a communicating element, practically every element in a software environment that has abilities to transmit information, invoke methods provided by other elements, to provide own methods, listen for events or trigger events. The Endpoint is associated with a channel representation to express its ability to communicate.

5.2.1 Variety

While the general notion of variety can be expressed quantitatively using a general unit for measurement, the bit, the heavy semantic meaning and the symbolic representation of the exchanged variety between elements do not provide ways to express it with a uniform, generally meaningful quantitative unit.

In this work the notion of variety embraces the variety of structural content between software elements. In this narrowed definition, the variety can be classified by two major factors - variation in different classes of information exchange, and variation in a single class - the main force of its existence being the evolution of the structure in time.

The software-engineering currently has several outlined patterns for variety in

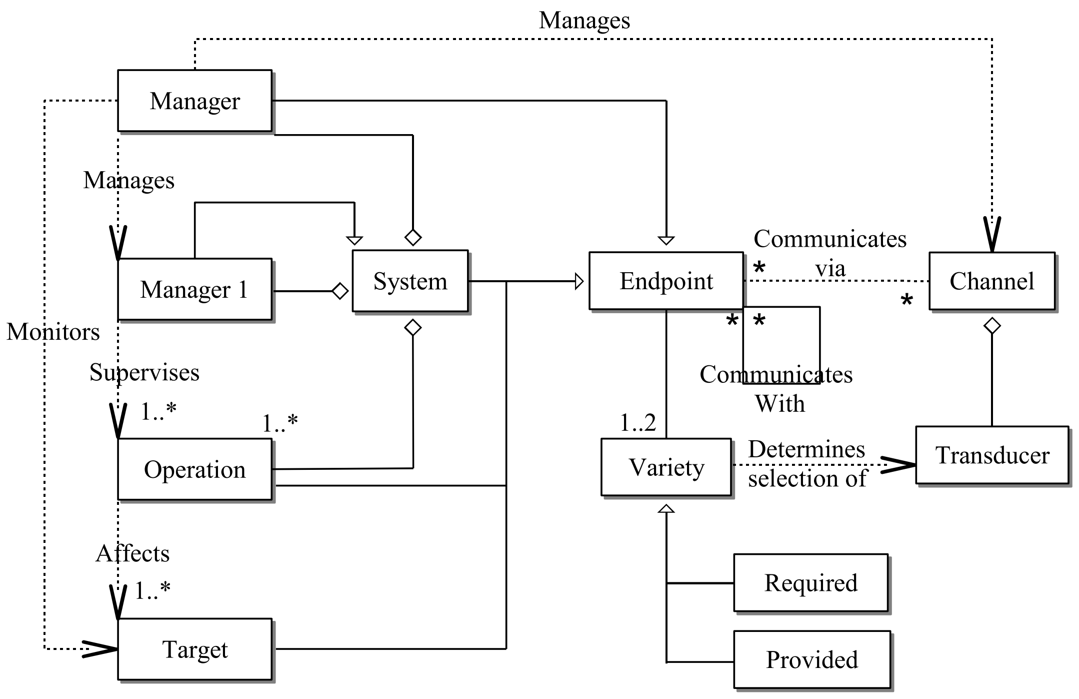


Figure 5.2: Meta-model of the cybernetically viable architecture.
The channel model on the right consists of endpoints providing variety and channels utilizing transducers for translation or filtering of the variety between the communication endpoints. The VSM-based organizational model is presented on the left side by levels of nested management systems with a terminating Operations unit directly affecting the target of control.

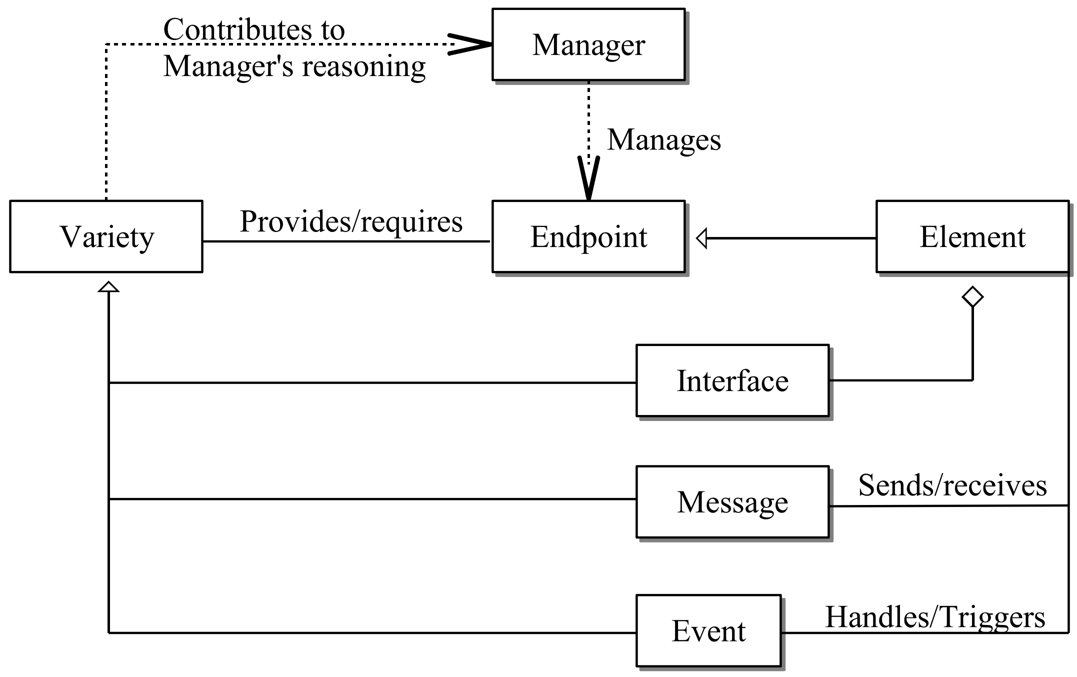


Figure 5.3: Abstracting from a concrete variety exchange method.
The manager reasons on the basis of the common notion of variety while the endpoint represents the actual software element initiating the communicative act.

software element communication - messages, events and interfaces. All of them care the common meaning of communication ability, but are associated with different dialects of communication organization. The notion of variety used within the adapted VSM model unifies them and abstracts the management subsystem from details related to the concrete way of its perception and processing.

Figure 5.3 illustrates a simplified management approach using variety abstraction. The element and its three styles of communication are mapped on the metamodel of the manager, respectively to the *Variety* class and the *Endpoint*. The *Manager* operates on the abstract level and when necessary, refers to the mappings to perform actual management operations.

5.2.2 Transducer

Adaptivity is a key requirement for a system to survive changes, thus to be viable. An unique aspect of adaptivity is the ability to communicate, regardless of the changing circumstances. Only then an intelligent management system is able to perceive and react accordingly. The VSM requirements associated with communication flow define a mandatory presence of capacity or ability to transmit variety between its constituents. To achieve this a communication channel has to have *Requisite Va-*

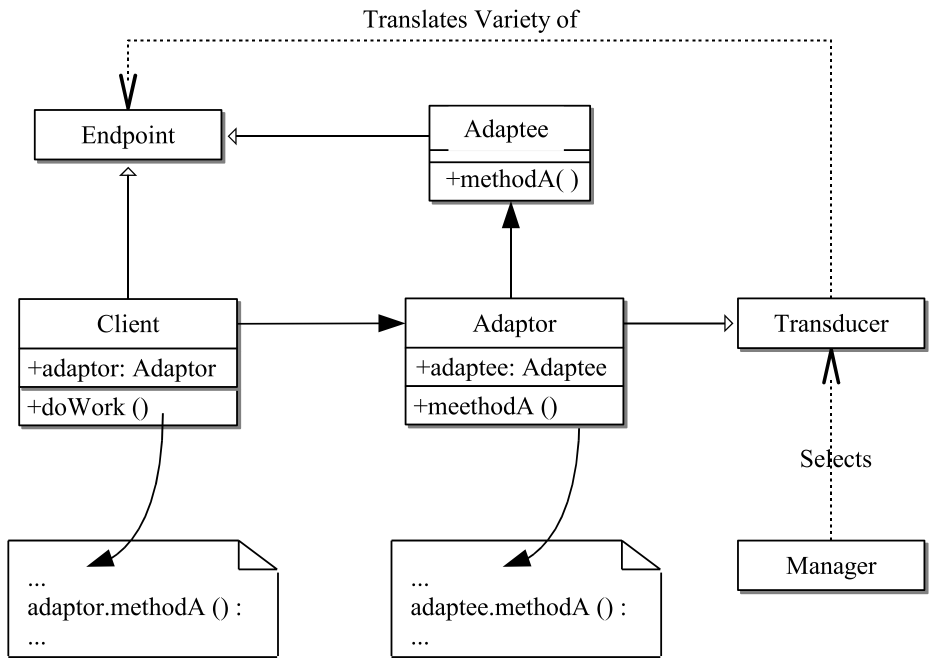


Figure 5.4: Representing interface adapter as a transducer selected by a manager.

riety regardless of the type of *transduction* which the exchanged variety may need. Transducers are cybernetic structures that act as variety adapters. Given an input of state values for a set of variables, the transducer produces output according to a state transition table.

The notion of transducer is applied to the meta-model and the architecture under the limitations of the variety definition for software element communication.

In the case of interface adaptation the existing and commonly used pattern *Object Adapter* can be treated as transducer. The added value of this mapping is the capability for a dedicated manager to select a proper adapter according the requirement for Requisite Variety in aspects interface differentiation and interface evolution. The latter is often neglected as a necessity for consideration of system design although especially in heterogeneous and loosely coupled systems it may have a considerable impact on system stability. Among the other commonly used design patterns that can benefit the abstraction of transducer are the *Mediator* and the *Bridge* patterns.

In a scenario of remote event handling it is very important to be aware of the variety which a component is allowed or is able to accept from an open network environment. The forces of parallel component evolution and lack of development coordination in a distributed system may cause system malfunction, and in the worse case security-related compromising or exploit. The common approach to remote eventing in distributes system is the *Reactor* behavioral pattern. The notion of transducer in this case can be applied to assist in a refined selection of event handlers by the initiation dispatcher which invokes the handler callbacks for event processing. This scenario is illustrated on Figure 5.5. Similarly to the case of interface adaptation, a regulated dispatching of events among multiple versions of components is an important requirement for evolving and adaptive system. The principles discussed here for adaptation are the foundation and the motivation to abstract the way two components communicate following the cybernetic requirements for communication by the introduction of a "virtual" managed communication medium - the *Managed Communication Channel (MCC).*

5.2.3 Channel

Communication channels represent medium for exchange of variety between communication software elements, the same way VSM defines communication channels for exchange of variety between elements of a VSM-modeled system.

As shown by 5.2.1 and 5.2.2 adding management of variety creates the opportunity to control the flow of the events, messages and execution of methods provided by interfaces.

The main reason for the introduction of MCC as a standalone element of the meta-model is to facilitate management of the system communication flow and to allow enforcement of VSM requirements by a dedicated manager. The meta-model defines MCC as a way of communication for any endpoint that provides information for the provided or required by it variety as a way to define dependencies. In the custom case of no variety translation or no component evolution, channels can be treated as dependencies in the accepted by software engineering general meaning.

In the general case where variety is transformed from one endpoint to the other, a channel has a heavier semantic value than a dependency for its ability to be adapted while keeping the semantic dependency between elements intact in both run-time phase and deployment phase. In development phase, channels may become accessible for inspection by developer's integrated environment which using the information may assist in code-generation or conflict resolution.

Examples for concrete application of MCC as a concept for communication management in software system are illustrated on Figure 5.7. The main purpose of the manager is to regulate the variety communication and satisfy the requirement for requisite variety of channels and transducers, thus keeping the capabilities of the channel in tact to transfer correctly variety from the first endpoint to the second. Figure 5.7 is an illustration of the abstract meaning of the channel as a medium for variety transformation. Transformation happens on the level of user interface - part of the system state is represented and "transmitted to the user in understandable for him way. A potential application for this specific scenario is development of autonomic user interfaces providing flexible composition of appearance and behavior according to user's role and the available for him information.

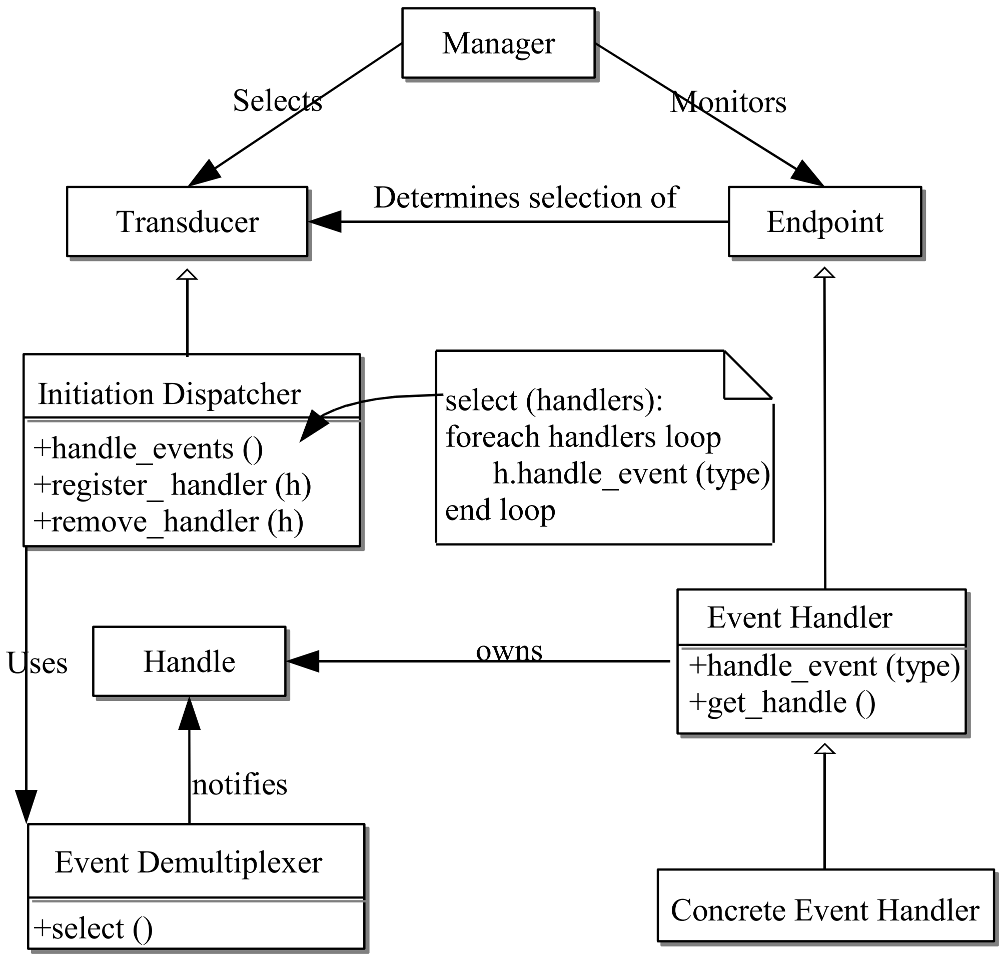


Figure 5.5: Distributed event handling with supervision by a manager.

Manager is monitoring supporting types by event handlers (endpoint variety) and selecting corresponding transducer to translate events (variety) coming from the event multiplexer. In the absence of transducer and its management the system lacks the needed adaptive mechanism throughout its evolution.

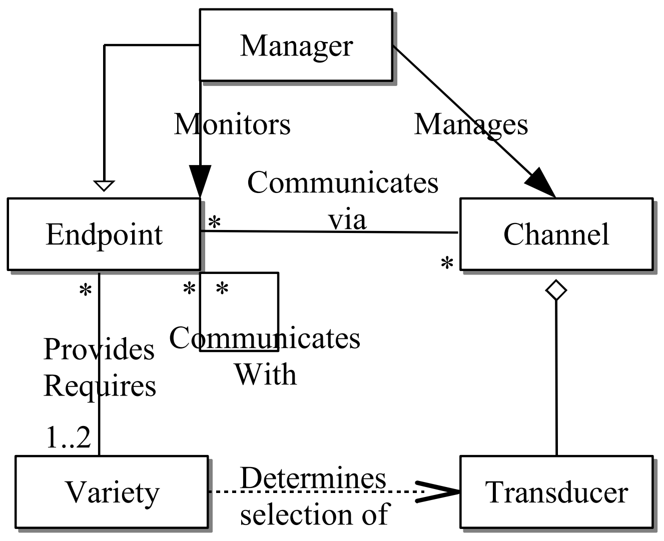


Figure 5.6: The concept of MCC.
Channels associate endpoints and translate variety using selected by managers transducers. Managers are responsible for channel creation and initiation.

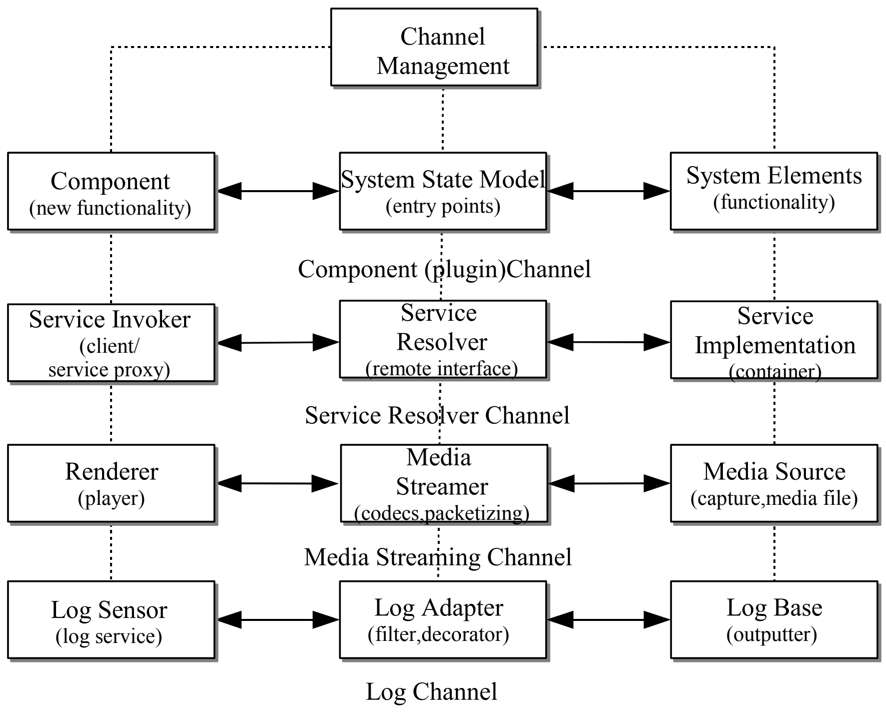

Channel Management
Component (new functionality)
System State Model (entry points)
System Elements (functionality)
Component (plugin)Channel
Service Invoker (client/ service proxy)
Service Resolver (remote interface)
Service Implementation (container)
Service Resolver Channel
Renderer (player)
Media Streamer (codecs,packetizing)
Media Source (capture,media file)
Media Streaming Channel
Log Sensor (log service)
Log Adapter (filter,decorator)
Log Base (outputter)
Log Channel

Figure 5.7: Application of the MCC concept in software engineering.
The endpoints exchange variety with the help of transducer (between them). The correspondent transducer fulfilling the requisite variety requirement is selected by the manager which also takes care of channel composition and initialization.

5.2.4 System

On the left hand side of the meta-model illustrated on Figure 5.2 is expressed the adaptation of VSM in organizational aspect. The key characteristics that determine the value of the model are: complexity management through recursion, separation of management roles, and requirements for variety distribution among the management units.

Assignment of management roles is an important element in the design of a system that allows for concrete policy propagation and task distribution. The management organizational model proposed here outlines four basic elements that describe management roles, policy distribution and recursive management-subsystems - *Manager*, *Operation Manager*, *Operation* and *Target*. These elements are all associated to a single class *System* which symbolizes belongingness to a distinctive unit of organization. *System* is an abstract class that is never used directly to describe a real system, rather it serves as a base class for units containing self-referring systems of the same type.

5.2.5 Operation Manager

The class of management elements associated with *Operation Manager* are recursive, self-referring systems containing the same set of management roles, management organization and communication capabilities as the system to which they belong and to which the same regulations and principles are applied. This approach of containment is the foundation for building viable systems, as specified by the *Viable System Model*. Example in software architectures are visible on the level of operating systems, virtual machines, component containers and components, resembling the separation of roles and tasks on every level of recursion respectively as process management, program management, component management and service management.

Figure 5.8 illustrates an example of organization of administrative support of an IT system. The system shows a recursive containment by encapsulating components into containers, such as applications, which on their turn are resided in servers, which are connected through a network. The task for management is assigned to a single unit "Administration". The lower levels of containment rely on the support of the upper management level.

This kind of mapping of elements to representations of their containment and applying the VSM management requirements shows the weak spots in an IT system - in this example, the upper management level (administration) has to be able to handle a higher amount of variety in order to manage the subsystems bellow it, thus missing Manager3 and Manager4 system elements contributes to a higher probability for management error.

This conclusion corresponds directly with the key objective of Autonomic Com-

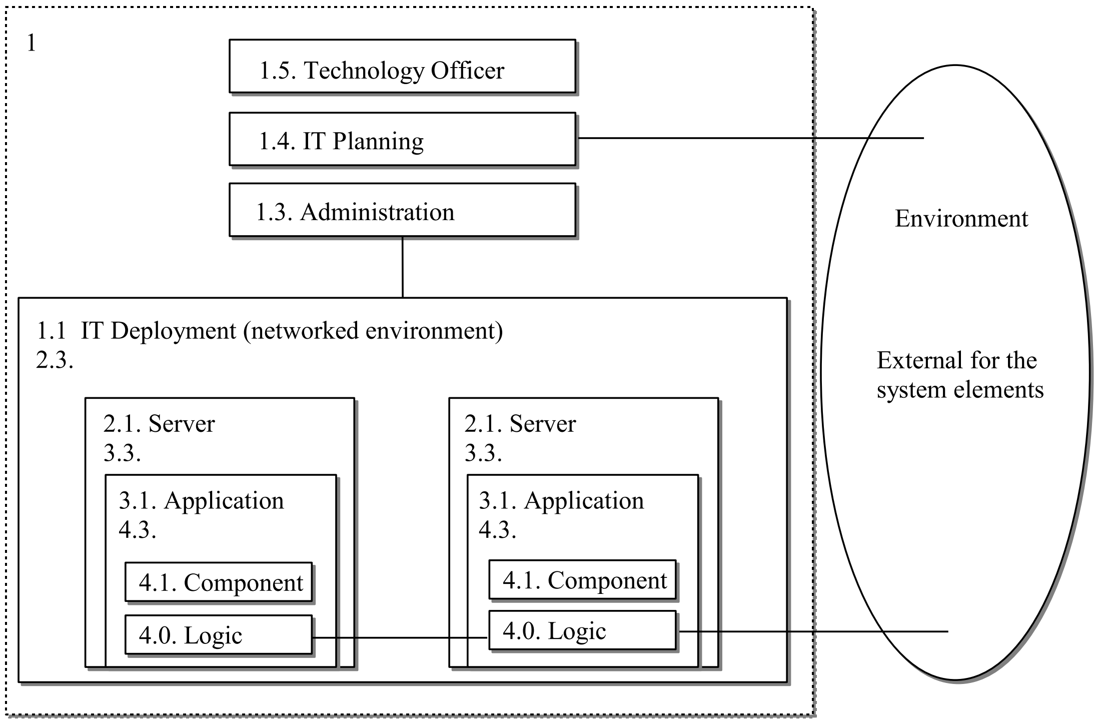


Figure 5.8: Management of an IT system with networked servers and communicating components.

Elements are indexed with major digit as level of containment, minor as level of management. An IT system managed by the Administration unit on level 1.3. Levels 2, 3 and 4 are entirely dependent on the higher level of management. According to the principles of VSM, higher levels of management absorb a smaller set of variety, which should be filtered across the low-level management. This example illustrates the need of additional management support on levels 2, 3 and 4 to reduce amount of handled variety and the probability of error.

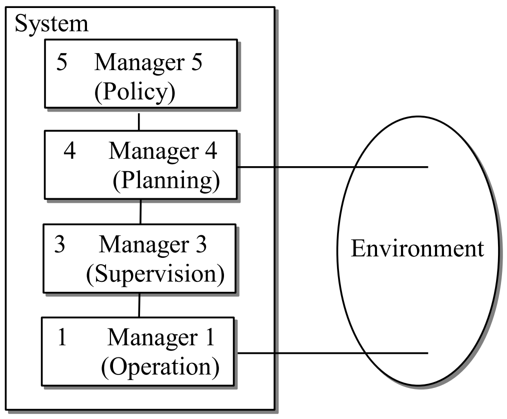


Figure 5.9: Types of Manager Systems.
Manager 4 is responsible for planning and prediction and requires access to the environment, monitoring it pro-actively as a requirement for viability in changing environment.

puting to reduce human intervention with automated management and set of configuration parameters on the level of applications and components.

5.2.6 Manager

The manager class in the meta-model represents a set of elements that are responsible for the distribution of tasks, planning and supervision of operation management units (Figure 5.9).

These are visible on Figure 5.8 as elements indexed with 1.5, 1.4 and 1.3 respectively, corresponding to VSM elements System 5 (Policy, Identity), System 4 (prediction,planning) and System 3 (supervision,resource management). This type of management elements may require access for monitoring of the environment in which the system operates, especially concerning management function for prediction and planning and are an important and necessary element in systems with autonomic management support.

5.2.7 Operation

Entities that have direct influence on the internal or external for the system environment state are associated with the *Operation* class of elements. Their role is solely to execute procedures scheduled and propagated by the management units and to react to events, messages and communication acts from the elements of the environment in which they reside. The VSM model suggests that the amount of variety which *Operation* elements absorb is considerably larger than the variety needed to be communicated between the management units, decreasing in direction Manager5, thus the organization of operations, their behavior and the way they perceives the environment is essential for the overall system stability.

The meta-model describes Operation elements as endpoints communicating with their environment using channels, the same way other elements in the environment communicate, however, their communication is in direct relation to the state of the system and affects the functionality of the system, differing from the management communication which primarily affects the aspect of configuration, adaptation and incident management.

5.2.8 Target

The *Target* class represents all elements that are not associated with the managed system but influence it by communicating with it and introducing variety in form of messages, local and remote events or by means of established communication protocol. The difference between Target and Endpoint is the ability of the Target elements to be visible and managed by managers of the systems to which they belong or communicate with, while Endpoint represents any element that may provide to or require variety from the environment.

5.3 Management Adaptation Elements

A vital requirement for a managed system is the ability of managers to observe the state of the managed components and eventually modify it. While certain amount of information may be acquired by using the standard management capabilities of the existing software frameworks, it may be not necessarily enough to achieve desired communication monitoring and control for communication assistance and adaptation.

The architecture defines a package of elements associated with a layer for acquisition of management data through both standard means and explicitly defined software elements used in components communication activity. There are two major functions that a management adaptation element may perform - monitoring state of an element and influencing it. Monitoring elements are perceiving state of manged elements and are known in the autonomic computing literature as *sensors*. Elements that influence the state of an element are known as *effectors* for their performing operations that effectively change the state of the management element. The sensors and effectors represent the intersection points between a system environment and management units accessing it. The aim for communication assistance and management requires not only information regarding software component state, but also sensing of information related to channel establishment, such as mediating services provided by either the hosting component frameworks.

The communication management middle-ware uses both, sensors and effectors,

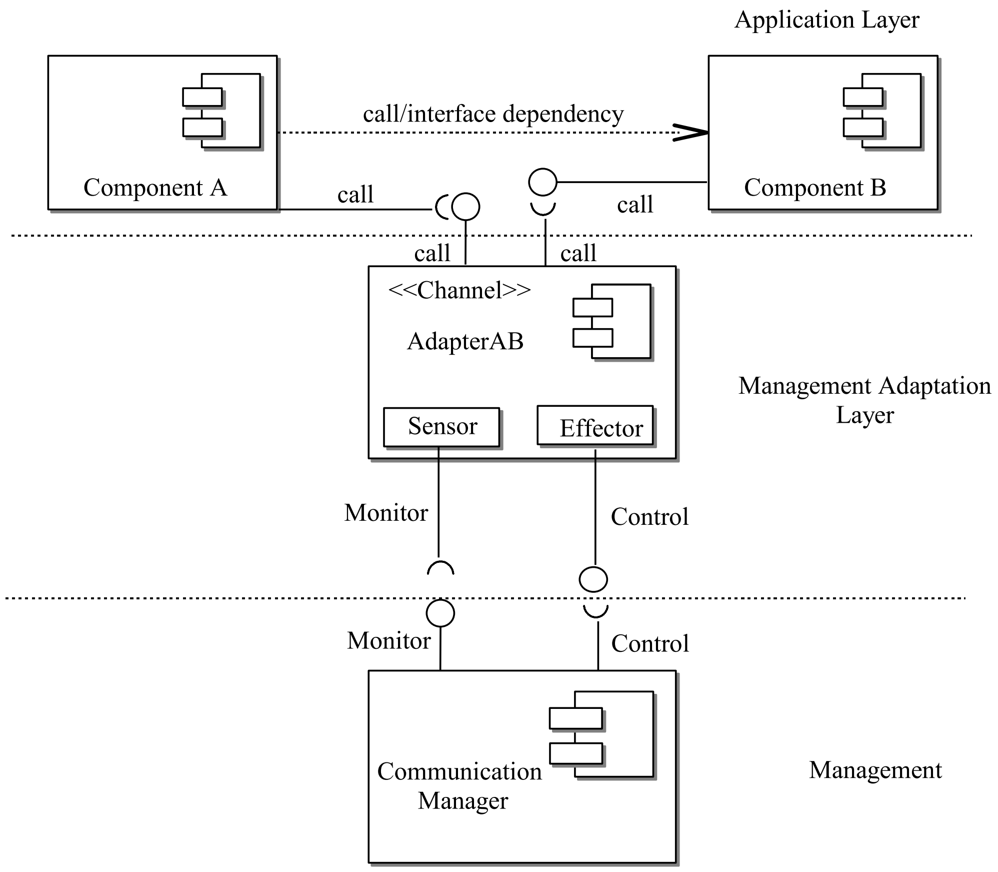


Figure 5.10: Management adaptation layer.

It provides management information to the manager through sensors and effectors embedded in base classes. The channel management framework on the management level evaluates the transmitted variety and sets up element's parameters through its effectors.

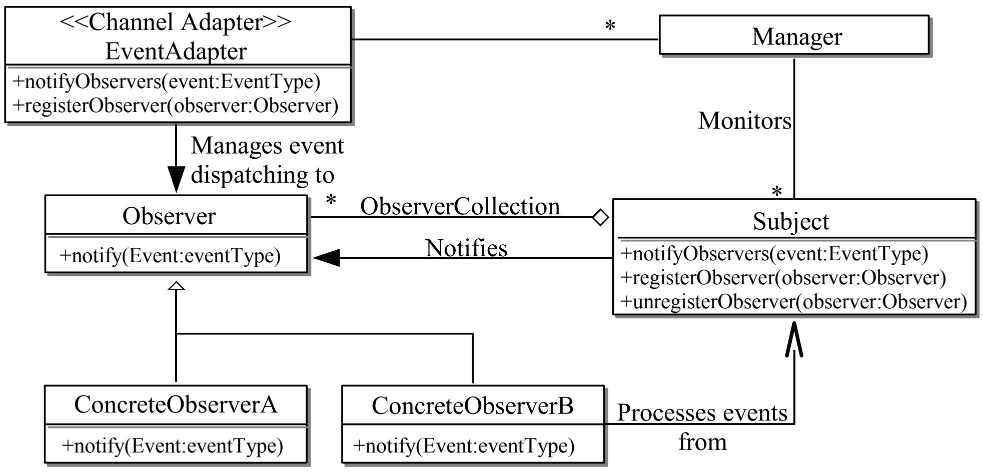


Figure 5.11: Event channel adapters.
They mediate information about event handler registration and event delivery to event adapter selecting a concrete observer implementation.

for respectively evaluation of communication dependencies and setting up parameters relevant to communication channel establishment or behavior. To achieve this the package includes abstract components incorporating necessary sensors and effectors to facilitate transparent communication establishment in derivative elements. These are called Channel Management Adapters and can be distinguished on the example diagrams by the stereotype <<Channel Adapter>> Figure 5.10 illustrates management of communication between two elements. A detailed description of the structure of a channel adapter is given in Chapter 6.

5.3.1 Management Channel Adapters

Depending on the type of information the management adapters provide there is variation in the way it is perceived and transmitted to the management level. According to the constrained definition of variety supported by the management channels channel adapters are responsible for transmitting of information about events, messages and interfaces provided by interfaces belonging to communicating elements.

5.3.1.1 Eventing Channel Adapters

Event-based component communication is a well developed technique in software engineering. A commonly used pattern that implements event communication is the *Observer* pattern, where participating observers are notified about events from subjects by iteration on handling interfaces belonging to registered with the subject observers.

The task of an event adapter is to notify the channel middle-ware about the act of sending event from the subject to the observer in order to verify the ability of the observer to handle the events which have been sent. This can be achieved in two different phases of the interaction between the participators - the first being the registration of the observer with the subject, where the observer declares its intention to receive events, and secondly in the phase of the actual event broadcast from the side of the observer. Figure 5.11 illustrates the actual usage of the sensors with the observer pattern. This application of channel-based management information delivery is particularly useful in scenarios with remote eventing, where the events are coupling architecturally separated and parallel evolving software elements in a networked environment and the change in event semantic or type is important for the operation of the system as whole.

5.3.1.2 Messaging Channel Adapters

Loosely coupled components use messages to exchange information or trigger desired behavior with the preposition that the components do not necessarily know about their existence before they have been deployed in production environment. Similarly to the event-based communication, message-driven sub-components react to incoming messages which can be interpreted or internally converted to events, however a message may provide little or no information about the context of its usage by the sending element, on the opposite of event-based behavior where observer and subject are linked prior to message exchange. This kind of communication is used in stateless components or services that have no implicit relation to the system they react to. Message adapters are elements that intercept intention for message transmitting on the side of the sender or intercept message receiving on the side of the receiver to facilitate channel management for exchange of messages. A direct effect from this particular channel management is the ability to detect intrusion detection or denial of service attacks.

The effects of parallel evolution in loosely coupled messaging systems may result to undefined behavior by both communicating parties and management of their communication is an important factor for the successful operation of the systems including the components. Figure 5.12 illustrates management of message sending and receiving in aspects assistance in delivery and adaptation and enabling filtering of incoming for the receiver messages.

5.3.1.3 Interface Access Channel Adapters

Object oriented design defines the concept of object as entities that communicate by sending messages. The concrete implementation of the message exchange mechanism highly varies depending on the programing language and the design of the object support in program runtime. Widely adopted object-oriented languages such as C++ and Java implement communication using methods defined in so called

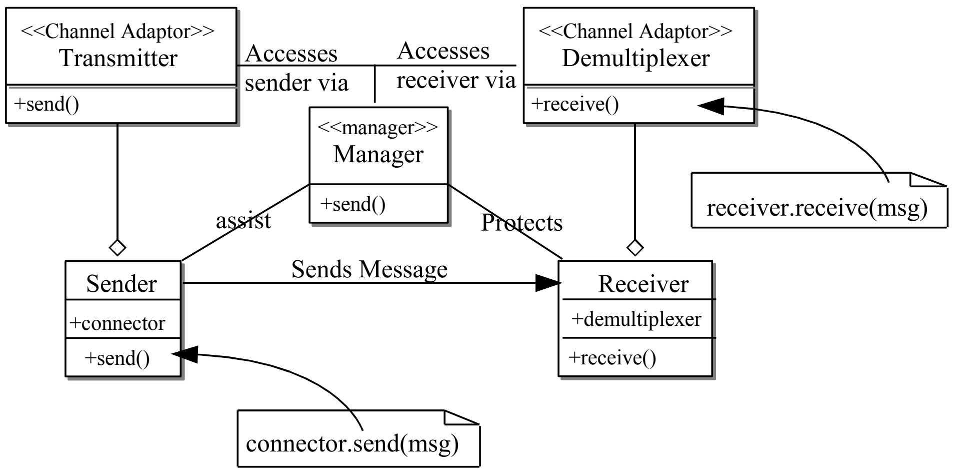


Figure 5.12: Message channel adapters.
They provide management information to connectors (mapped on to the meta-model as channels) which assist the sender in message adaptation if necessary and protects the receiver from unexpected messages delivered to it through the network.

interfaces that are known and accessible by other objects. Depending on the knowledge of the methods contained in the interface of a target object communication happens by the methods being "invoked" usually by specifying method name and passing a set of arguments. Distributed computing heavily relies on interface definitions for communication as unit of reference to object's communication ability. That is why interception of access to methods of an interface, whether remote or local, is an important way of tracking communication between objects in distributed and local environment.

Software engineering has achieved significant developments in specification of mechanisms for separation of object interfaces from the implementation of the functionality that they manifests. There are established patterns that are used to facilitate flexibility, evolution and integration of software components communicating by invocation of methods.

A common way to obtain an object's interface is to use a third component for querying the object implementation by a specified name. The returned reference to the implementation is then narrowed to a known by the calling object interface. This approach can be implemented locally using the Factory pattern, or remote using Naming Service, similarly to the way EJB interfacing works.

In this case, a channel adapter may become active in the moment of object request after which the consequent management operations are able to influence the selection of object implementation. Similarly to 5.3.1.1 and 5.3.1.2 the adapter elements wrap functionality that is responsible for the actual communication establishment and assists in management data delivery to the channel management element.

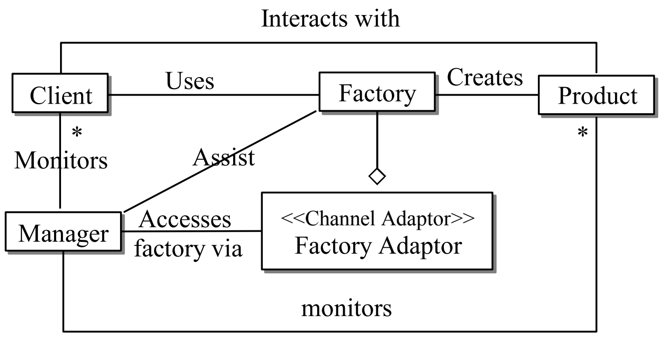


Figure 5.13: Factory channel adapters.

They deliver management information about intention of the Client to consume functionality provided by the Product.

An object may obtain a reference to an interface that matches object functionality not only in the phase of target instantiation as seen in the example with factories. Another way of decoupling interface from implementation is the service-oriented approach used in modern distributed systems and embedded devices. An object requests a list of supported by an already created and functioning component interfaces by querying a registry holding object details published by an object. Once the interfaces are retrieved, the calling object selects an interface and communicates directly with the object. This approach supports true loose-coupling, however there are drawbacks that one has to consider when designing service-oriented applications - the remote object keeps the right to change and re-publish its interfaces, often without feedback to the interested parties, thus inconsistencies in truly distributed but flexible environment may occur with a higher probability.

Partially this is being solved by the abilities of the registries to support multiple versions of interfaces, letting the important choice of selection of proper interface entirely depend on the the callee object. An adapter in this case may provide vital information about the intention of an object to communicate, and thus facilitate assistance in interface selection in a way to satisfy existing dependencies and the requirement for requisite variety. Service-oriented communication as approach is not applied only in distributed systems. For example the OSGi Framework is built around the notion of component services, thus this kind of communication interception is entirely valid for systems that integrate OSGi components. Similar organization has the COM model, where components register interfaces accessible by other elements on demand.

5.4 Meta-model Bindings

The meta-model and its elements presented in Section 5.2 are not useful without defining of concrete relations that associate them with the software entities which

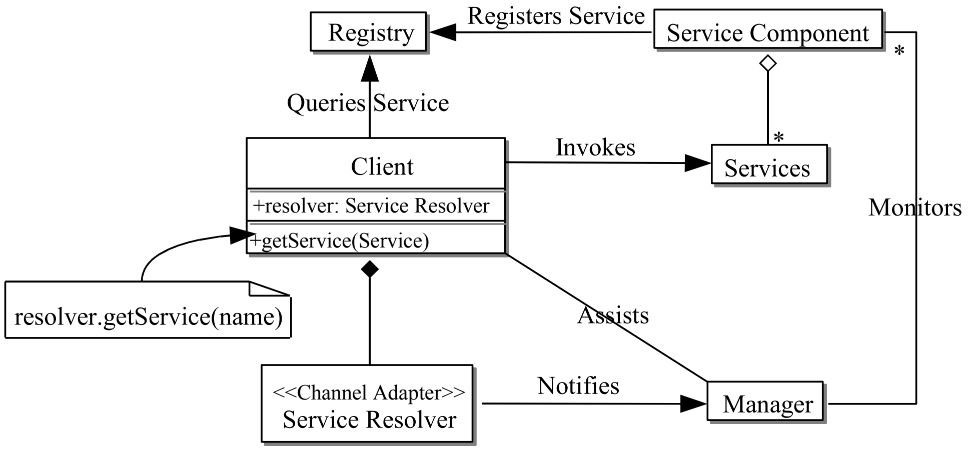


Figure 5.14: Service Resolver Channel Adapter.

It informs the manager about the intention of a client to communicate with services published by a component. The manager decides about the selection of a concrete interface by monitoring the component that provides the services and the adapter information delivered by the service resolver about the required variety.

it aims to describe - the elements of the management system for reflection purposes and the managed elements.

The elements of the management adaptation layer have to be mapped on to the meta-model in order supply the system with information about its own infrastructure and to achieve self-awareness with ability to react accordingly and prohibit fault management operations. The map has to be able to cover the communication by exchange of messages, eventing and invocation of methods and the specific organization associated with these variety types of the system for communication assistance as described in Section 5.3. The bindings are a set of relations that connect meta-model elements with software elements and augment the system with management semantics interpreted in the process of management strategy enforcement.

5.4.1 Meta-model Bindings Interpreter

The information provided by the mapping relations is interpreted by a module the task of which is to monitor the relations of the meta-model and to have the requisite knowledge for the interpretation of the relations binding the meta-model and manageable software elements. It is a required module that contributes to the completeness of the management model in aspect organizational knowledge. Figure 5.14 illustrates the dependencies in the management model and the role of the meta-model interpreter for resolving the bindings.

The example on Figure 5.15 illustrates how elements of the management system having requisite knowledge about the semantics of the bindings are able to enforce

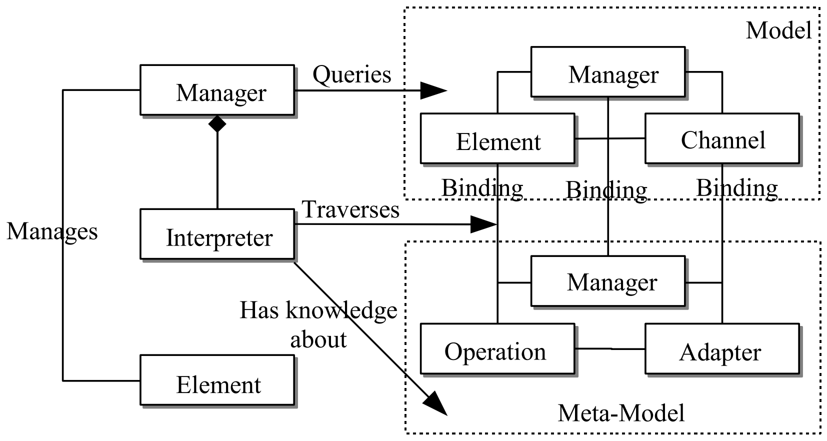


Figure 5.15: A simplified representation of meta-model binding. Elements are mirrored by a model that binds its elements to the meta-model and is stored into model repository. The manager is able to query the repository and respectively retrieve contained instances of the model and inspect their semantics with the help of bindings interpreter that translates the associations between the elements instances and the meta-model to enforce management strategy on a managed Element.

management actions on managed elements, including their own operations.

5.4.2 Channel Management Adapters Bindings

Making the management infrastructure reflective using its own means of information acquiring, descriptive and reasoning methods adds flexibility in the way the system perceives its own state.

Having in mind the semantics in the meta-model elements and the implications of their meaning, the management adaptation layer can be bound to the meta-model following the principles from sections 5.1, 5.2 and 5.3 of this work:

- Events are provided and required variety that components emit and process
- Messages are provided variety to which a service component responds
- Interfaces are provided and required variety which elements use and provide
- Software Elements are communicating endpoints exchanging variety of type events, messages and interfaces
- Channel Adapters are represented as Transducers that can be associated with channels serving the intention of a component to use variety of another component or the ability of a component to accept variety from elements of a system in the form of messages, events or interfaces.

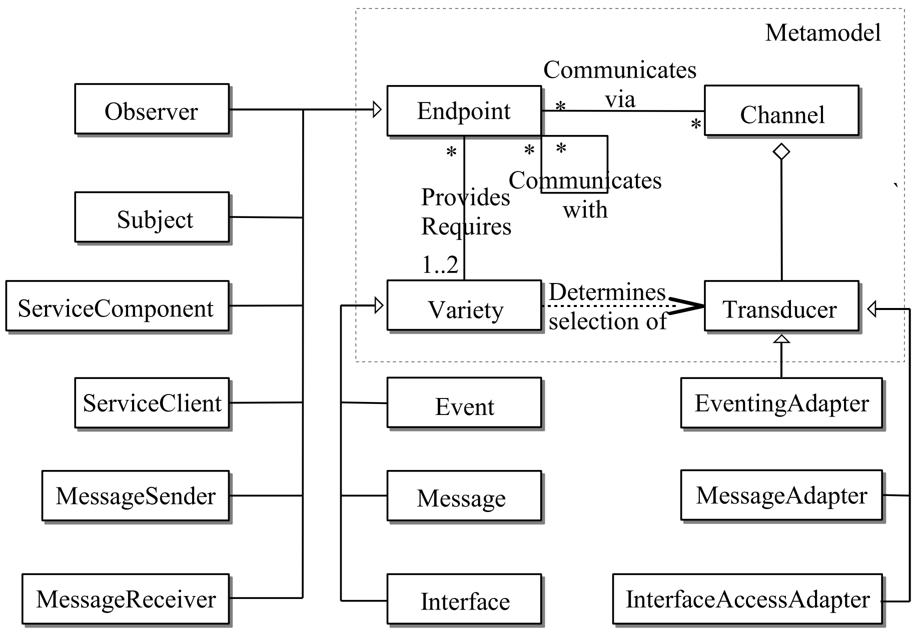


Figure 5.16: Binding the elements of the management adaptation layer to the meta-model.

These prepositions reference the complete variety the management adaptation layer provides in terms of its variety communication and the ability of a management unit to perceive its own state.

Figure 5.16 illustrates the bindings, represented as subclass relations between the classes of the meta-model and the classes representing software entities. The three basic categories of variety transfer - messaging, eventing and interface access - are bound to the variety concept which serves as a criteria for communication channel management and determines the way it is transformed using *Channel Adapters* corresponding to the *Transducer* class from the Meta-model. The communicating endpoints, sub-classed by the three types of communicating parties, and the transducer on the meta-model level are coupled into the virtual medium *Channel* to provide management semantics both for the adaptation and application layers which are interpreted by the semantic interpreted of the respective channel manager monitoring and controlling the communication between elements.

5.5 Communication Management Middle-ware

Before the information about a channel is being passed on to the level of requirements management, the procedures for establishment or initialization of channels and communication coupling is first served by a middle-ware that the architecture defines as *Communication Management Middle-ware*.

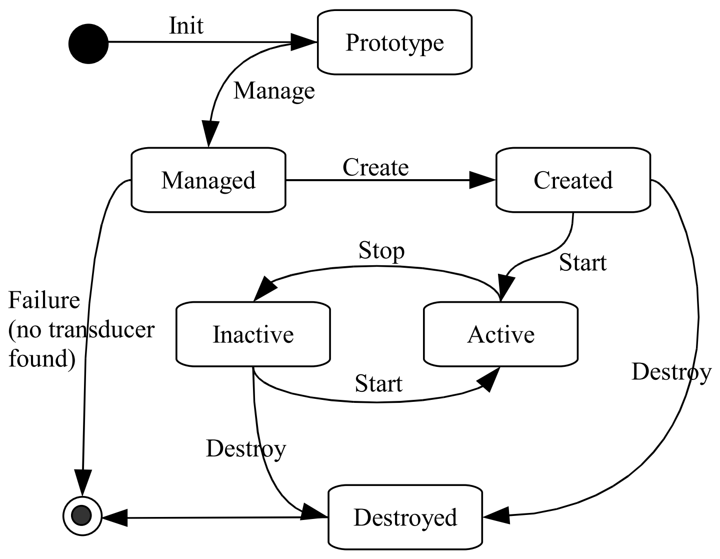


Figure 5.17: Managed Communication Channel Life-Cycle.

The main task of the middle-ware is to assist in the management of the life-cycle of a communication channel.

5.5.1 Communication Channel Life-Cycle

A managed communication channel has a life-cycle that defines six distinctive states, which the middle-ware handles and in which procedures related to creation, initiation and destruction of a channel are consequently executed before and after the actual communication between elements occurs. Figure 5.17 illustrates the states of a channel and the transition handled by the middle-ware.

The channel is created first as a prototype reflecting the intention of the communicating elements, after which a manager prescribes concrete values of its parameters to be created. In the moment of actual component communication channel state is marked as active and not-active after the act of communication. When the components no-longer reference each other the channel is destroyed.

The states of a manager correspond to procedures which the middle-ware executes upon association of a state:

- **Channel Prototyping** - This phase includes steps for creation of a prototype of a channel having the same characteristics of a final channel, but without a specifically associated variety transducer.

- **Manager Cycle** - performed when the channel is in state *Managed.* The manager analyzes the prototype and selects an appropriate transducer to satisfy the requirement of Requisite Variety.

- **Channel Instance Creation** - a channel instance is created having instantiated and associated transducer for variety transformation.

- **Channel Activation and De-Activation** - the phases in which the procedures of channel activation and deactivation can be executed are varying and are related to the concrete mechanism of variety exchange, as outlined in Section 5.4.

- **Channel Destruction** - procedure performed when the communicating components no-longer reference each other. The states indicates no direct influence of one object on the state of others.

5.5.2 Channel Prototyping

When a component declares attempt or intention to communicate with a component in or outside the system, the channel middle-ware creates a **Channel Prototype**. It is a regular channel instance, containing references to the communicating endpoints, but with an empty field for transducer for variety adaptation. A channel prototype can be monitored and accessed, thus its existence allows fine-grained monitoring of interaction, e.g. communication establishment logging, visualization, etc., on the level of channels.

5.5.3 Manager Cycle

Once a channel prototype has been created, a *channel manager* can be activated in order to perform management operations on it. The specific operation that has to be completed in this specific phase is the selection of an appropriate variety transducer the reasoning for which is based on the existing data contained in the channel prototype. The assignment of concrete transducer is a result from the four-step autonomic MAPE management cycle. The management process is decoupled from the specifics of channel information in order to stimulate the utilization of common autonomic management interfaces. The four phases of the management cycle are as follows:

- **Monitoring:** Once a channel prototype has been created, the information relating to the act of communication is being passed to the sensors of the manager.

- **Analysis:** The information is processed and normalized for the internal reasoning mechanism to interpret them, for example rule engine and belonging to it rule sets. The analysis evaluates the communication abilities of the channel against requirements for requisite variety.

- **Planning:** The output of the analysis phase consists of a set of transducer selection options which are then prepared to be used for transducer initialization.

- **Execution:** A transducer is initialized and assigned as subscription for creation of channel instance.

5.5.4 Channel Instance Creation

During the phase of channel instance creation, the prototype is assigned the status of a real channel which is ready to serve variety communication and adaptation between the participating in the act peers. This phase, however only defines the ability and readiness of a channel conforming to the requirements of the management system and does not involve any actions related to communication. The act of communication is declared available, but for actual transformation the channel has to be in state "active".

5.5.5 Channel Activation and Deactivation

The procedure of channel activation includes except flagging of the channel as one that is capable to transmit and transform variety, also necessary notification of the communicating peers. Deactivating the channel indicates the end of communication with the presumption that the two pears still refer to each other and further iterations of communicative acts is possible. Activation and deactivation is handled by the middle-ware itself and is transparent for the peers.

5.5.6 Channel Destruction

When two components are no longer referencing each other this is indication that there is no present intention for communication, thus the existing channel that was created earlier can be destroyed in order to flag inactive dependency between the elements.

5.5.7 Middle-ware-Manager Interaction

The primary role of the middle-ware is to co-ordinate interaction between elements and provide the necessary information about the communication to the channel manager (Figure 5.18).

The mechanism relies on manager decisions to complete the life-cycle and completely satisfies the requirements for transparency of communication, by providing management feedback on every essential point of interaction. Events triggered by the middle-ware are responded by a detailed actions sets, described in detail in 5.7.

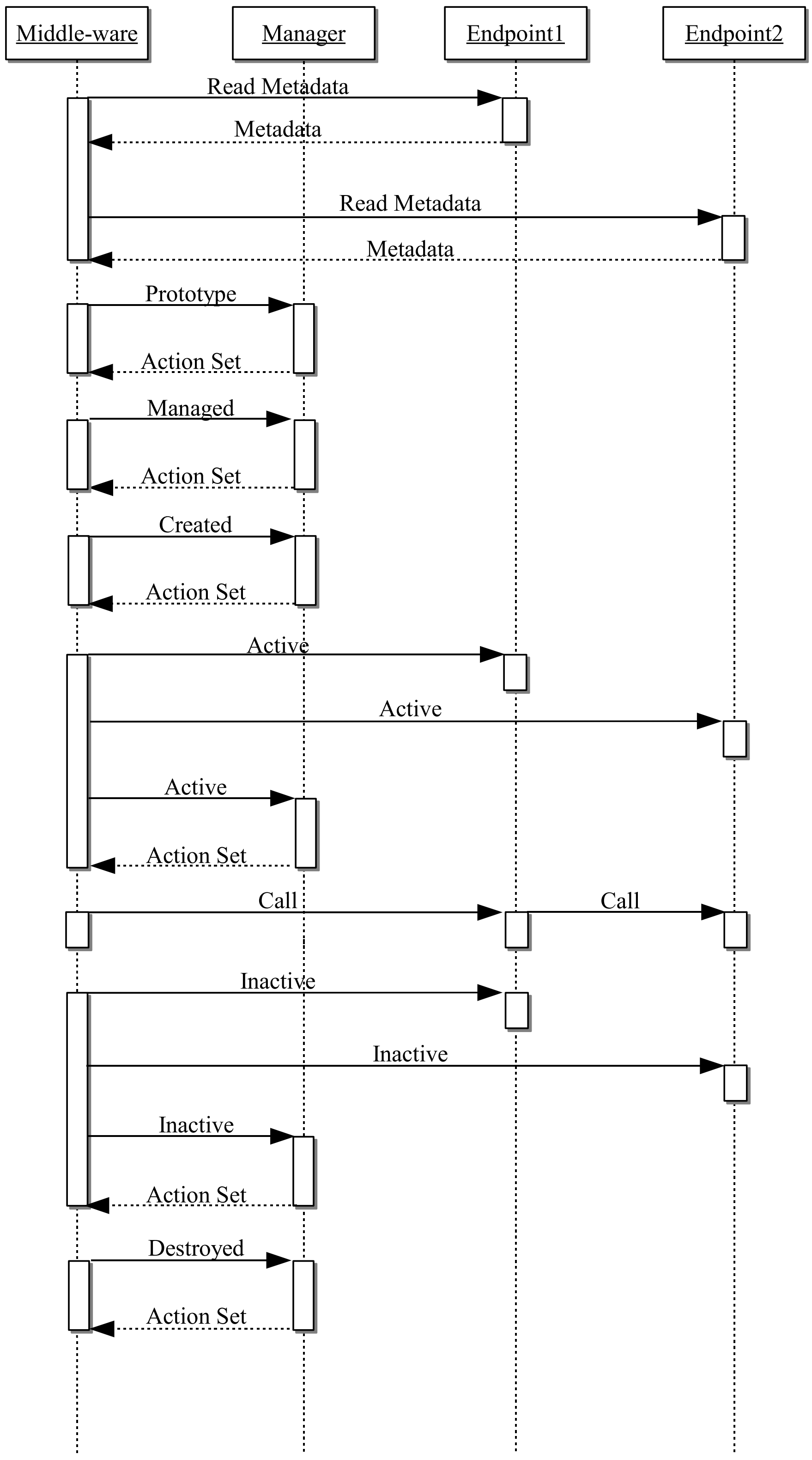


Figure 5.18: Full life-cycle of a managed communication channel.
The middle-ware manages the state transition in coordination with channel manager before and after actual communication.

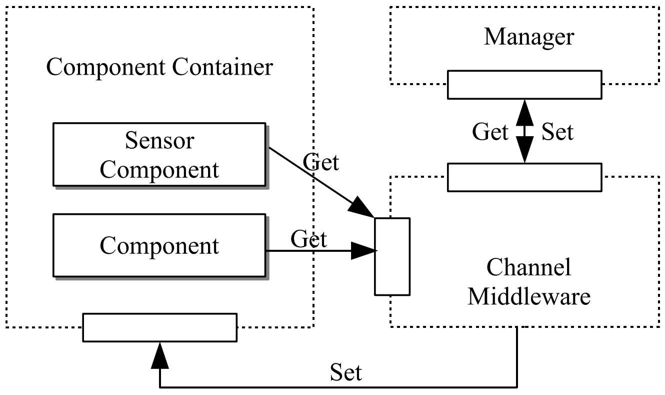


Figure 5.19: Endpoint Interfacing with the middle-ware.

The middle-ware is accessed through its interfaces by regular components that communicate and by sensor components that monitor deployment events. The middle-ware access the components through the container's management interfaces.

5.5.8 Interfacing

The middle-ware provides services for channel handling and endpoint monitoring by access to application programming interfaces (API). The two major groups of interfaces are Endpoint-Middle-ware interfaces and Middle-ware-to-Management interfaces.

5.5.8.1 Endpoint Interfacing

The endpoints declare their intention to communicate by providing necessary information prior to the actual communicative act. In the case of software components there are two ways to do this - by explicitly defining required and provided communication type and interfaces using descriptors (heavily used in component deployment), or in case the dependencies are not clear prior to component deployment this is done through interfaces provided by the middle-ware, the methods of which can be used to achieve actual communication in the forms of events, messages or methods and interfaces. While the first way of communication dependency declaration facilitates preliminary channel types observation, the second indicates exact runtime communicative act and facilitates dynamic management.

The optimal way in which a component can define its dependencies and intentions to communicate is to use deployment descriptors for on-deployment declaration of dependencies, and to utilize the middle-ware API in run-time upon communication with a local or remote component.

The two ways of channel observation outline two main access points to the interfaces of the middle-ware. Endpoints declaring in run-time access the middle-ware directly, while descriptor-based declaration requires interception of deployment events on the level of component container, because the endpoints are not active in the moment of deployment. While the first type of access is obvious, the second one may introduce difficulties when the architecture of the component container does now allow interception of certain events. This thesis focuses on well known industry standards that allow such observation, thus these kinds of problems are not discussed.

Figure 5.19 illustrates the interception of intentions for communicative acts based on the discussed two types of endpoint-middleware interfacing. The delivery of communication events is driven by notification either from the components themselves or from components that intercept deployment events and read data related to communication dependencies. The information is passed to the middle-ware layer respectively in run-time or in deployment time.

5.5.8.2 Manager Interfacing

The middle-ware performs the operations required for extraction of information from the peers and the managing the life-cycle of communication channels.

The rest of the required functions are delegated to the management layer, which takes decision about the future state of a channel. That is why the middle-ware serves as a mediation layer between the endpoints and the channel managers. The different strategies and the way a manager works may vary, depending on the type of managed variety, that is why the specification of interfaces through which the middle-ware accesses manager's functionality and the interfaces used by the manager to access middle-ware have to be specified carefully and with re-usability in mind and to enable dynamic integration of managers. For this purpose the design pattern *Abstract Factory* can be used, assuming predefined interfaces. For potential changes in both manager and middle-ware access interfaces the *Extension Objects Pattern* offers the ability to define additional interfaces to existing objects. Figure 5.20 illustrates usage of these patterns within the context of middle-ware - manager interaction, with the ability to change the managers, depending on the type of the managed channel that is being handled.

5.5.9 Channel Meta-data

One of the most important aspects of component management is the way relevant information about the state of a component, used resources or dependencies is delivered to the management module. In the case of component communication, and the introduced concept of communication channel, the relevant information for the management modules comprises the following elements:

- *Type of communicated variety* - the type of communicated variety (variety class) can be *message*, *event* or *interface*. Every concrete variety of one of these types is associated a version number indicating the evolution.

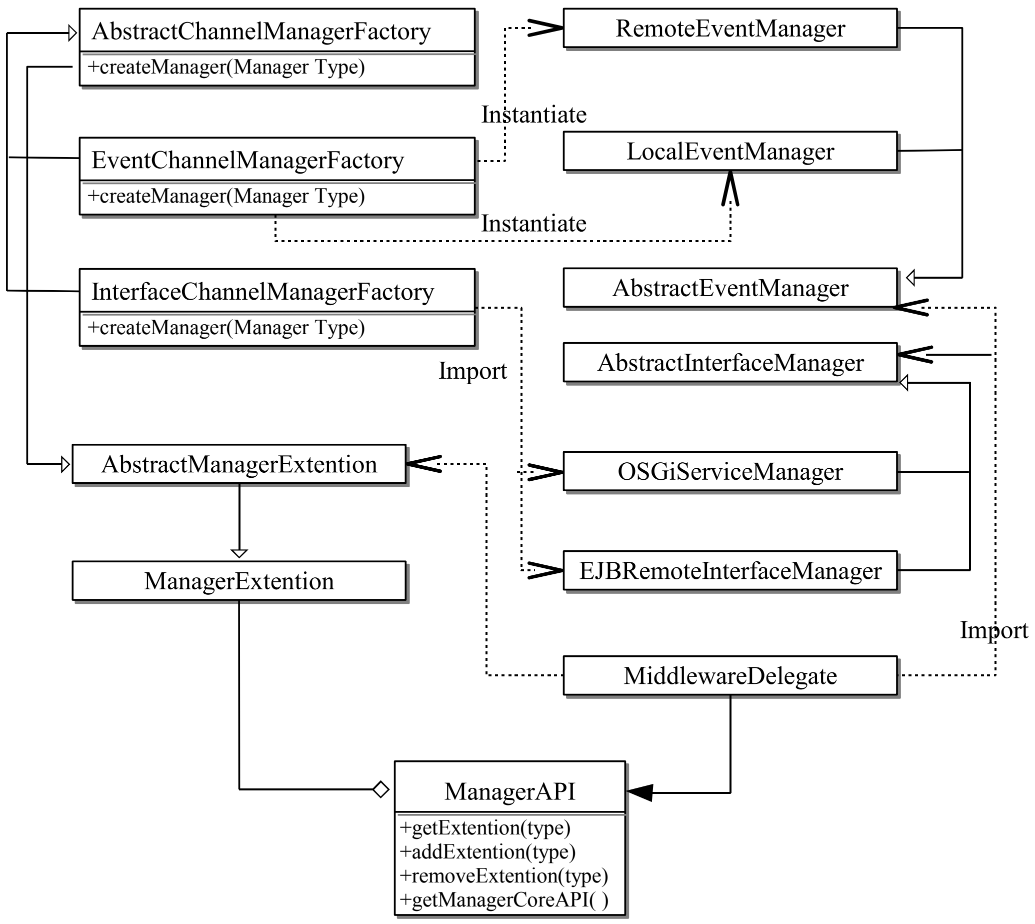


Figure 5.20: Interfacing between Middle-ware and the Manager.

The middle-ware delegate uses the manager API to access the core interface for creation of the integrated manager types. The API can be extended with additional functionality using the Object Extensions pattern, by specifying the concrete type of extension.

Meta-Data Class	Representatives	Description
Variety Type	Event, Message, Interface	Type of exchanged variety
Required Variety	EventHandler, MessageParser,Interface Endpoint	Declared by endpoints to express intention of communication
Provided Variety	Event, Message, MethodInvoker	Declared by endpoints to express availability of variety handler
Transducer	EventVersion, InterfaceVersion, EventHandlerVersion, MessageVersion	Declared by the management system to express availability of variety transformation

Table 5.1: Meta-data classes and representatives.

- *Variety requirements* - every communicating peer declares requirements about a concrete variety that it expects or requires during communication with other elements. This information can be used to construct a table of possible interactions and dependencies with towards other elements of a system.

- *Provided variety of sender and receiver* - every communicating peer declares concrete variety that it provides as means of communication that is initiated from other elements.

- *Available variety transducers* - management units need to know what type of variety transformation is available in order to select appropriate transducer. A Transducer Variety Class is composed of element of element variety requirements and provided variety (see above) for its input and output. A version is associated to indicate transducer's evolution.

The concrete formulation of meta-data records used with the middle-ware is described in Appendix B.

5.5.10 Declaring and Consuming Meta-data

The nature of the heterogeneous component systems determines a large variety of meta-data expression. Some of the frameworks exploit the flexibility of XML as a way to express extensive sets of data, but others are still using plain text based, custom formatted fields or system-dependent registry mechanisms.

In order to support independent and loose relation between the managed system and the management system, there are no meta-data declaration requirements related to the alien for the particular framework formats of declaration. This leads to the requirement the elements (sensors) that are responsible for data acquisition to be able to parse the channel meta-data records that various elements are declaring or are exhibiting in run-time.

An endpoint may declare its variety meta-data in two ways - static and dynamic.

5.5.11 Static declarations

Static declarations are needed in the phase of component deployment. This is the moment where new variety is introduced in the system, and the moment in which the system management has to be able to observe possible dependencies and communication of the newly introduced component with other parts of the system.

The declaration of variety happens with the help of explicitly defined list of attributes attached to the actual code of the component. In the case of EJBs the meta-data is encapsulated in the form of XML files. The J2EE specification defines the proper place for explicit, non-specification related meta-data which is accessible and readable by the framework environment. In the case of OSGi additional meta-data is attached to the bundle manifest file together with the standard meta-data records defined in the framework specification. However, the concrete implementation of the OSGi framework may introduce different ways of meta-data storage.

Static declarations are recommended way to inform the framework and its managing components about possible interactions at run-time.

5.5.12 Dynamic declarations

Dynamic meta-data declarations are used to indicate a concrete act of communication that may correspond to meta-data declared statically, but it may also be used to inform the framework management about previously undeclared dependencies. Dynamic declaration is achieved by simply passing argument to the channel API provided by the middle-ware in the moment of acquisition of references to other elements. The middle-ware then compares the meta-data with the existing channel dependencies defined in advance and drives the preparation of channels through the cycle described in Section 5.5.

5.5.13 Consuming Meta-Data

The way variety meta-data is consumed does not depend on whether it is statically or dynamically defined. The middle-ware provides internal interface through which the core accesses specific meta-data parsers and normalizes it to the internal representation of the channel structures. Dynamic declaration is achieved through the channel API. Figure 5.21 illustrates the basic approach for normalization of variety meta-data in statically and dynamically defined meta-data records.

As illustrated on Figure 4.7 a deployment sensor is a first-class component in the domain of a framework hosting the communicating components. In the case of

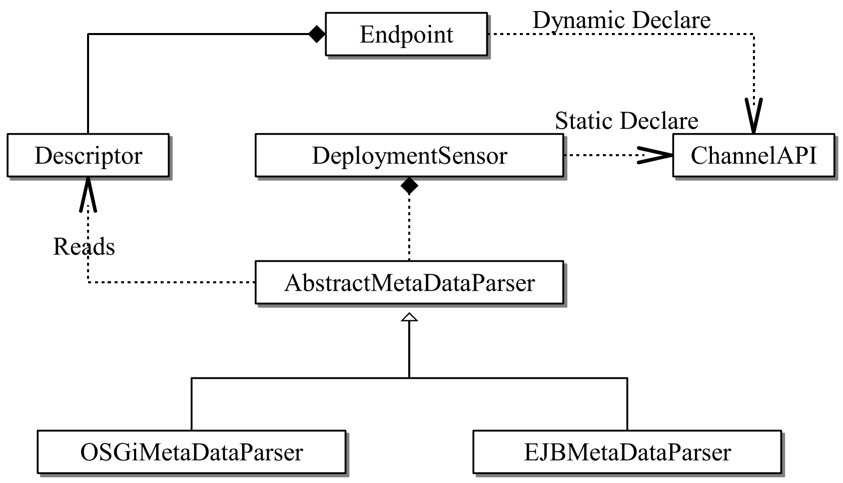


Figure 5.21: Consuming variety meta-data.
Parsers are independently reading meta-data information from statically defined records, initiated by the deployment sensor on component deployment, while endpoints are declaring variety meta-data dynamically. Both types of declaration are handled through the Channel API.

the OSGi Framework a deployment sensor is a bundle that is subscribed to receive bundle deployment notifications from the framework. The passed to it context object allows it to access the deployment descriptor of the bundle in question. In the case of EJB deployment sensors are JMX dynamic beans which are configured to receive notification from the JMX framework when a bean is deployed. Both methods are standard way for monitoring element activities in their hosting environment and do not influence the operation logic neither of the hosting framework in the process of deployment nor the components themselves.

The way static meta-data can be acquired from other component technologies such as COM and SLEE is similar to the presented one - by subscribing for deployment events delivered through the underlying management framework.

5.6 Cybernetic Viability Requirements for Software Systems

The adaptation of the Viable System Model requires to be associated with the meta-model representation explained in Sections 5.1 and 5.2 but this mapping is insufficient as it is only associates software entities with the abstract concepts of the model. VSM includes a set of rules, axioms and guidelines that reflect the way a system has to react depending on the changes in its environment. Thus, an adaptation of the VSM requirements for the purpose of its application with software

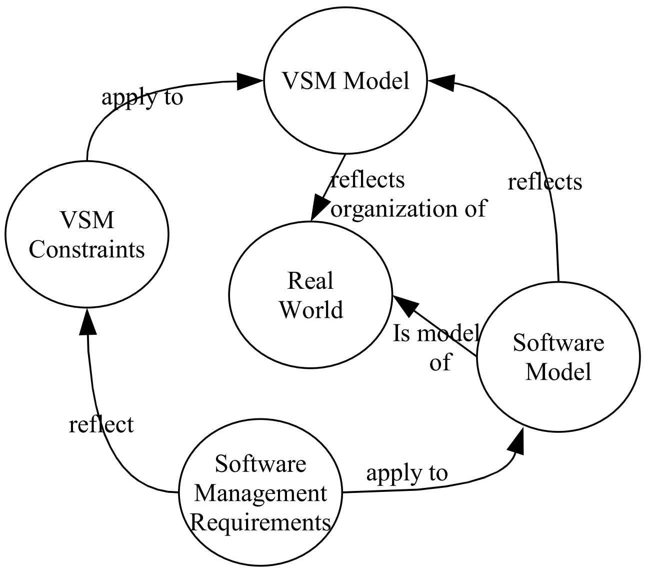


Figure 5.22: Relations between the VSM model, its software adaptation and the respective requirements.
The real-world domain is a point of reference for both software model and VSM, but at the same time the software-model reflects the VSM model to comply with its constraints through adapted management requirements.

is needed to complete the model with its corresponding policy for management.

Such an adaptation has to re-define the original requirements in terms of software entities as described in the base meta-model. The evaluation of requirements have to be enforced by the management modules over the entities that are associated with the meta-model.

Figure 5.22 illustrates the adaptation process and enforcing of VSM rules over software entities. It can be easily compared to Figure 4.1 in terms of model relations to the real-world domain. Introducing the VSM model as an additional reference model for the software model contributes to a formal method of requirements definition and management. Among the many advantages of automation is the ability to introduce parametric handling of management routines.This aspect of management is reflected in autonomic system design practices by integration of management modules that can be governed by a modifiable policy.

The VSM requirements which have priority in the process of adaptation are the already introduced law of *Requisite Variety* and *Requisite Knowledge.* Additionally VSM introduces axioms related to the flow of knowledge in two dimensions - horizontal and vertical - expressing guidelines for amount of *variety absorption* on the different levels of management. Finally, the dynamics component of the requirements set deals with the speed of variety flow across the boundaries of communication, as well as speed of management decisions.

5.6.1 Requisite Variety Management

Communicating components exchange information through their communication channels which define the existing variety of possible way of interaction between them. Arguably the most important one in the set of VSM requirements is the *Law of Requisite Variety* (LRV).

5.6.1.1 Requisite Variety of a Component

An adaptation of LRV for software has the following definition:

> *For a component to remain stable in its hosting environment it has to be able to handle at least the variety of events, messages or communication interfaces used by other components to communicate with it.*

This definition references the already defined types of communication used to abstract the different classes of communication channels in the proposed meta-model. The rule applies not only to the ability to transmit the variety, but also to be able to process it flawlessly before it has been consumed in the pipeline of the business process that the component supports.

5.6.1.2 Requisite Variety of a Communication Channel

In the definition of Beer's VSM (Section 2.4.3) LRV is applied to the communication channels connecting the operations and management layers of the model. The adapted version for software communication channels is defined as:

> *The communication channels used for communication between software components must have the requisite variety to be able to transmit the variety of events, messages and interfaces exchanged between the communicating endpoints.*

The ability of the communication channel to transmit correctly variety from one endpoint to the other requires the transducers involved in specific variety transformation to have the requisite variety to be able to modify, amplify or filter variety in a way to prevent variety loss on the receiving end of the channel. This requirement is also part of the directives related to LRV applied in VSM communication management. Its adaptation in software terms defines specific modification interfaces, events and messages in the following way:

> *A software variety transducer involved in channel communication between two software elements has to be able to transmit communication data in the form of events, messages or interface communication, modify it or filter it in such a way that the receiving endpoint is able to consume*

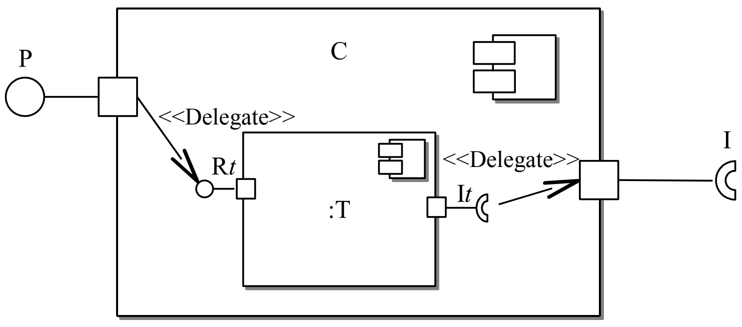


Figure 5.23: An example for a communication channel.
Channel C with port P and destination reference I, having transducer T with source variety Rt and destination It.

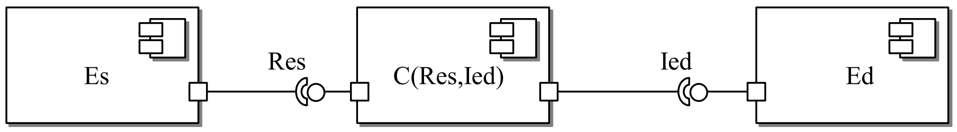


Figure 5.24: Application of a communication channel in interaction between software components.
Components Es is the source and Ed is the destination.

> *variety with a semantic load equivalent to the one sent by the transmitting endpoint but adapted to the communication interface, event handler or message handler of the receiving part.*

The rule states that although a transducer may modify transmitted through it variety, it has to provide the receiving party an understandable interface, but at the same time meaningful data, enough to be processed as meant by the sending endpoint.

5.6.1.3 Formal Expression of LRV for Software

While expressing the requirements with natural language gives a narrative way of understanding the mapping, a formal representation contributes to the algorithmic way of expression which is directly used for the implementation of a strategy for verification of the requirement for every communication channel.

The annotations on Figure 5.23 and Figure 5.24 are used to create a discrete function for evaluation of the *Channel compatibility* and the requirements for requisite variety.

There are two components E_s and E_d and E_s, where is initiating communication to E_d for its actual variety I_{ed}. Expected variety of E_s is notated as R_{es}. A channel C is determined by its *input port P* and its output I. Figure 5.24 shows a selected channel $C(R_{es}, I_{ed})$ for R_{es} and destination variety I_{ed}.

Differentiation between the P and R_{es}, I and I_{ed} exists to constitute channel existence as a first class element in component communication.

A *channel transducer* T is determined by its input variety R_t and its output I_t and is selectable among a set of available transducers N:

$$T_x(R_t, I_t) \text{ where } X = 1 \ldots N \tag{5.1}$$

with requirement for requisite variety of a channel:

$$R_t = P, I_t = I \tag{5.2}$$

The expected variety R_{es} indicates the type of communication the source element is expecting when initiating communication to the destination element - a concrete type of event, message or interface.

Actual variety I_{es} is the type of communication the destination element can handle - a concrete type of event, message or interface. A concrete type of element is a software element tagged with evolution identifier, such as version number having semantics assigned to its minor and major components.

If we are able to translate these semantics to meaningful values in the *set of compatibility* in communication between two elements, then the channel compatibility function is:

$$B(V(P), V(I)) = [-1, 0, +1] \text{ where } P = R_{es}, I = I_{ed} \tag{5.3}$$

where the set of values [-1, 0, 1] have the following semantics and represent the set of values for *management feedback for communication*:

(-1) - communication impossible, no transducer available

(0) - no transducer needed for communication

(1) - communication possible with transducer

In the case when no transducer is needed for communication (1), the channel simply connects the two endpoints. This type of channel is referred to as Null-Channel C0 and represents a basic dependencies in its classical meaning.

$$C_0 R_{es} = I_{ed} \tag{5.4}$$

This is a special case where the channel has the requisite variety and is enough to indicate proper component communication but generally, any value of B greater then 0 indicates requisite variety Vr:

$$V_r : B(R, I) \geq 0 \tag{5.5}$$

Expression 5.5 defines the requisite variety when 5.1,5.2 and 5.3 being satisfied.

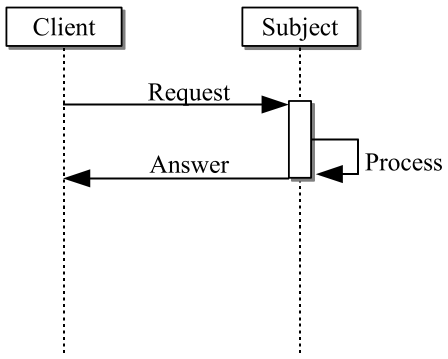


Figure 5.25: Communicative act with feedback.

It requires processing from the side of the requested. The variety exchanged between the two elements has to be supported by existing knowledge about it for adequate answer.

5.6.2 Requisite Knowledge Management

Although LRV has to be applied and without it the communication is unpredictable there is the need the received variety to be interpreted.

Without the knowledge about how information has to be processed and what semantic value it carries a manager is incapable of taking fully backed with arguments decisions about its reaction on event or situation change.

This statement is known as the Law of Requisite Knowledge (LRK) and is a requirement for meaningful communication between managers and managed elements and more generally, communicative acts with feedback iterations (Figure 5.25). The adapted version of this requirement for software elements states:

> *The logic that implements a provided interface, event or message handling has to represent the required knowledge for adequate processing of variety upon receiving. When the processing logic only provides means for delegation to dynamically loadable modules, there have to be at least as many of them to support the provided variety.*

Often event or message handlers implement directly event processing, however in the case of frameworks (containers, platforms, etc) handling is abstracted to the point where event is only absorbed but no processing logic is triggered implicitly. Then, explicit event handling by loading of components that are capable of adequately process the incoming events (plugins, bundles, etc) represent the requisite knowledge only in the case where the combination of one, two or more of them is able to process the supported by the framework events or messages.

In a software environment with loadable modules for processing of incoming variety the monitoring of available modules and the provided variety to external for the system elements may prevent unexpected security issues or potentially improve stability by ensuring that there are no open or no handled situations where the whole system may be exposed to unexpected behavior. The declaration of processing knowledge is already available in the overviewed component platforms.

5.6.3 Vertical and Horizontal Variety Balance Management

As a preposition for efficiency of management VSM defines additional requirements which assert equality of variety in different checkpoints in the model. The *first axiom of management* states the following:

> *Axiom 1: The sum of horizontal variety disposed by all the operational elements equals the sum of vertical variety disposed on the vertical components of system cohesion.*

This axiom expresses the basic importance of homeostasis where additional variety coming from the environment into the operation then into the management of the operation, has to be canceled out by the variety coming down the vertical channels of the supervision system (System 3).

The adaptation of this axiom and its application as a management policy for software systems with autonomic control can influence the planning and assignment of concrete roles to management and and to operations according to the active software environment in which the system operates. The second axiom of management states:

> *Axiom 2: The variety disposed by the supervision system (System 3) resulting from the operation of the first axiom equals the variety disposed by the planning system (System 4).*

The axiom sets the requirement for balance between the senior management and environment prediction, where System4 has to be aware of the future state of the environment but not to force System 3 in directions where the system may not profit or realize operations, but make expenses on system resources.

The axioms involve quantitative measurement of variety, while software management operates only with discrete variety, thus the function for evaluation of management disposal throughout the system has to be re-defined with the discrete values of in the *set of management feedback for variety disposal (Table 5.2)*. The set maps an aspect parameter to a balance value, giving the opportunity to refine the monitoring and analyze concrete unbalanced areas. Horizontal disposal refers to Axiom 1 while Vertical aspect refers to Axiom 2. The implementation of this policy ensures proper management organization and directly relates to the ability of a system to be self-aware.

This mapping is used in the implementation of the prototype (Chapter 6).

Aspect	Value	Description
Horizontal	0	Unbalanced variety – the operation disposes more than the manager can handle
	1	Balanced
Vertical	2	Unbalanced vertical disposal
	3	Balanced

Table 5.2: Set of Values for Management Feedback for Variety Disposal

5.6.4 Communication Dynamics Management

VSM defines a principles for organization which are both static and dynamic. While static ones relate purely to the functional separation of duties and roles, the dynamic are introducing the element of time.

The channels carrying information between the management unit, the operation and the environment must each have a higher capacity to transmit a given amount of information relevant to variety selection in a given time than the originating subsystem has to generate it in that time.

This principle states that a channel has to be able to transmit as fast as possible the information between the communicating components, for a shorter time than then the element that produces variety sent over the channel. If this rule does not hold the system becomes unstable.

This rule is directly applicable to the specifics of software component communication with the requirement for distributed systems to be able to communicate in timely fashion, especially in high load of generated variety of events, messages or remote procedure calls. Critical for the operation of the systems are communications related to database querying and synchronous remote operations. The ability of a channel to transmit a required amount of information is often identified by measuring the time for receiving a message passed through it. A policy monitoring communication dynamics can observe these values and depending on a threshold defined in its management policy a channel may be marked as faulty in respect to this requirement.

Another method of measuring dynamics or responsiveness in communication is the average number of variety units waiting in a queue to be sent over the channel in a certain period of time - an established practice in system load measurement in Unix-Like systems process scheduler queue. System administrators may monitor the load values for three consequent periods of time and depending on the variation of these values the administrator can be automatically informed about the system being unresponsive or under heavy load which indicates impossibility to react to introduced amount of information.

The proposed adaptation of this requirement for channel communication between software components introduces a similar strategy for monitoring the responsiveness of the middle-ware in connecting requested peers by the ability to set up a threshold value for duration of channel establishment. When this border value *Tmax* is being crossed, the middle-ware indicates the communication failure in the log with a value within the set of *Channel Dynamics Status (CDS)* and optionally notifies the administrator. The discrete function that evaluates a concrete channel ***C*** with threshold value ***Tmax*** the status is:

D (C, Tmax), {CDS}, where CDS = {0,1} (5.6)

The semantics of the values within CDS are 0 for dynamics in allowed range, 1 for indication of difficulties in communication.

This approach effectively discovers communication problems and difficulties which otherwise are fairly invisible to the system management staff.

5.7 Rule-based VSM Policy Enforcement

The formal elements and requirements for management of software component communication described in sections 5.3 through 5.6 have to be implemented within the logic of the channel managers. In order to facilitate tuning, easy addition of rules, that may not be inside the set of VSM management or other modifications, the middle-ware provides a flexible mechanism for definition of policies with the help of rule sets. A rule set is a chain of conditions and associated actions executed upon occurrence of events. Rule-based management is the foundation of knowledge-base management and general policy enforcement within modern, intelligent systems with integrated mechanisms for automated behavior.

The proposed set of events, conditions and action declarations aims to enable custom implementations of channel management routines in general, and in the concrete case the adaptation of the general VSM rules, axioms and guidelines. The set consists of declarations of events delivered by the channel management middle-ware, conditions for definition of constraints applied to these events, and a set of actions executed upon condition fulfillment. The actions affect final channel configuration and deliver feedback in the form of log, mirroring the status of communication channel (Figure 5.26) with its *Management Feedback for Communication*, *Variety Disposal* and *Channel Dynamics Status.*

A manager that implements channel management policy has to be aware of the whole event set delivered by the middle-ware in order to comply with the LRK.

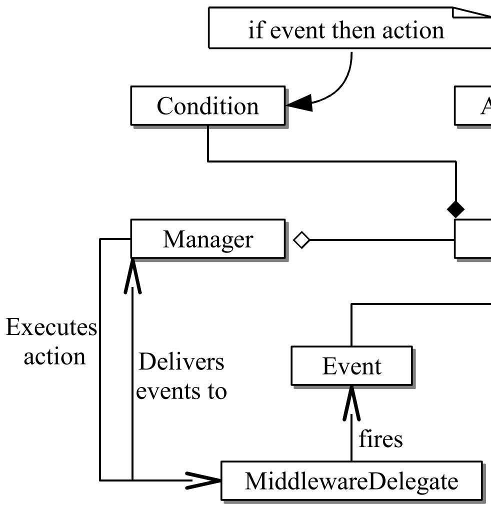


Figure 5.26: Event-Condition-Action rules implemented as a policy in manager.

5.7.1 Channel Events Set

The middle-ware communicates with the channel managers through event-driven API, where for every communication act there is a set of events responding to peer and communication channel's state in its life-cycle. The payload (Table 5.3) that an event carries includes all elements needed for a manager to start the *manager cycle*, reason with its condition set and create an instance to complete it, after which the middle-ware takes over and activates the channel.

Every event carries information about itself and information related to the initiation of the channel. The *Ref_ Channel_Prototype* property refers to the created by the middle-ware channel prototype and is used by both manager and middle-ware in the prior to management phase and after the manager has applied its operations. *Ref_Src_Endpoint* and *Ref_Dst_Endpoint* are used to refer to the actual instances of the elements in order to be accessed directly if needed.

The elements are produced sequentially and every event is assigned a unique ID and a time-stamp, indicating the precedence order and the actual time of creation. Both properties, respectively *Event_ UID* and *Event_ TimeStamp* are important for purposes of dynamics management as described in Section 5.6.4. Although the manager cycle is performed once during the life-cycle of a channel, the existing set of event types assures manager feedback on every single state transition. That is why there are several types of events that the middle-ware may assign to the Event_Type field of the event class. Whether the manager will trigger a complete cycle over the triggered event depends on the concrete strategy implemented by the rule-sets. In the default implementation of the VSM adaptation manager feedback is performed only when the channel is in state “Manager_Cycle”, however, in a custom implementation the state-transition may be modified according to the feedback delivered

Property	Description
Ref_Channel_Instance	Reference to the channel prototype or created instance
Ref_Src_Endpoint	Initiating communication endpoint
Ref_Dst_Endpoint	Destination communication endpoint
Event_Type	Type of event
Event_UID	Unique ID of event
Event_TimeStamp	Time-stamp of event instance

Table 5.3: Payload properties of the Channel Event.

Channel State	Event Type	Description
Prototype	ChanState_Prototype	Channel has been intercepted and prototype created
Managed	ChanState_Managed	Manager cycle needed
Created	ChanState_Created	Channel has been created successfully
Not-Active	ChanState_NotActive	Channel is inactive, both peers are not exchanging any information, but are still referencing each other
Active	ChanState_Active	Channel is active, both peers are communicating
Destroyed	ChanState_Destroyed	Channel was destroyed

Table 5.4: Channel State-event mapping.

The manager is able to respond to any change in the state of the communication channel.

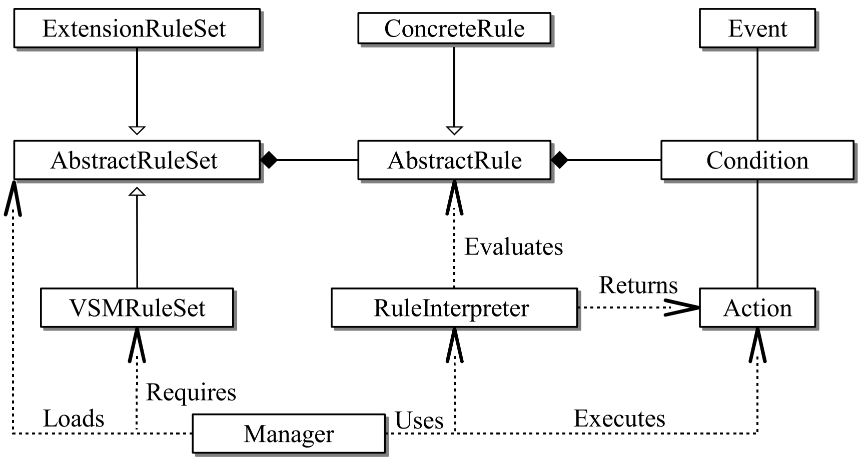


Figure 5.27: Rule-sets representing executable policies upon evaluation of event. The VSMRuleSet is required by the interpreter for intrinsic VSM LRK completeness, while an ExtensionRuleSet defines additional requirements that a specific environment may demand in order to be managed.

by the manager for every new state. Table 5.4 shows the mapping of channel state and events with brief description of their purpose. The events that the manager processes have to carry enough information about both dynamic and static characteristics of the communication. The event *ChanState_Prototype* supports both variety management information with its reference to the endpoint meta-data and at the same time it includes time-stamp information that helps in discovery of time-delays in management. This enables every management operation to be measurable in term of duration in time and contributes to the ability of the management framework to monitor its own operation dynamics. Events *ChanState_Active* and *ChanState_NotActive* have additional meaning for manager because their order in the channel state transition together with the time-stamp information that the event caries determine exactly the duration of every communication, thus providing the rules evaluating dynamics metrics the necessary information for accurate reasoning.

5.7.2 Condition Sets

Distinctive characteristic of policy-based reasoning is the ability to change and adjust policies according to the requirements for management.

That is why an important aspect in channel management is the ability to dynamically define the decision logic according to which the middle-ware handles communication, thus allow for tuning and specific configuration of channel handling in general autonomic computing management, and VSM-based parameter tuning for the specific case of VSM requirements management for software.

A VSM condition set represents the needed reasoning logic that a manager can

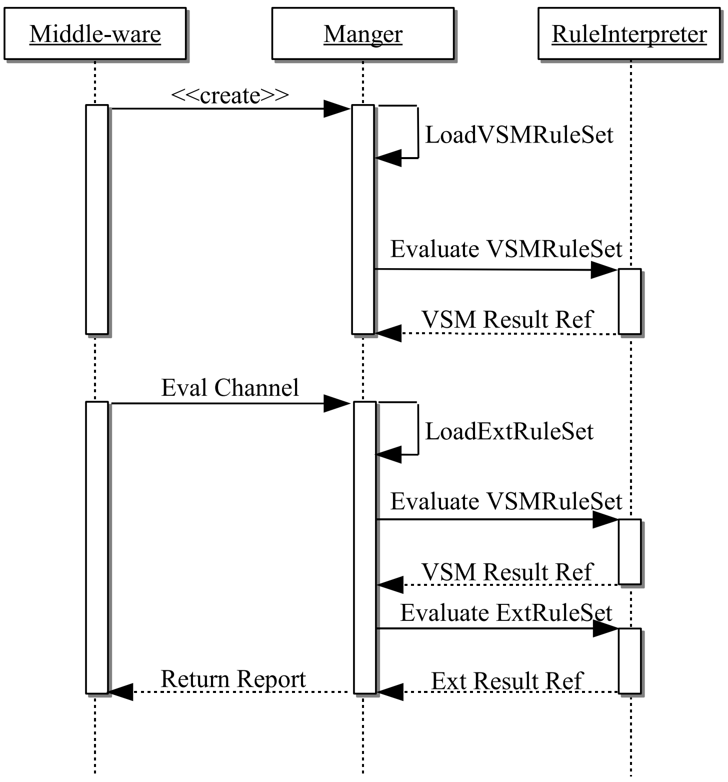


Figure 5.28: Rule evaluation by channel manager.
The first routine that a manager executes is evaluation of the VSM set of rules upon its own creation. When the middle-ware requests evaluation for concrete channel, the manager takes into consideration both sets - VSM and Extended rule-set.

interpret to evaluate the state of component communication, abilities and needed parameters for software components to achieve expected communication behavior.

Autonomic management implies the characteristic of self-awareness, and for compliance with this requirement the proposed here architecture assumes intrinsic specification of routines that perform self-monitoring and decisions related to self-analysis. While self-monitoring capabilities were discussed in Section 5.1, the self-analysis requirement is met by the specification of a core rule set, corresponding to the adapted principles for VSM management in three aspects - channel variety management, knowledge variety management, and dynamics management. In addition, an extension set of rules can define additional constraints or rules that trigger desired actions. Figure 5.27 illustrates the hierarchy and the bound requirement of a manager for existence of the core VSM requirements set.

The core set is not a subject of change and decoupling from the management whole rule-set is avoided in order to provide the pre-conditions for management health in aspects sufficient ability to handle channel events and knowledge for rea-

Action Property	Description
Src_Event_UID	declares which event has triggered the rule
Action_UID	unique action id
Report	contains understandable by humans reason for the action
Timestamp	time of action issuing
Action_Name	concrete action name
Action_ParamSet	parameter values

Table 5.5: Action properties.
Property values have to ensure human-readable reports and concrete action instructions.

soning about them. Figure 5.28 illustrates the sequence of calls preceding the actual evaluation of a communication channel by the channel manager. In the moment of manager creation, the core VSM requirements rule-set is loaded to assure sufficient knowledge for channel management, after which the channel is evaluated against them and optionally against the additional set of extension rules. Using the same communication scheme the middle-ware may require evaluation by the manager on every event from Table 5.4, thus increasing the verbosity of management feedback and respectively the opportunity to inspect communication between distributed elements in greater details, a characteristic that potentially saves time for expensive inspection and manual event correlation.

5.7.3 Actions Set

The role of the manager is to prescribe a list of actions along with reports about a situation, which will be carried out by the middle-ware in the appropriate time for channel tuning. This prescription is in the form of list of action structures initialized by the manager with the appropriate values and represents the planning step in the standard MAPE management cycle.

An action is a structure with several properties that are needed by the middle-ware to continue operation.

In order for the middle-ware to correlate the event and the action, the manager has to provide the needed information.

This is done through a field *Src_Event_ UID* carrying the *ID* of the event triggered by the middle-ware. In order to indicate the speed of reaction of the manager, the action provides a *Timestamp* field having as value the exact time at which the action was issued. Additionally, every action in the list has its own ID which is later used for the actual execution log. Although autonomic systems aim to reduce the

Action Name	Parameters	Description
SetTransducer	Ref_Channel, Ref_Transducer	sets a transducer for the specific channel
NotifyEndpoint	Ref_Endpoint, Message	notifies endpoint with a message
DestroyChannel	Ref_Channel	force channel state to destroyed
ActivateChannel	Ref_Channel	activate channel and set state active
DeactivDeactivateChannel	Ref_Channel	de-activate channel and set state to inactive
NullAction	-	used only for report purposes

Table 5.6: Action names and parameters for channel management.

human role in management, the final instance of management is the human administration, that is why every management decision carries a human-readable report, which may be logged in the process of system inspection or debugging. The fields *Action_Name* and *Action_ParamSet* are providing the necessary instructions for the actual execution of the management decisions. The two fields form a semantically valid call that corresponds to the channel API functions for control of the channel instance.

As it can be seen from Table 5.6 the actions can be used in any of the states in which a channel is set. This way for every state transition, the middle-ware refers to manager decisions, which are queued and executed in order of issuing.

Chapter 6

Implementing a Middle-ware for Management of Component Communication in Heterogeneous Software Environment

This chapter overviews an implementation of a middle-ware for distributed management of component communication and focuses on the fulfillment of requirements presented in Chapter 5 in order to demonstrate a proof of concept for the proposed approach.

The middle-ware was implemented according to the guidelines of the proposed architecture and provides the necessary characteristics for a distributed management of component communication and addresses the autonomic management challenges described in Chapter 2 and Chapter 3. In order to achieve this, the middle-ware includes a model repository, interface channel adapters from the management adaptation layer, a channel management run-time and VSM requirements manager. The implementation can be extended for monitoring of component communication for a additional set of technologies by implementing standard management adapters for the concrete framework.

6.1 Components

The implementation consists of elements which communicate in a distributed fashion without a concrete limitation for their geographical location, as far as network delays do not affect the dynamics of its operation. The communication is achieved through platform-independent protocols for both access to core management functions and delivery of management information (Figure 6.1).

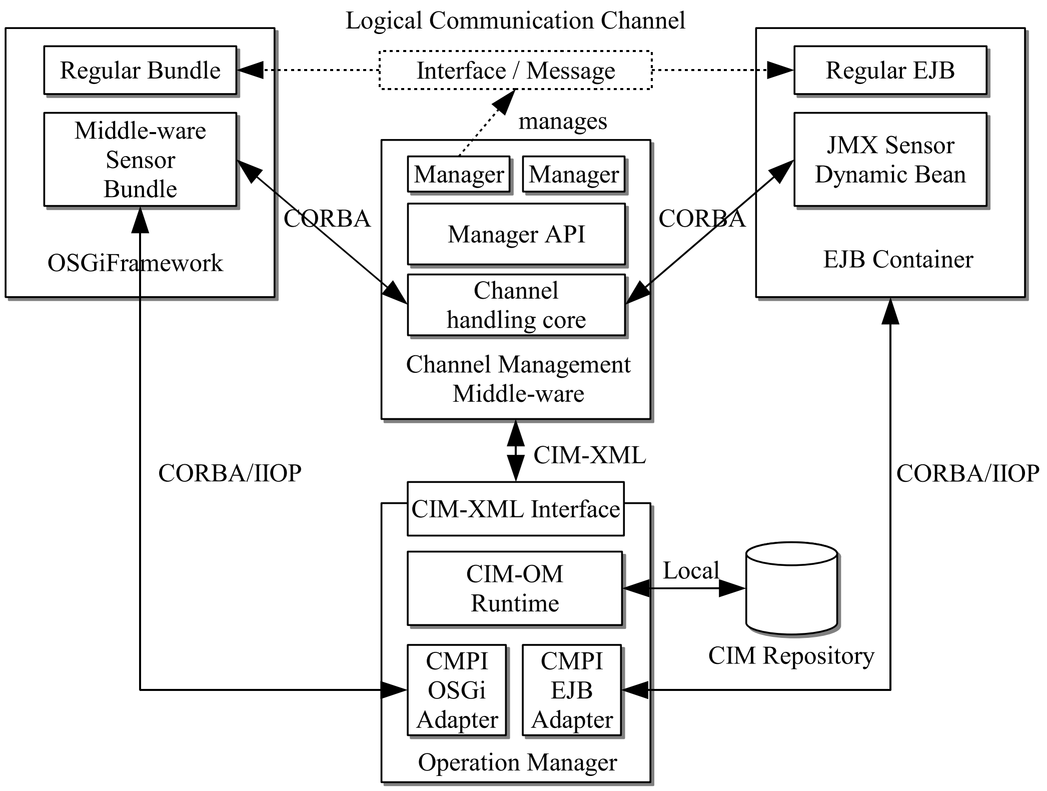


Figure 6.1: Structure of the prototype of the channel management middle-ware. It is a communication hub and channel coordination component in the cybernetically viable architecture. Communication channels which connect its interfaces with the rest of the system are implemented in entirely platform-independent manner for greater extensibility and management capability.

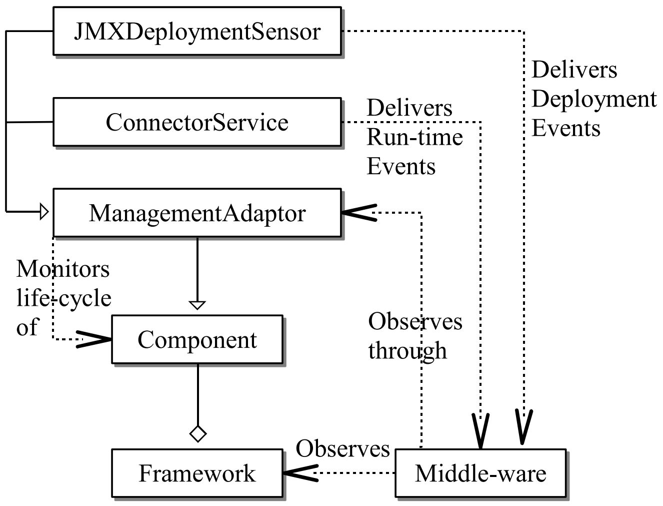


Figure 6.2: Component in Management Adaptation Layer.

Deployment events are delivered to the middle-ware runtime through JMX notifications, while connector services are responsible for provision of connectors and transparent channel representation.

6.1.1 Management Adaptation Layer

The management adaptation layer in the sample implementation consists of a two sets of components that deliver the necessary management information to the run-time. The first group is responsible for communication of events related to deployment of components, while the second delivers information about communication acts in run-time phase.

Deployment information is delivered through Java Management Extensions (JMX) integrated components which are aware of events reported by component hosting frameworks, such as OSGi, EJB or Service Logic Execution Environment (SLEE) Building Blocks (BB) including deployment of component, registration of service, undeployment and unregistering service.

The run-time event sensors are components which either provide wrappers for system interfaces or standalone services with the help of which components declare the necessary meta-data for representation of communication channel handled by the middle-ware run-time.

A component that has the role of sensor has to be able to inspect the state of the framework and the life-cycle or communication activity of other elements. In the case of deployment awareness this can be achieved by registering a *management extension* that translates internal system events to events understandable by the middle-ware. In the case of run-time declaration of intention for communication, the provided services for establishment of connection between the elements have to trigger events or calls understandable by the middle-ware. This functionality is

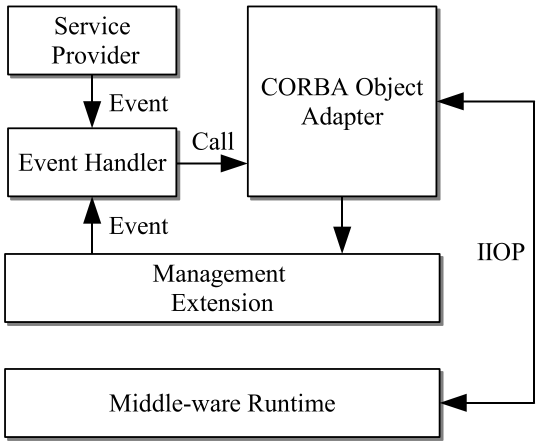


Figure 6.3: Management Adapter Structure.
Events generated by the management extension component and the service provider are delegated as calls to the CORBA object adapter sending them over the network to the middle-ware runtime.

realized with the help of a generic event handler that processes events from both deployment monitor and connector service providers (Figure 6.3). Once the events are normalized and consumed by the handler it delegates the execution of calls to the middle-ware by using the CORBA Object Adapter.

The common event handler provides both generic interface for channel monitoring and singularity in the way the middle-ware perceives channel events, thus simplifying the architecture.

6.1.2 Middle-ware run-time

The central element in channel handling is the middle-ware run-time. It acts as a hub for communication, channel life-cycle support, integration of channel management expertise and model reflection.

It has three main components that support the three aspects of its role - a CORBA Object Adapter for platform-independent communication, a life-cycle state machine for management of the channel's state-transition, a model management client for operations related to element mapping, semantics retrieval and querying in the model-driven management sub-system (Figure 6.4).

6.1.2.1 Object Adapter (CORBA)

The adapter exposes the functions for channel manipulation and event processing as a platform-independent remotely accessible application interface, accessible by deployment and real-time communication sensors. The remote calls are then translated by the run-time to events passed to the state-machine. Platform independence

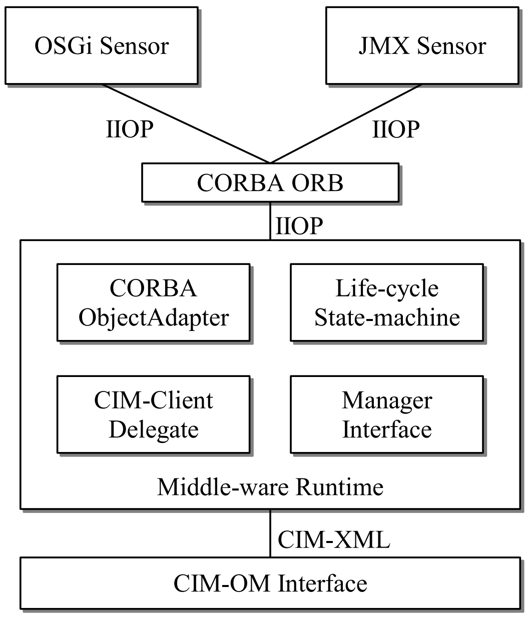


Figure 6.4: Elements of the Middle-ware runtime.

The run-time interacts with sensors over CORBA interfaces, while communication with the operation manager is achieved through platform-independent and standard XML message-based protocol.

in communication is an important requirement for extensibility and multi-framework support. CORBA is a mature platform which focuses on inter-operable communication and allows remote procedure calls between elements implemented on different platforms with a large set of programming languages.

6.1.2.2 CIM-Client

In the process of acquisition of management information from the frameworks, the middle-ware utilizes the standard management platform, implementation of the WBEM specification which provides a management abstraction (see Section 6.1.3) and is physically separated from the middle-ware. The communication occurs with the standard for the management platform protocol CIM-XML. The function of the client incorporates execution of management procedures, subscription for indications, querying using a standardized language, association traversal, creation and manipulation of management instances.

6.1.2.3 Life-cycle State machine

The life-cycle of a channel (see Chapter 5) is managed by a finite state automate that reacts to the raised by the management sensors events, and by internal events, fired by a channel management or the runtime itself. The output of the state-machine is a set of action instructions that are applied by the run-time to affect channel state. The state-machine is the element that coordinates the autonomic management loop.

6.1.2.4 Channel Manager Adapter

The run-time allows selection of a channel management strategy, implemented by a specific manager class. The *channel manager adapter* allows selection on-demand and can be used as an interface for integration of external incident expertise, for example a remotely accessible expert system or knowledge-base for incident management. The prototype includes a static set of ECA rules implemented as a Manager class implementing the VSM requirements policy described in Section 5.7.

6.1.3 Operation Manager

The primary aim of autonomic computing is to provide tools and methods for management of heterogeneity in IT systems. For this purpose a management system has to be able to act simultaneously with management interfaces of different kind. The *Operation Manager* (Figure 6.5) is a re-usable management framework, part of the WBEM set of management components, the role of which for the developed prototype is to provide an abstract object-oriented level and model-driven management on a common base of interfaces. It realizes instance representation of the existing managed elements in the form of accessible and query-enabled knowledge-base implemented using the standard Common Information Model (CIM) specification. A specific positive characteristic of this approach is its open specification and platform-independent communication mechanisms. Another reason for the selection of CIM as information base is that it virtually dominates the market of management solutions being integrated in the major commercial operating systems, such as Solaris, Microsoft Windows, and recently in commercially supported Linux-based solutions. This ensures a potentially good integration and guaranteed homogeneity in the domain of management.

Figure 6.5 illustrates the internal communication flow by arrows. The CIM-OM receives indications and executes management calls by routing requests to the appropriate management adapters which on their turn communicate with the managed elements over CORBA. The CIM-OM is capable of supporting model persistence by utilizing relational database systems, but at the same time to integrate object-oriented real-time state reflection and persistent models.

The particular features that are heavily used in the prototype are the query mechanism *Web-based Enterprise Management* (WBEM*) Query Language* (WQL) (see Figure 6.6) and the platform independent XML-based Remote Procedure Call (RPC) protocol, specifically designed for management operations between *CIM-Client* and *Operation Managers.* In addition, for better integration with graph-based algorithms for semantic interpretation, clustering or others, the framework provides

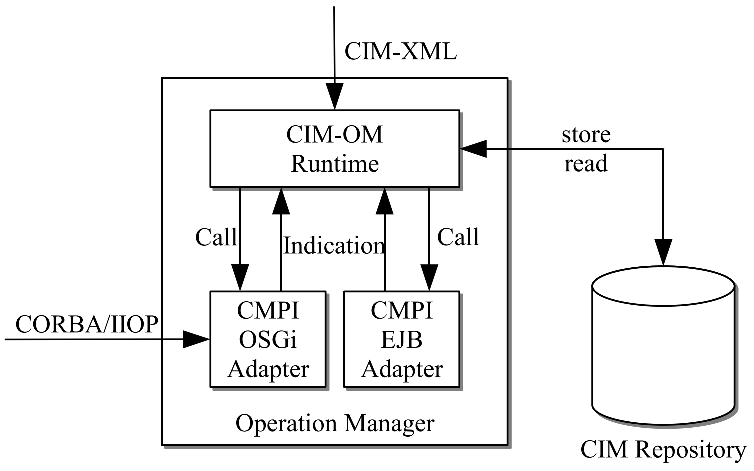


Figure 6.5: Structure of the Operation Manager.
Accessible for queries and graph-oriented association traversal through platform independent protocol. The adapters communicate with the components over CORBA interfaces and provide a reflection of the real world which the CIM-OM maps on-to its model schema stored in the CIM repository.

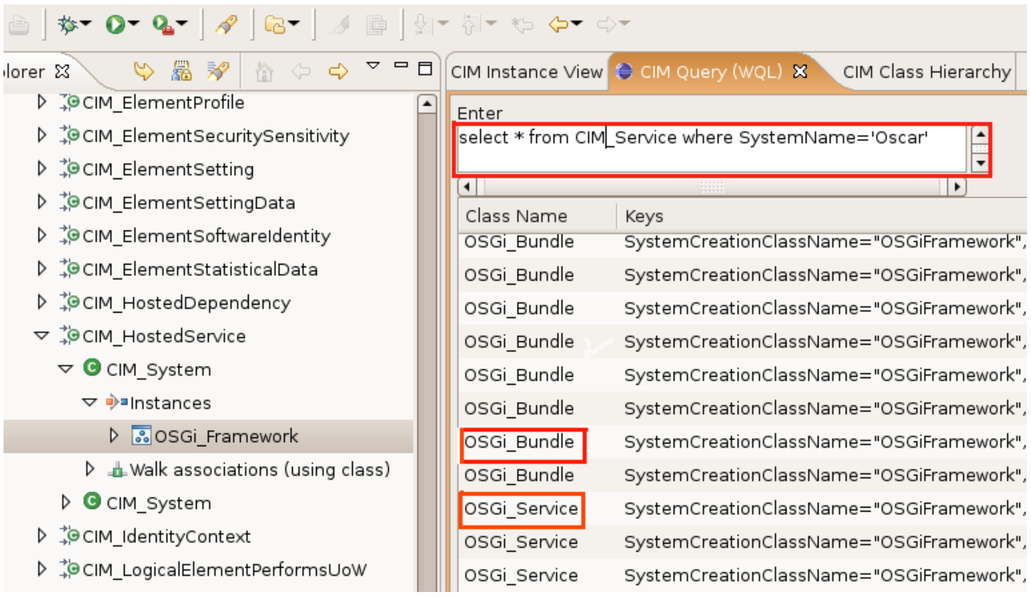
ways to access and traverse associations between managed elements. Every element which has been associated with the CIM model is observable by standard means of the WBEM framework.

6.1.4 Communication Adapters

Figure 6.5 shows that the CIM-OM incorporates and depends on management adapters (OSGi Adapter, EJB adapter). Their single role is to transform the state of the managed elements which they are able to observe into understandable for the CIM-OM form. Every adapter utilizes an embedded CORBA adapter in order to communicate with the management environment. Both component sensors and management adapters deliver management information to the middle-ware, but it differs in the aspect of information usability: the component sensors initiate the management cycle, while the management adapters retrieve an actual environment state. The first is important for the middle-ware run-time operation, the latter for the ability to access complex environment dependencies in a generic manner.

6.1.5 Meta-Model

A notable opportunity opened by the selection of WBEM as a management integration approach is the ability to map and later query reflections of the real state of the management domain. For this purpose the CIM-OM allows extensions of its management base model to be imported dynamically, corresponding management

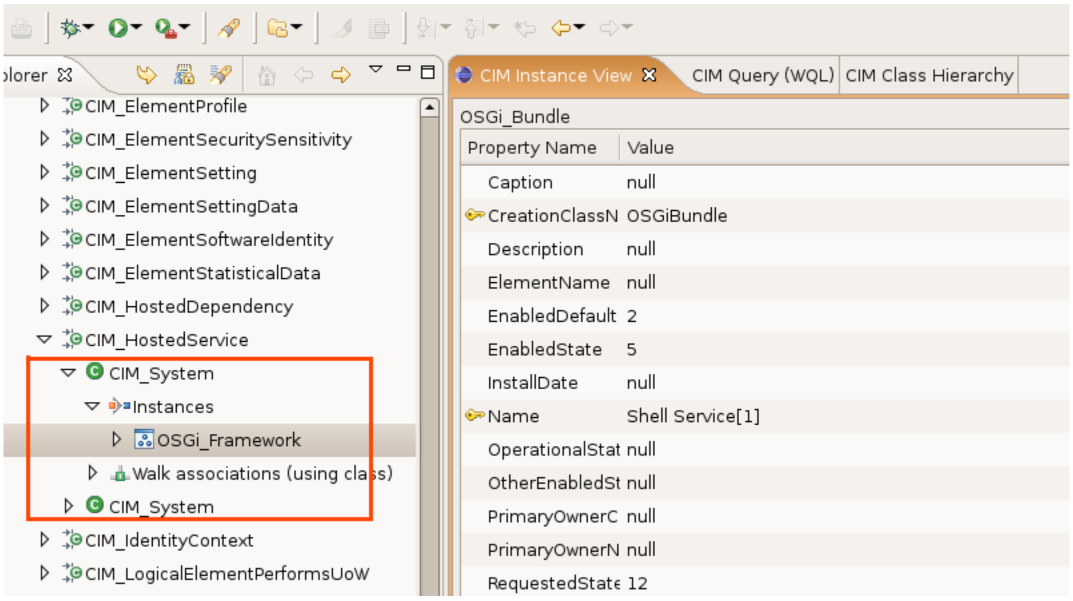


```
CIM_ElementProfile
CIM_ElementSecuritySensitivity
CIM_ElementSetting
CIM_ElementSettingData
CIM_ElementSoftwareIdentity
CIM_ElementStatisticalData
CIM_HostedDependency
CIM_HostedService
  CIM_System
    Instances
      OSGi_Framework
    Walk associations (using class)
  CIM_System
CIM_IdentityContext
CIM_LogicalElementPerformsUoW
```

```
select * from CIM_Service where SystemName='Oscar'
```

Class Name	Keys
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Bundle	SystemCreationClassName="OSGiFramework",
OSGi_Service	SystemCreationClassName="OSGiFramework",
OSGi_Service	SystemCreationClassName="OSGiFramework",
OSGi_Service	SystemCreationClassName="OSGiFramework",

Figure 6.6: WQL query results visualized in CIM-Client User Interface
The result from executing the query is a list of available instances of managed elements.



```
CIM_ElementProfile
CIM_ElementSecuritySensitivity
CIM_ElementSetting
CIM_ElementSettingData
CIM_ElementSoftwareIdentity
CIM_ElementStatisticalData
CIM_HostedDependency
CIM_HostedService
  CIM_System
    Instances
      OSGi_Framework
    Walk associations (using class)
  CIM_System
CIM_IdentityContext
CIM_LogicalElementPerformsUoW
```

OSGi_Bundle

Property Name	Value
Caption	null
CreationClassN	OSGiBundle
Description	null
ElementName	null
EnabledDefault	2
EnabledState	5
InstallDate	null
Name	Shell Service[1]
OperationalStat	null
OtherEnabledSt	null
PrimaryOwnerC	null
PrimaryOwnerN	null
RequestedState	12

Figure 6.7: Accessing data about instances associated with the CIM base.

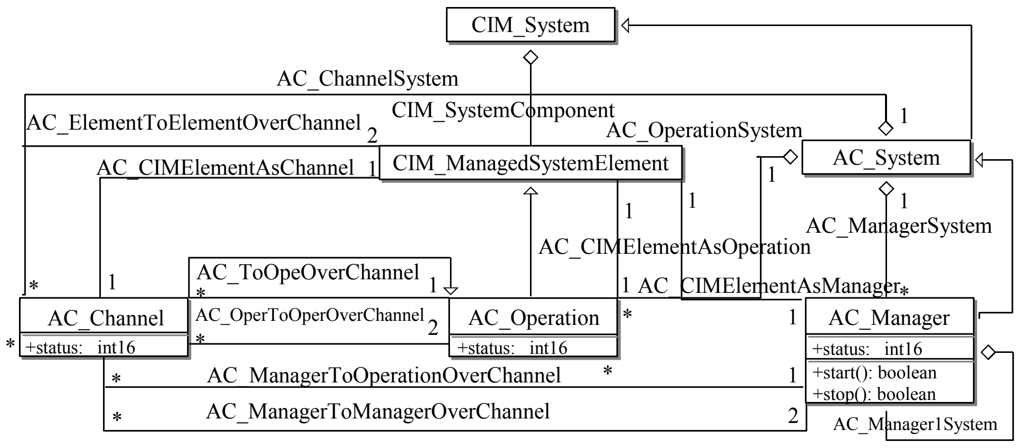


Figure 6.8: The Meta-model.
The adaptation of VSM structural requirements in the form of CIM extension.

adapters to link to them and respectively handle the reflection. As a consequence this functionality introduces the ability to model and later import the discussed in *Chapter 5* meta-model as extension of the rich management base and link it dynamically to already existing management elements.

6.1.5.1 VSM Adaptation

An adaptation of the structural requirements of VSM has to sufficiently provide means for modeling of recursive systems, layers of management and a descriptive mechanism for definition of communication channels, as perceived by the cybernetic definition in VSM. The developed meta-model incorporates these features into a single CIM extension in order to allow easy and standards-based means for augmenting existing management base (Figure 6.8). This approach converts the introduced CIM-OM capabilities into advantages for the possibility to dynamically link existing managed systems and elements without any modifications in their already deployed CIM representation or management adapters.

In this case reasoning about the managed system is determined through analysis performed by the channel management framework on the current state of the environment and semantic relations of the elements with the introduced adaptation.

Figure 6.8 represents the complete CIM extension as it is imported in the MO. *AC* stands for *Autonomic Computing* and is the prefix for CIM and implementation classes in the prototype. The CIM model is capable of describing recursive structures by using the concept *AC_System* and *AC_Manager*. While the first is the base class aggregating manageable elements in a VSM-like system, the second extends with the notion of management. At the same time it is a regular System as defined by CIM. The resting two extending classes *AC_Operation* and *AC_Channel* are accomplishing the model by introducing respectively the the notion of final element of system containment - the operations, and the medium for information transfer - the channel.

In order to retain relation to the existing management base, the model defines relations that enable description of communicating external for the system elements. The association *AC_ CIMElementAsOperation* and *AC_ CIMElementAsManager* help in identifying of depending Manager-Operation pairs in the existing set of managed CIM elements using the notion of communication channel. The association *AC_ElementToElementOverChannel* is the associating relation for the purpose of identifying any dependency between two CIM elements. The model does not define explicitly the way two end-points declare their variety - required and provided, but for this purpose the prototype uses the already defined by the CIM model way for association of software element identification *(CIM_ SoftwareIdentity and CIM_ Dependency).*

6.1.5.2 System State Adaptation and Access

The meta-model is used by the CIM-OM as a reference to the associations between elements in order to narrow the selection of entities from which the CIM-OM acquires the real state of the managed environment. The information must be relevant and up-to-date. For this purpose the CIM-OM uses a specialized interface through which it calls native functions responsible for the aggregation of the data - the *Common Management Protocol Interface* (CMPI). Modules which implement this interface are called management providers and can be loaded dynamically or statically in the run-time environment and register themselves to serve specific sets of imported classes in the repository and provide in real-time the current state of the observed managed elements. The implemented management providers are able to retrieve this information on demand by the CIM-OM, or to act independently and send notifications to it, signaling about particular event of interest.

In the process of query processing the CIM-OM takes into consideration the responsible for every class management provider and delegates the task of retrieving the information in the correct sequence, with the appropriate protocol or any necessary interactions, for example aggregation of data and normalizing the presentation. The developed prototype utilizes CMPI to deliver information about the component states and their hosting frameworks by a connection between the providers and the management interfaces with the help of CORBA interfaces.

This creates a standard communication mechanism for a large variety of frameworks and depending on their location (remote or local) can be adjusted accordingly to optimize performance for local or remote interaction. The concrete implementation is a CORBA adapter for the JMX interface and opens the opportunity by using a single interface to interact with all component technologies which use JMX as management platform, such as J2EE, SLEE or OSGi with the help of JMX-enabling bundles.

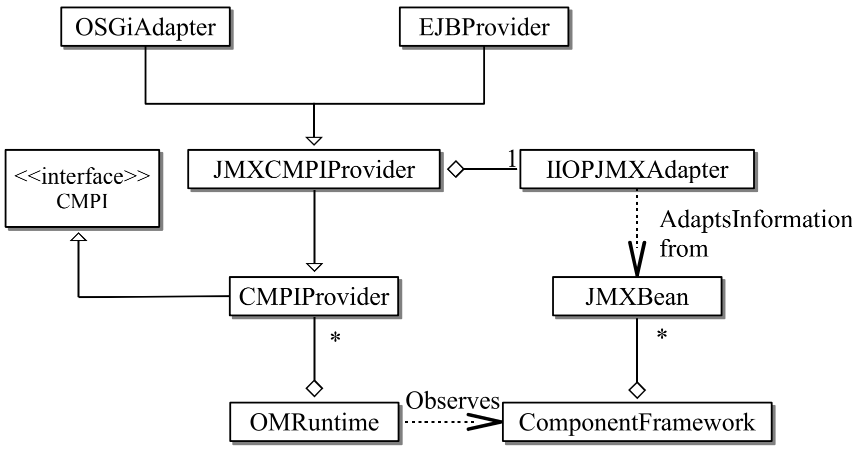


Figure 6.9: Adaptation of the environment state.
An operation manager is able to observe the state of a component framework with the help of management providers that utilize a CORBA-JMX adapter.

6.2 Extending The Middleware System

The middleware implements a system that may be integrated either locally or with distributed components communicating over a network connection. It allows integration of new elements for extension of its spectrum of management. Extensible aspects are:

- Management adaptation layer - the number of management adapters is not limited to the implemented set of communication handlers. The new elements are not required to be pre-registered for their operation because they implement a common channel activation interface.
- Framework runtime - the specific way that a manager handles the communication establishment and reasoning can be specified by implementation of the manager's interface which assures interoperability with the framework handles for manager interaction.
- CIM-Modeling - the operation manager ensures a flexible, dynamic registration of new classes in the information base. Any new extension schema related to new managed frameworks are not strictly bound to the management meta-model and can exist standalone, if required.
- Management Providers - the set of management providers can be easily extended by just plugging them into the CIM-OM environment (dynamically or statically). This is implicitly defined by the management framework.

The system is extensible with modules by integrating new elements on every level of operation (Figure 6.10) - management adaptation layer is a loose set of adapters,

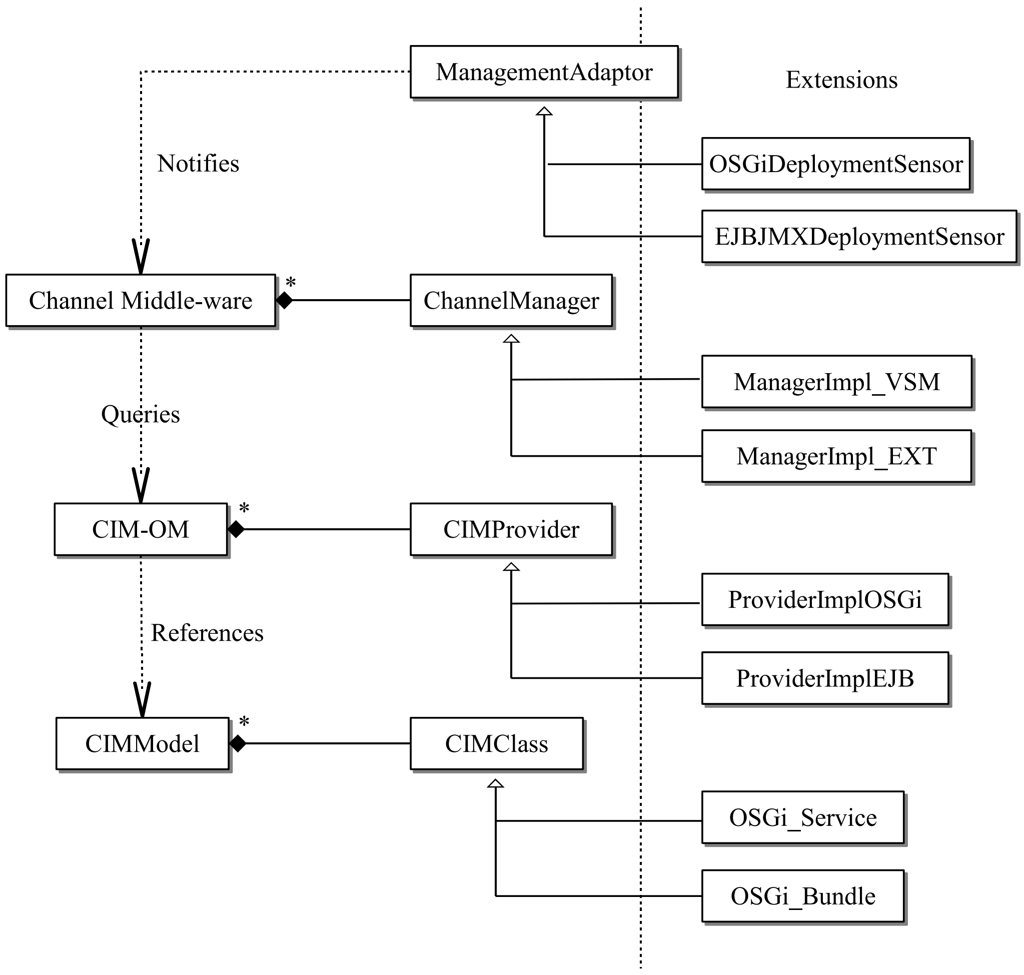


Figure 6.10: Extending the system.
Extending the system on every level (left) of operation is a matter of adding new elements (right) and registering them with the corresponding layer.

not related to each other, the communication managers provide different strategies for handling communication and can be selected according to the specifics of the communication. Pluggable CIM management providers are responsible for the perception of the state of the environment, while the set of imported into the CIM repository classes is extensible by specification (WBEM).

As it can be seen, the prototype allows extension of its functional richness of the system without changing its architecture and breaking its self-description mechanism. Following the architectural approach will ensure the ability to map itself using the meta-model.

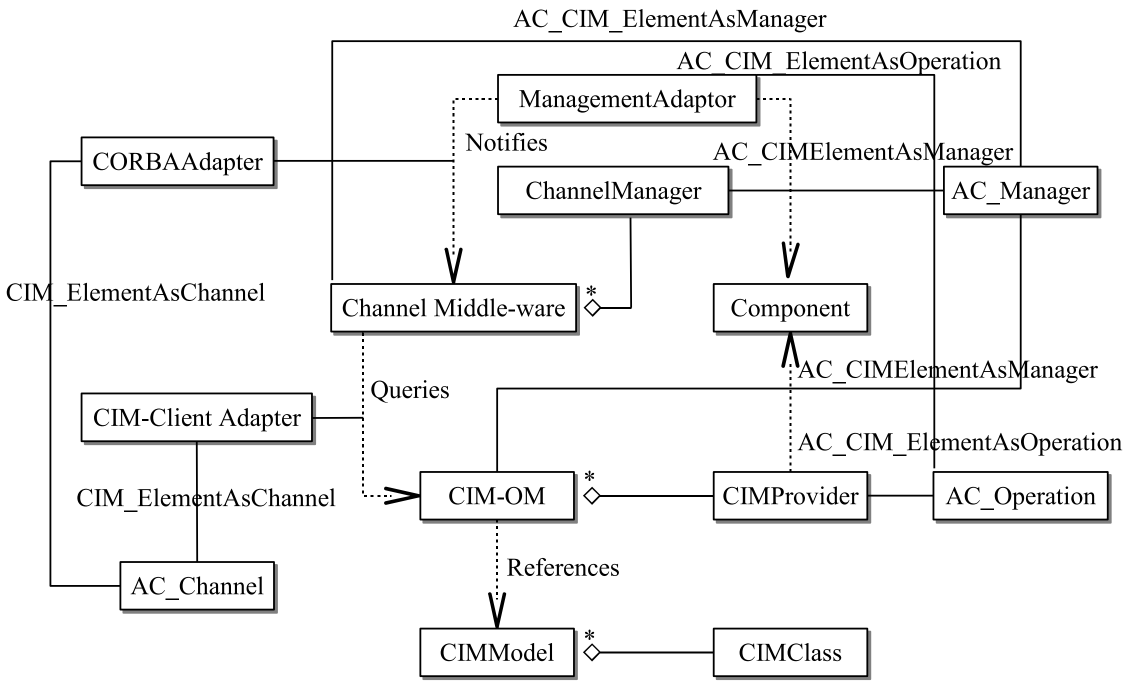


Figure 6.11: Prototype elements mapped on the VSM meta-model.

Associations AC_CIMElementAsOperation and AC_CIMElementAsManager help for treating the framework as Operation Manager, channel manager as Manager3 and the communication between them is expressed as event and interface channels.

6.3 VSM Mapping

The described prototype of a management middleware is capable of monitoring interactions between software components that are not part of itself. For a full self-management the middleware system has to be able to recognize its own elements with the same set of management routines used to monitor the rest of the system. The only missing part for this fulfillment is a proper mapping of the elements of management system with the help of the CIM model. Once their relations to the management meta-model have been defined the middleware is capable of providing communication details related to the management operations to the respective managers. This enables discovery of potential inconsistencies in the communication between management adapters, middleware and operation manager.

6.3.1 Structural Map

Figure 6.11 illustrates the relations between the elements of the implemented architecture and the VSM meta-model used by a manager to reason on the compositional aspect of the system. The middleware is associated as a containing manager (System/Operation Manager) for its role of interaction and management of sensors.

The managers belonging to it are associated as Manager3 for their role of higher management function (System/Manager3). The operational set of VSM elements is associated with the sensors and management providers which are the final point of interaction of the management framework with the system under control.

6.3.2 Communication Map

The three components communicate using either CORBA interfaces or CIM-XML. For the purpose of communication management inside the management middleware itself these elements provide meta-data in the form of statically defined fields submitted to the framework and evaluated by a manager using the channel API described earlier, with a rule-set implementing no concrete correction actions, but mainly notification mechanism for reporting of inconsistencies. The CORBA adapter responsible for invocation of remote channel middleware interface and the CIM client adapter are mapped with the association AC_CIMElementAsChannel to indicate an actual instance of a channel between the components.

Chapter 7

Results

The Viable Software Architecture for Autonomic Management of Distributed Heterogeneous Component-based Systems has been implemented in the form of prototype described in Chapter 6. This chapter overviews the results gathered in the process of its development and deployment.

The introduced management model, software architecture and communication middleware create a new concept for autonomic management. Advantages of this new approach are the consistent model for system organization management, transparent dynamic interfacing for component communication and self-aware architecture. A key feature of the demonstrated approach is the support for common communication patterns implemented with standards-based components.

The proposed architecture satisfies the management requirements of real-world scenarios with distributed components and addresses the challenges of the autonomic management of software components.

7.1 Referencing Real World Scenarios

The presented in this dissertation model, architecture and middleware for autonomic software component communication targets the development of a concept for management of existing mixed component-based environments.

The developed middleware has been designed to operate with the popular component technologies OSGi and EJB to refer to Smart Home real-world scenario for solution developed at Siemens AG, CT and a custom container for knowledge autonomous knowledge acquisition developed in the frame of Sfb 374 as a technology component of a scenario for Distributed Product Development System. Table 7.1 presents a map of those technologies related to the target scenarios.

The application of the architecture in two completely different and separated domains of application is a demonstration of its flexibility with both existing and custom implementation of component technologies. The communication in both sce-

Technologies	Smart Home	DPD
OSGi	Gateway	Remote Interfaces
EJB	Back-end	Knowledge-base
Software AgentContainer	-	Communication Framework
CommunicationProtocols	Web Services, Local Interfaces, Remote Interfaces	Protocols Interfaces, Custom XML, Local Interfaces, Remote Interfaces

Table 7.1: Real-world scenarios and used component technologies.
Every scenario involves at least two communication protocols, making them complex to manage and support.

VSM Mappings	Number
Number of System 1 Elements	8
Number of System 3 Elements	2
Number of Management Channels	3
Number of Transducers	1
Number of Mapping Associations	11

Table 7.2: Architectural elements of the management adaptation in Smart Home Scenario

narios involves at least two different communication protocols making both systems distributed and exhibiting the problems of overhead in component deployment management, management granularity, parallel component evolution, deep component dependencies and management complexity overhead.

7.2 Smart Home

The investigated solution for Smart Home was developed as a concept at Siemens AG, CT. Particularly interesting characteristic of the adopted architecture is its modularity and total integration of user interface devices, platform services and third party services in a single domain of management. Its configuration was a good use-case for a heterogeneous distributed system that needed a higher level of management support that integrates cross-container dependency management. The environment had to provide services running without interruption to a potentially large number of end-user subscribers and back-end service provision to third party services. This scenario offered the possibility to verify the ability of the VSM management model adaptation to describe the existing assets of a completely distributed, service-oriented, component-based system and had directive role in the development of the management adaptation layer.

The management adaption of the Smart Home solution consisted of sixteen map-

Framework Aspects	Adaptation Method
OSGi Bundle Deployment	No change
EJB Deployment	No change
OSGi Monitoring	Adapter
EJB Monitoring	No change
Bundle Communication	Resolver Channel Adapter
EJB Communication	Interface Channel Adapter

Table 7.3: Adaptation methods for Smart Home scenario

Types of Instances	System 1	System 3	Transducer
OSGi Framework	x	x	
OSGi Service Resolver	x		
EJB Container	x	x	
EJB Interface Resolver	x		x
OSGi Framework Monitor	x		
EJB Framework Monitor	x		
OSGi Bundle	x		
EJB	x		

Table 7.4: Mapping of Smart Home elements

ping relations in total, having three container systems (EJB, OSGi, .NET), three points of variety transduction and three management channels, respectively sensors (Table 7.2).

As it can be seen on Table 7.3 and Figure 7.1, the adaptation of the solution elements for integration of the management system requires little or no modification of the existing component technology. Deployment monitoring utilized the services already available in both OSGi and EJB to deliver information about the internal state of the component life cycle.

For the purpose of dynamic meta-data declaration in actual component communication the service and interface resolvers are implemented as proxies. However proxy-based service resolving is a trivial approach in development of distributed systems, which virtually does not affect the way of development of new applications.

7.3 Distributed Product Development Systems

The system serving as a reference for a modern distributed product development support was implemented in the frame of Sfb 374 - Innovative Solutions for Rapid Product Development, at the Institute for Product Development Support Systems, IRIS, University of Stuttgart. The configuration of the system included a set of completely distributed elements, based on component frameworks, including an in-

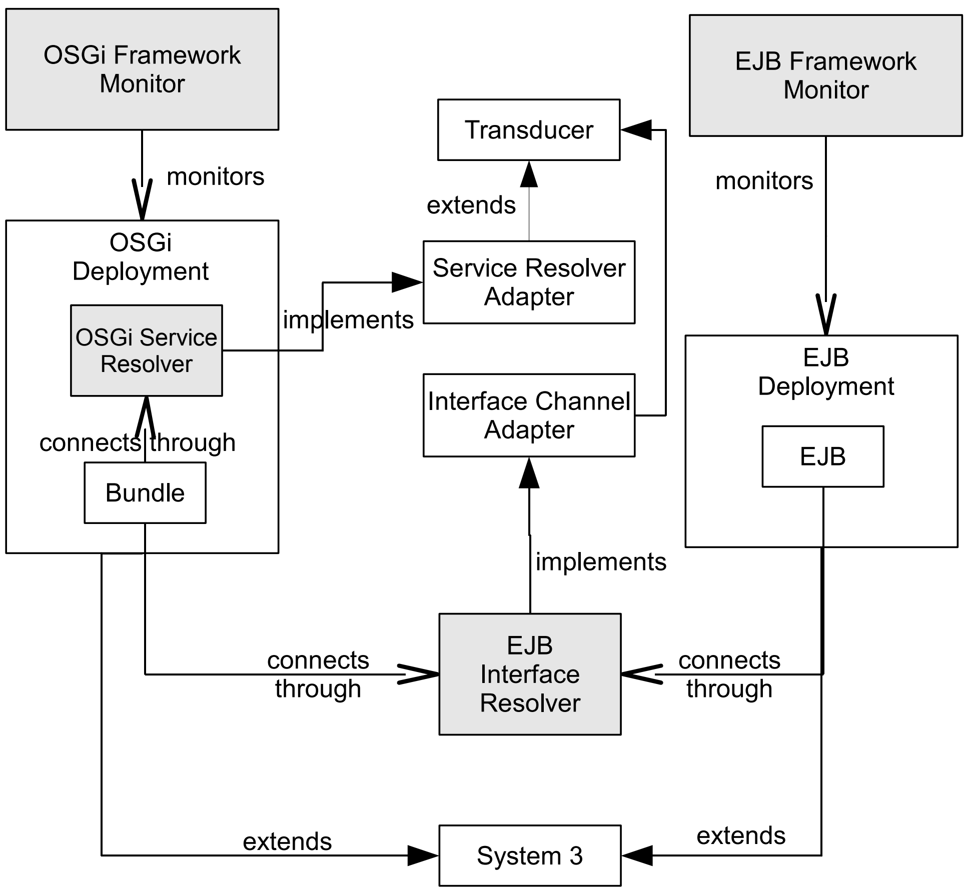


Figure 7.1: Smart Home Mapping Associations

VSM Mappings	Value
Number of System 1 Elements	10
Number of System 3 Elements	3
Number of Management Channels	3
Number of Transducers	3
Number of Mapping Associations	16

Table 7.5: Architectural elements of the management adaptation in Distributed Product Development Scenario

Framework Elements	Adaptation Method
Agent Deployment	No change
EJB Deployment	No change
MAS Monitoring	No change
EJB Framework Monitoring	No change
Agent Communication	Message Channel Adapter
EJB Communication	Interface Channel Adapter

Table 7.6: Adaptation methods for distributed PDP scenario

ternally developed communication platform with autonomous components. This scenario was particularly interesting for its decentralized communication flow and because it served as a reference to the ability of the management architecture to support in-house developed component frameworks.

Additionally the knowledge-base implemented as EJB container had its own meta-model for description of RPD process participators and resources. This feature opens the opportunity for the meta-model to be able to monitor not only software-elements, but representations of real-world process entities appearing as instances of persistent EJB components.

The management adaption solution consisted of nineteen mapping relations in total, including three container systems (EJB, RCP, MAS), three points of variety transduction and three management channels (Table 7.5, Figure 7.2).

As it can be seen on Table 7.6, the adaptation of system elements for integration of the management system required little or no modification of the existing component technology. Deployment monitoring utilized the already available services of both Agent Container and EJB server to deliver information about the internal state of the component life cycle. In the case of software agent framework, monitoring was facilitated by the multicast environment and implemented as first-class element in the network - a monitoring agent. The same approach is applicable with any FIPA-compliant software agent framework, for example the popular JADE. For the purpose of dynamic meta-data declaration in actual component communication the RPC clients of agent calling remote EJB methods are implemented as proxies.

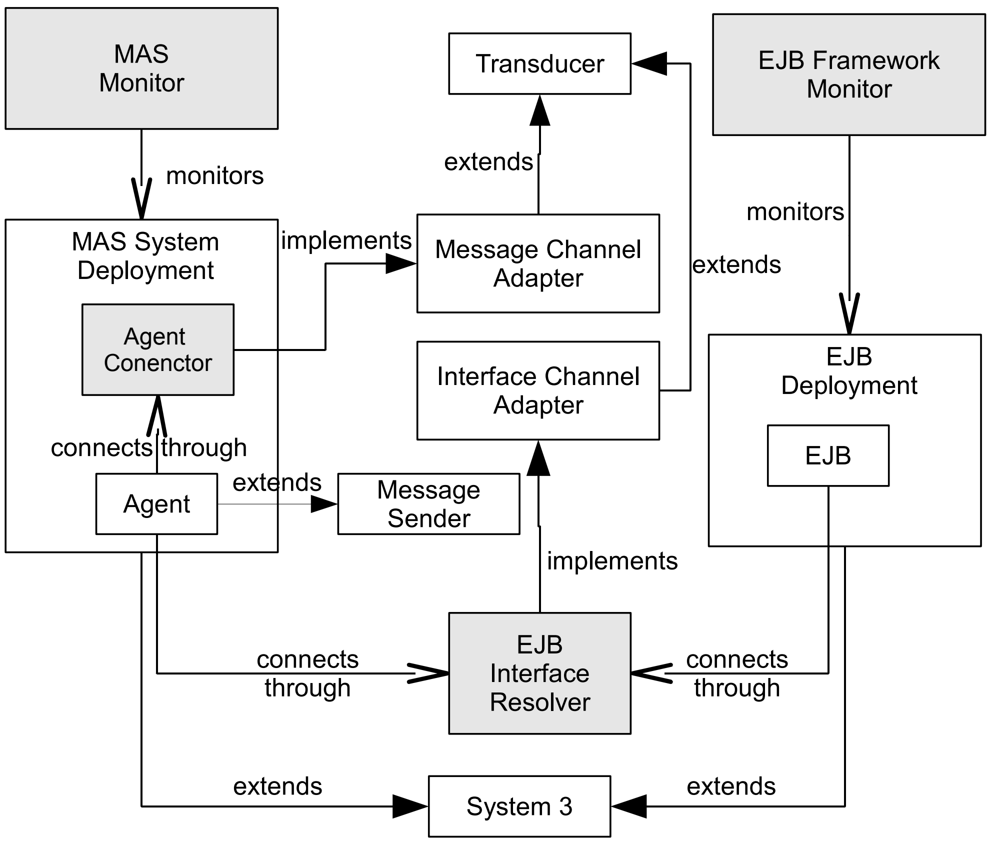


Figure 7.2: Distributed RPD MAS mapping associations.

Types of Instances	System 1	System 3	Transducer
Agent Deployment	x	x	
EJB Container	x	x	
RCP Platform	x	x	
RCP Module	x		
EJB Entity	x		
EJB Interface Resolver	x		x
Agent Connector	x		x
Module Interface Resolver	x		x
MAS Monitor	x		
Agent Entity	x		

Table 7.7: Mapping of DPD Elements

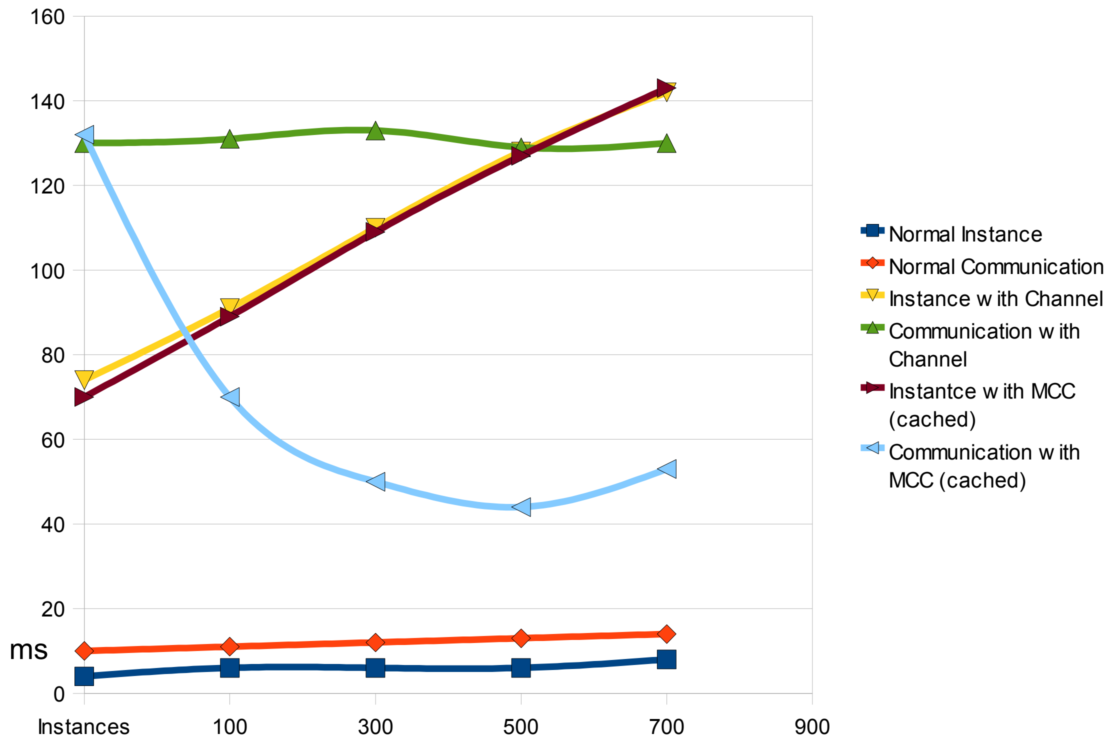


Figure 7.3: Channel Performance Measurement

7.4 Performance

The concept of communication management introduces unavoidable load for the operating environment. In order to see the dynamic behavior of communicating endpoints with MCC, measurements have been conducted using local objects, communicating over Interface Channel Adapter, the implementation of which is based on a dynamic proxy approach, using matching interface delegator. This approach involves reflection, which is known for being a not high-performant method, but still a consistent, predictable and sufficient for the purpose of investigating the dynamic communication behavior. Communication between local objects was selected in order to avoid network traffic fluctations which may introduce undesirable noise.

There are two stages which are important for the operation of a component during its life-cycle - its creation and the establishment of connections to other component interfaces. These two stages have been tested with a set of 700 instances of objects in three modes - pure instantiation and communication (no middleware involved), instance creation and communication assisted by the channel framework (without caching resolved and instantiated elements) and a caching version of the channel runtime. Figure 7.3 shows the results.

As it can be easily noticed, the channel assisted interaction is significantly slowed down, largely due to the ineffective proxy interaction. However, introducing a simple hash-based instance cache mechanism reduced the average time of making an interface call. The linear growth of time for creating instance is due to the increasing pool of deployed compoennts, which naturally increases the time for creating or finding a reference to a managed component. The cache mechanism in this measurement is useless, as the number of cached elements equals the number of instances. These results clearly illustrate the existing trade-off "flexibility-performance". Increasing performance using managed communication is a subject of further research.

7.5 Management Model Adaptation

A key motivation for building autonomic systems is the desire to reduce complexity and heterogeneity. The existing management systems failed to provide adequate solution for this aim. On the opposite, the implemented adaptation of VSM as a management model with communication and architectural requirements provided a unified and sufficient basis to build management support for those systems. Its ability to describe complex distributed relations in deployment scenarios with popular in the industry component frameworks has proved its usefulness and viability as a reference model. Despite of its relatively compact set of structural elements the adaptation managed to facilitate the modeling of the implemented management middleware with no changes expressing the target autonomic requirement for self-awareness and homogeneity. The model can be extended and mapped for a larger variety of application exhibiting recursive structural organization and management hierarchies.

The developed implementation illustrated the usefulness of the channel notation by being able to express local interaction and distributed communication and to facilitate verification of communication consistency for external managed elements and with its own internal requisite knowledge and requisite variety requirements without having the need to introduce variations of dependency management abstractions or mechanisms. This characteristic confirms the sufficiency of the model to enable dependency observation, management strategies and transparent communication handling of distributed elements. The implementation of the model's constraints for variety management and dynamics proved to be useful in dynamically changing environment of evolving components. The tested scenarios included multiple deployment different development snapshots of communicating OSGi bundles and EJB components as a scenario for evolving communicating system and the adaptation of the VSM principles for organization were sufficient for expression of component containment and distribution of variety. Further more the adaptation handled successfully with no needed for additional modification or editing the internal management elements variation resembling possible variety in strategies or version of management interfaces. These results illustrate the validity of the VSM management constraints and their adaptation for software component management.

The three basic aspects of the model showed have sufficient characteristics for modeling of autonomic management systems for component-based software in heterogeneous environment.

7.6 Architecture Implementation

The prototype described in Chapter 6 implements the architecture for autonomic management of component-based systems which served as a reference management environment addressing the challenges in autonomic management of component-based systems. The aim was every challenge as described in Chapter 1 to be addressed by the abilities of the architecture and its respective implementing elements in the developed prototype.

7.7 Effect on Autonomic Management Challenges

The *overhead in component deployment management* is usually manifested by the need to have knowledge about the component life-cycle and dependency definition, thus human management has higher probability of fault decision. The implementation of a per-framework deployment monitor and the normalization of the meta-data reduces this risk. The concrete elements responsible for this process in the prototype are the JMX-based monitors which receive notifications from the OSGI and the EJB containers and deliver the information to the middleware by means of normalized interfaces implemented with CORBA-based communication.

Management granularity is a problem mainly triggered by the heterogeneity in management interfaces provided by the vendors of the deployed frameworks. The prototype (respectively the architecture) addressed this problem by a adaptation of the management information from the JMX, OSGi and custom XML-messages to calls in an unified interface for monitoring of communication activity. This interface is implemented by the middleware. Additionally for the purpose of environment state monitoring and not only communication, the middleware includes a standard component for normalized management information querying - the CIM Operation Manager. This eliminated the need to separately monitor the state of the OSGi framework, the EJB server and the Agent container and manually aggregate the needed information.

The problem of *parallel component evolution* is inevitable for it being a major characteristic of component-based software development. Components re-usability is a main economic factor driving encapsulation of functionality and its re-usage in third-party applications. To address the problems related to evolution, semantic and communication incompatibility the architecture uses the notion of variety without a concrete semantic about the "type of variety" transmitted between two communicating components. A higher-level of delegated management for specific decision based on concrete interpretation determines the selection of communication interface and optional adaptation element. The implementation of this mechanism in the prototype is in the form of separation of channel management middleware and delegation of decisions to a selectable manager. This configuration brought flexibility in the selection of decision making algorithm and action sets for specific cases of communication for dynamic interface selection. For example the self-monitoring feature was implemented with the help of a separate manager with reduced action set, while external system management could be implemented with a complete set of response action for channel set-up. Decisions about communication to deployment of components with variable versions were brought to a specification of a set of rules, instead of hard-coding them in the components. The architecture allowed to still follow the re-usability principle of encapsulated code, but helped in avoiding the drawback of components evolving separately.

A drawback of introducing management mechanisms and automated monitoring is the additional amount of code and respectively complexity needed to bind managed elements with the management infrastructure, hence the *management complexity overhead*. Indeed, currently developed autonomic management solutions are complex in both architectural and implementation aspects. The proposed architecture allowed implementation using mainly standard approaches for communication, both for the purpose of mature interoperability and for support of existing development community. The prototype proved to allow integration with minimum amount of intervention in the existing component-software configuration. The adaptation layer allowed to develop independent sensors and effectors that used the existing management capabilities of the frameworks under control, thus reduced complexity on the level of application infrastructure and respectively the business process implementation. Integration of a standard management framework, such as CIM-OM provided an even more open and flexible solution for easy extension and implementation of management tools, for example GUI consoles or policy programming and deployment.

The developed concept of managed communication channels applied with the VSM-adaptation proved to provide much greater flexibility in the definition of dependencies between elements on different levels of containment. Current dependency models do not take the location of a component inside the hierarchy of containment as necessary preposition for reasoning, creating a large graphs of dependencies without semantic charge for concrete domains and sub-domains of dependency relevancy, thus component dependencies are major hurdle in management of complex systems. The architecture proved to facilitate semantically charged dependency management by using the value VSM adaptation as hierarchic and recursive model. Dependencies can be defined on the levels of servers, frameworks, components and services creating different views/layers of dependency chains which allows for easier localization of communication problems.

7.7.1 Trade-offs

The implementation of the architecture, however, proved to exhibit a traditional for the automated management trade-off - the larger the system, the more information about the system has to be provided before its actual deployment, respectively a longer testing phase. Initial difficulties in implementation are rewarded by further automation of the management routines. This was manifested during the implementation of the prototype, which required inclusion of numerous specific adapters, extension models and specialized code for monitoring the state of the frameworks. Additionally, because of the heterogeneity and the need for normalization, the implementation includes code implemented in several programming languages. However this drawback does not affect the initial target of handling management complexity overhead in aspect coupling of management with managing elements.

A second trade-off was demonstrated in the implementation of the prototype - a higher level of abstraction tends to slow the operational performance in favor of greater flexibility and independent components.

The prototype manifested the ability of the proposed architecture to address the problems of overhead in deployment management, granulated management, parallel component evolution, management complexity overhead and deep component dependencies by a relatively simple management concept and standard communication and management components. The major drawbacks of the system proved to be the traditional automation-complexity trade-off and the slowed reaction during deployment process and initial resolving of target communication channel variety, the latter being a subject of further research and development.

Chapter 8

Conclusion

This dissertation focused on the problems of modern distributed software management, scenarios, existing architectures and methods for development of self-managed systems (Chapter 2 and Chapter 3) and proposed an architecture for autonomic management of distributed component-based software environments (Chapter 4 and Chapter 5). Chapter 6 described an implementation of the architecture in the form of distributed monitoring and communication management middleware followed by the results of the adaptation in Chapter 7. This chapter summarizes the approach and results, outlines the relevance of the contribution and discusses the approach limitations as well as potential directions for future development.

8.1 Approach Overview

Distributed software exhibits a growing complexity in management of communication and deployment of individual components. The major difficulties that follow as a result of this growth are reflected in higher probability of human administration error due to one of the five challenges that this dissertation addresses: deployment management overhead, deep dependencies, parallel component evolution, management complexity overhead and management granularity. As a starting point of the research, this work examined the available concepts for distributed management and autonomic computing approaches as a main direction for reducing human error caused by system complexity. The approach that this work used to address the challenges and minimize human-error is based on adaptation of the mature management organization model, The Viable System Model (VSM), and a software architecture that reflects its requirements as a foundation for a consistent system management model in both structural and dynamic aspects. A communication management middleware based on the adaptation and the architectural approach helped to examine its benefits and drawbacks and prove its applicability in heterogeneous component-based environments with the improvements addressing the autonomic management challenges. The scenarios of Smart Home and Distributed Product Development

Systems were target domains exhibiting software heterogeneity and distributed communication. Both domains had different use cases and target domain of operation but shared the concept of component encapsulation. The implementation of the middleware satisfied the management demand and confirmed the applicability of the introduced concept for managed communication. Main drawbacks of the system is the relatively complex implementation due to the requirement to serve heterogeneity and its impact on dynamics metrics in both deployment and communication aspect.

8.2 Relevance of Contribution

The presented in this dissertation research contributes to the research field of autonomic management with a consistent management model that suits the goals and vision of Autonomic Computing - heterogeneity, complex system management, decreasing of human error. In aspect heterogeneity management this thesis contributes with a common concept for communication management and a set of assertions for communication health. A key advantage of the concept is the added value of evolution semantics in communication between decoupled components.

The field of complex system management will benefit from the adaption of the Viable System Model. Current autonomic frameworks miss to provide a consistent complexity management model designed especially for this purpose. Beside the IT oriented adaptation, the model is extensible on higher levels of IT operation and integration potentially allowing even higher order of management on the level of business processes and project management.

The work supports the effort to decrease human error in IT operation by utilizing the proposed model for automation in management of component communication. The middleware implementation provides a component-based approach to life-cycle management of communication channels on the abstract level, potentially opening the possibility for integration of communicating elements that were not previously known or intrinsically defined in the management framework.

8.3 Limitations

There are some limitations that this architecture in its current state of research does not address. The first of them is due to its focus on component-based architectures. Heterogeneity in real-world applications implies deployment of software, which does not strictly follow the concept of encapsulation and life-cycle management. There are cases in which a system may be "patched" by a standalone piece of code which does not have the notion of component, meta-data or container, but simply provides functionality. These cases are not covered in this work in both aspect management model and architecture.

The second limitation which the presented architecture has is related to the ability of the middleware to inspect component containers that do not have integrated management interface. While the notion of component-based sensor (deployment or dynamic) is convenient on the level of component communication, the management of component containers is highly dependent on the ability of the specific component framework to inspect component state and provide this information through a management interface. While this limitation does not affect currently available and popular component technologies, there are cases in which in-house developed frameworks are difficult to adapt to this requirement.

The large variety of frameworks requires a larger variety of sensors - dynamic communication and deployment sensors. Although the architecture is not dependent on the concrete way of sensor implementation and provides common interfaces for meta-data delivery, a larger system with higher degree of heterogeneity will require a larger set of implemented sensors which determine the quality of management feedback, thus the requirement for rich set of management sensors. This limitation is a manifest of the law for requisite variety in control systems and can be overcome by common development practices following the ideas behind management architecture frameworks, such as JMX. The research presented in this dissertation did not focus on performance of management, thus the architecture may introduce limitations on improvements in aspects deployment and communication establishment speed. Variable performance impact is due to the overhead of communication wrapping, management queries, synchronization and meta-data processing.

8.4 Future work

This dissertation introduced a novel approach for autonomic heterogeneity management for the domain of distributed computing. The domain of Distributed Product Development and systems for home automation were investigated. It addressed the five challenges related to automated management of component-based system and illustrated successful implementation of the proposed architecture but because of its architectural focus and it still needs improvements in several aspects.

8.4.1 Research in optimization of performance

Every management system adds a degree of overhead affecting the performance. The proposed architecture needs optimization of the mechanics for evaluation of channel requirements. Possible approaches that may address performance is the adoption of caching algorithms for faster channel resolving and channel establishment hints for pre-caching of channels in relation to the announced deployment dependencies in the meta-data declarations of the deployed components.

8.4.2 Research in integration of non-component logic

Component-based application development is not always the choice of software vendors. Sometimes, in-house development produces standalone solutions that does not always fit any component model. Additional research effort is needed to outline the possible ways for encapsulation of in-house developed solutions that do not adopt component-models, or are difficult to be wrapped in component interfaces.

8.4.3 Better tools for monitoring

System management is highly dependent on accessibility. The administration tools which a system provides often determines the ease of use and the degree of configuration errors. The proposed approach provides an excellent opportunity to build rich administrative applications for visualization of component dependencies, component communication activity, VSM requirements satisfaction and dynamics metrics.

Bibliography

[1] Agarwal, M., Bhatt, V., Liu, H., Putty, V., Schmidt, C., Zhang, G., Zhen, L. and Parashar, M., *AutoMate: Enabling Autonomic Grid Applications*, In: Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop (AMS2003). Seattle, WA, USA (2003) pp. 48-57, IEEE, 2003

[2] Agrawal, H., Horgan, J.R., Krauser, E. W. and London, S.A., *Incremental regression testing,* In Proceedings of the Conference on Software Maintenance – 1993, pages 1–10, 1993

[3] Antoniu, G., Bouziane, H. L., Breuil, L., Jan, M. and Pérez, Ch., *Enabling Transparent Data Sharing in Component Models*, Technical report, Research Report RR-5796, INRIA, IRISA, Rennes, France, November 2006

[4] Arnautovic, E., Kaindl, H., Falb, J., *Gradual transition towards autonomic software systems based on high-level communication specification*, Proceedings of the 2007 ACM symposium on Applied computing, Seoul, Korea, ACM Press New York, NY, USA, 2007, Pages: 84 – 89

[5] Ashby, W. R., *Introduction to Cybernetics*, Chapman & Hall, London, 1956

[6] Aulin, A., *The Cybernetic Laws of Social Progress*, Pergamon, Oxford, 1982

[7] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., *Business Process Execution Language For Web Services Version 1.1 Specification*, ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[8] *Common Base Event Specification*, http://www-106.ibm.com/developerworks/webservices/library/ws-cbe/

[9] Beer, S., *Diagnosing the System of Organizations*, John Wiley & Sons, 1995

[10] Beer, S., *Cybernetics and Management*, English Universities P., 1967

[11] Bertalanffy, L., *General System Theory*, George Braziller, 1969

[12] Blackman, D., *Debian Package Management, Part 1: A User's Guide*, Linux Journal, Volume 2000 , Issue 80es (November 2000), Specialized Systems Consultants, Inc., 2000

[13] Bouchenak, S., De Palma, N., Hagimont, D., Krakowiak, S., Taton, C., *Autonomic Management of Internet Services: Experience with Self-Optimization (short paper)*, in: 3rd International Conference on Autonomic Computing (ICAC), Dublin, Ireland, June 2006

[14] Bouchenak, S., De Palma, N., Hagimont, D., Taton, C., *Autonomic Management of Clustered Applications*, in: IEEE International Conference on Cluster Computing, Barcelona, Spain, September 2006

[15] Brada, P., *Metadata support for safe component upgrades*, 26th Annual International Computer Software and Applications Conference, COMPSAC 2000, pp. 1017 – 1021, 2000

[16] Brooks, R.A., *Intelligence Without Reason*, in 'Proceedings, IJCAI-91', Sydney, Australia, 1991

[17] Brown, A., *A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors With System-Wide Undo*, UC Berkeley Computer Science Division Technical Report UCB//CSD-04-1304, December 2003

[18] Brown, A. and Patterson, D. A.*, Embracing Failure: A Case for Recovery-Oriented Computing (ROC)*, High Performance Transaction Processing Symposium, Asilomar, CA, October 2001

[19] *OSCAR Bundle Repository*, http://oscar-osgi.sourceforge.net/

[20] Bustard, D., Sterritt, R., Bendiab, A., Laws, A., Randles, R., Keenan, F., *Towards a Systemic Approach to Autonomic Systems Engineering*, Proceedings of 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05) , pp. 465-472, 2005

[21] Cheng, S., Garlan, D., Schmerl, B., Sousa, J., Spitznagel, B., Steenkiste, P., Hu, N., *Software Architecture-based Adaptation for Pervasive Systems*, International Conference on Architecture of Computing Systems Trends in Network and Pervasive Computing, Karlsruhe, Germany, Volume 2299, April 8-11, 2002

[22] Chen, G., Kong, Q.*, Integrated Management Solution Architecture*, Network Operations and Management Symposium, IEEE Press, 2000, ISBN 0-7803-5930-5

[23] Chung, S., An, J., Davalos, S., *Service-Oriented Software Reengineering*: SoSR, 40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007, E-ISBN: 0-7695-2755-8

[24] Claudel, B., De Palma, N., Lachaize, R., Hagimont, D., *Self-protection for Distributed Component-Based Applications*, in: 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (formerly Symposium on Self-stabilizing Systems) (SSS 2006), Dallas, TX, USA, November 2006

[25] Eclipse Community Page, http://www.eclipse.org

[26] Java Community Process, http://www.jcp.org

[27] Java Community Process JMX Remote API Specification (JSR 160), http://jcp.org/en/jsr/detail?id=160

[28] NetBeans Community Page, http://www.netbeans.org

[29] IBM Corporation, *An architectural blueprint for autonomic computing*, third edition, June 2005, White Paper

[30] IBM Corporation, *Log and trace analyzer for autonomic computing*, Online documentation available at: http://www.alphaworks.ibm.com/tech/logandtrace

[31] Dalakakis, S., Dieterich, M., Roller, D., Warschat, J., *Multiagentensystem zur Wissenskommunikation in der Produktentstehung-Rapid Product Develoment.* In: Wirtschaftsinformatik 2005, "eEconomy eGovernment eSociety". Eds.: O. K. Ferstel, E. J. Sinz, S. Eckert, T. Isselhorst, Physica-Verlag, Heidelberg, 2005, ISBN 3-7908-1578-8, pp. 1621-1640

[32] Dalakakis, S., Stoyanov, E., Roller, D., *A Retrieval Agent Architecture for Rapid Product Development*, In: Perspectives from Europe and Asia on Engineering Design and Manufacture, X.-T

[33] Danciu, V., König, R., Treu, G., Weiss, D., *Policy–based Update Management in Smart Home Environments (PUSH)*, Kooperationsbericht Siemens — MNM–Team, Siemens/TUM–Kooperation, Technical Report, Dezember, 2006

[34] Alexandre Denis, Christian Pérez, Thierry Priol and André Ribes, *A Component-Based Software Infrastructure for Grid Computing.* Technical report, Research Report RR-4974, INRIA, IRISA, Rennes, France, October 2003Alexandre Denis, Christian Pérez, Thierry Priol and André Ribes. Padico: A Component-Based Software Infrastructure for Grid Computing. Technical report, Research Report RR-4974, INRIA, IRISA, Rennes, France, October 2003

[35] *Universal Description, Discovery and Integration* (UDDI), http://www.uddi.org/

[36] Diao, Y., Hellerstein, S.L., Parekh, S., Bigus, J.P., *Managing Web Server Performance with AutoTune agents*, IBM Systems Journal, Vol. 42, No 1, 2003

[37] Dudzik, S., Einhorn, J., Schönleber, T., *Untersuchung des IBM Autonomic Computing Toolkits*, Fachstudie, Universität Stuttgart, Institut für Paralelle und Verteilte Systeme, 2004

[38] Dulay, N., Heeps, S., Lupu, E., Mathur, R., Sharma, O., Sloman, M., Sventek, J., AMUSe: *Autonomic Management of Ubiquitous e-Health Systems*, Proceedings of the UK e-Science All Hands Meeting 2005

[39] Java 2 Enterprise Edition Specification, http://java.sun.com/javaee/ reference/index.jsp

[40] Wong, W. E., Horgan, J. R., London, S., Agrawal, H., *A Study of Effective Regression Testing in Practice*, IEEE TENCON Digital Signal Processing Applications Proceedings, 1996

[41] Fidge, C., *Fundamentals of Distributed System Observation*, IEEE Software, Vol. 13, No 6, pp. 77-83, November 1996

[42] Fischer, M., Gall, H. C., *EvoGraph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems*, In proceedings of the 13th Working Conference on Reverse Engineering (WCRE), Pages: 179-188, IEEE, 2006

[43] Fleury, E., Frénot, S., *Building a JMX management interface inside OSGi*, INRIA Rhône-Alpes ARES, 2003

[44] Fleury, M., Reverbel, F., *The JBoss Extensible Server*, Middleware 2003 - ACM/IFIP/USENIX International Middleware Conference, 2003

[45] Foster, I. and Kesselman, C., The Grid: *Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999

[46] Ganek, A. G., Corbi, T.A., *The Dawning of The Autonomic Computing Era*, IBM Systems Journal, Vol. 42, NO 1, 2003

[47] Gall, H., Klösch, R., Mittermeir, R., *Object-Oriented Re-Architecturing*, in 5th European Software Engineering Conference, September 1995

[48] García, V.G., Sobrado, I. and Uhring, D., *On Auto-configurable Network Devices*, From Proceeding (462) Internet and Multimedia Systems and Applications, 2005

[49] Garlan, D., Robert, T., David Wile, D., *Acme: Architectural Description of Component-Based Systems.* In Gary T. Leavens and Murali Sitaraman editors, Foundations of Component-Based Systems, Pages 47-68, Cambridge University Press, 2000

[50] Gjørven, E., Rouvoy, R., and Eliassen, F., *Cross-layer self-adaptation of service-oriented architectures*, In Proceedings of the 3rd Workshop on Middleware For Service Oriented Computing (Leuven, Belgium, December 01 - 05, 2008), MW4SOC '08. ACM, New York, NY, 37-42, 2008

[51] *Next Generation Operations Support System* (NGOSS) Lifecycle, http://www.tmforum.org/browse.aspx?catID=1683

[52] Gray, J., *Why do computers stop and what can be done about it?,* Symposium on Reliability in Distributed Software and Database Systems, 1986

[53] Griswold, W.G., Shonle, M., Sullivan, K., *Modular software design with cross-cutting interfaces*, IEEE Software, Volume 23, Issue 1, Pages 51 – 60, 2006

[54] Hale, J. C., *Seamless and Secure Interoperation of Heterogeneous Distributed Objects*, Doctoral Thesis, University of Tulsa, 1997

[55] Herring, C. and Kaplan, S., *The Viable System Model for Software*, In 4th World Multiconference on Systemics, Cybernetics and Informatics, 2000

[56] Hinchey, M. G. and Sterritt, R., *Self-managing software*, IEEE Computer, 39(2):107– 109, 2006

[57] Hnetynka, P., *A Model-driven Environment for Component Deployment*, Proceedings of SERA 2005, Mount Pleasant, Michigan, USA, IEEE CS, ISBN 0-7695-2297-1, pp. 6-13, Aug 2005

[58] Horn, P., *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation, October 15, 2001

[59] Chen, H., Kim, B., Yang, J., Hariri, S., Parashar, M., *Autonomic Runtime System for Large Scale Parallel and Distributed Applications*, UPP Workshop (Unconventional Programming Paradigms), Sept 15-17, 2004

[60] Chen, H., Hariri, S., Kim, B., Zhang, M., Zhang, Y., Khargharia, B., Parashar, M., *Self-Deployment and Self-Configuration of Network Centric Service*, IEEE International Conference on Pervasive Computing (IEEE ICPC), 2004

[61] The Internet Society, Network Working Group, *Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)*, 2002

[62] The Internet Society, *SNMPv3 Management Information Base Specification*, http://tools.ietf.org/html/rfc3418, 2003

[63] Iqbal, R., James, A., Gatward, R., *A framework for interoperability of heterogeneous systems*, Proeceedings from 14th International Workshop on Database and Expert Systems Applications, 1-5 Sept. 2003, pp.768 – 772, 2003

[64] Hutcheson, M. L., *Software Testing Fundamentals: Methods and Metrics*, Wiley, 1st edition, April 11, 2003

[65] Corba IIOP Specification, http://www.omg.org/technology/documents/ formal/corba_iiop.htm

[66] Kalibera, T., Tuma, P., *Distributed Component System Based On Architecture Description: The SOFA Experience*, Proceedings of DOA 2002, Irvine, CA, USA, Copyright (C) Springer-Verlag, pp. 981-994, LNCS2519, ISBN 3-540-00106-9, ISSN 0302-9743, Oct 2002

[67] Keromytis, A. D., Parekh, J., Gross, Ph. N., Kaiser, G., Misra, V., Nieh, J., Rubenstein, D., Stolfo, S., *A holistic approach to service survivability*, Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems: in association with 10th ACM Conference on Computer and Communications Security, Pages: 11 - 22, 2003

[68] Kikuchi, S., Tsuchiya, S., Adachi, M., Katsuyama, T., "*Policy Verification and Validation Framework Based on Model Checking Approach*", Autonomic Computing, International Conference on, vol. 0, no. 0, pp. 1, Fourth International Conference on Autonomic Computing (ICAC'07), 2007

[69] Kozaczynski, W., Booch, G.Software, *Component-Based Software Engineering*, IEEE Volume 15, Issue 5, Sep/Oct 1998 Page(s):34 – 36

[70] Kuo, Benjamin C., *Automatic Control Systems (6th ed.)*, New Jersey: Prentice Hall. ISBN 0-13-051046-7, 1991

[71] Kwiat, K., Ren, Sh., *A Coordination Model for Improving Software System Attack-tolerance and Survivability in Open Hostile Environments*, sutc, pp. 394-402, IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06), 2006

[72] Kyaruzi, J., van Katwijk, *Concerns on Architecture-Centered Software Development*, A Survey, Transactions of the SDPS, Sep 2000, Vol. 4., No 3, pp. 13-35

[73] Lahmadi, A., Andrey, L., Festor, O., *On the Impact of Management on the Performance of a Managed System: A JMX-Based Management Case Study*, In: 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management - Management of Ambient Networks - DSOM 2005, Barcelona, Spain, Springer-Verlag, Oct 2005, vol. 3775, p. 24–35

[74] Lee, J., Kim, J., Gyu-Sang Shin, *Facilitating Reuse of Software Components using Repository Technology*, apsec, p. 136, 10th Asia-Pacific Software Engineering Conference (APSEC'03), 2003

[75] Lehman, M.M., *Programs, Life Cycles, and Laws of Software Evolution*, Proceedings of the IEEE 68(9), pp. 1060—1076, September 1980

[76] Lehman, M.M., *On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution*, with DE Perry and JF Ramil, Proc. Metrics'98, Bethesda, Maryland, 20-21 Nov. 1998

[77] Lehman, M.M., *The Future of Software - Managing Evolution*, inv. contr. to sp. ed. of IEEE Software, pp. 40-44, Jan. 1998

[78] Lehman, M.M., *Models in Software Development and Evolution*, Int. Conf. on the Software Process Modelling in Practice, London, 22-23 Apr. 1993

[79] Liu, D., Peng, J., Law, K.H., Wiederhold, G., Sriram, R.D., *Composition of engineering web services with distributed data-flows and computations*, Advanced Engineering Informatics 19(1): 25-42 (2005)

[80] Mahmooda, S., Laia, R., *A survey of component based system quality assurance and assessment*, Information and Software Technology, Volume 47, Issue 10, pp. 693-707, 2005

[81] Distributed Management Task Force (DMTF), *Common Management Model (CIM) Standards*, http://www.dmtf.org/standards/cim

[82] Sun Microsystems, *Java Management Extensions (JMX)*, http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[83] Object Management Group, *CORBA Naming Service*, http://www.omg.org/technology/documents/formal/naming_service.htm

[84] Meijer, E., Gough, J., *Technical Overview of the Common Language Runtime*, Microsoft Research, 2000

[85] Melcher, B., Mitchell, B., *Towards an Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications*, Intel Technology Journal, Volume 8, Section 4, November 17, 2004

[86] Meyers, R.A., *Cybernetics and Second-Order Cybernetics* , Encyclopedia of Physical Science & Technology, Academic Press, New York USA, 2001

[87] Mezini, M., Seiter, L., Lieberherr, K., *Component integration with pluggable composite adapters In Software Architectures and Component Technology*, Kluwer, 2000

[88] Moore, B., Ellesson, E., Strassner, J., Westerinen, A., *RFC 3060 - Policy Core Information Model*, Version 1, Specification

[89] Sun Microsystems, *Java Naming and Directory Services* - http://java.sun.com/products/jndi/

[90] Nett, E., Mock, M., Theisohn, P., *Managing dependencies-a key problem in fault-tolerant distributedalgorithms*, Fault-Tolerant Computing, FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium 24-27 Jun 1997, pp. :2 – 10, 1997

[91] Nosek, J. T., Roth, I., *A comparison of formal knowledge representation schemes as communication tools; predicate logic vs semantic network*, International Journal of Man-Machine Studies, Volume 33 , Issue 2 (August 1990), Pages: 227 - 239, 1990

[92] Opdyke, W. F., *Refactoring Object-Oriented Frameworks,* PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992

[93] Opletal, S., Dalakakis, S., Roller, D., *Towards Semantic-Based CAD User Interface and Core Components*, In: "Applications of Digital Techniques in Industrial Design Engineering", Proceedings of the 6th International Conference on Computer-Aided Industrial Desingn & Conceptual Desingn. Eds: Pan, Y., Vergeest, J., Lin, Z., Wang, Ch., Sun, S., Hu, Z., Tang, Y., Zhou, L., International Academic Publishers, World Publishing Corporation, Beijing, ISBN 7-5062-7444-2, pp. 497-502, 2005

[94] Oreizy, P., Gorlick, M.M., Taylor, R. N., *An Architecture-Based Approach to Self-Adaptive Software*, IEEE Intelligent Systems, Volume 14 , Issue 3 (May 1999), Pages: 54 - 62 , 1999

[95] Parashar, M., *AutoMate: Enabling Autonomic Applications*, IBM Visit, Rutgers University, NJ, USA, November 2003

[96] Patterson, D. A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N., *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, U.C. Berkeley Computer Science Technical Report, UCB//CSD-02-1175, University of California, Berkeley, March 15, 2002

[97] Pinzger, M., Fischer, M. and Gall, H.C., *Towards an Integrated View on Architecture and its Evolution*, Electronic Notes in Theoretical Computer Science, Pages: 183-196, 2005

[98] Universal Plug&Play (UPnP) Community, http://www.upnp.org

[99] Ranganathan, A. and Campbell, R. H., *Pervasive Autonomic Computing Based on Planning*, IEEE International Conference on Autonomic Computing (ICAC 2004), New York, NY, US,. May 17-18, 2004

[100] FIPA, *Recruiting Interaction Protocol Specification,* Foundation for Intelligent Physical Agents, http://www.fipa.org

[101] Riel, A. J., *Object Oriented Design Heuristics*, Addison-Wesley Pub Co, 1st edition, April 30, 1996

[102] Ritsko, J. J., Birman, A., *Evolution of Grid Computing Architecture and Grid Adoption Models*, IBM Systems Journal Vol. 43, No 4, 2004

[103] Sahoo, R. K., Rish, I., Oliner, A. J., *Autonomic Computing Features for Large-scale Server Management and Control.*, et. al. IJCAI-03 workshop on AI and Autonomic Computing, August 2003

[104] Sun Microsystems, *Java RMI Specification,* http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html

[105] Roller, D., Mešina, M., Lampasona, C., *Concurrency Control and Locking in Knowledge Base for Rapid Product Development*, In: Luo, Y. (Ed.), Cooperative Design, Visualization, and Engineering.Lecture Notes in Computerscience, Springer-Verlag, Berlin Heidelberg New York, ISBN 3-540 28948-8, pp. 79-85, 2005

[106] Rothermel, G. and Harrold, M., *Analyzing regression test selection techniques*, IEEE Transactions on Software Engineering, 22(8):529–551, August 1996

[107] Rothermel, G. and Harrold, M.J., *A safe efficient regression test selection technique*, ACM Transactions on Software Engineering and Methodology, 6(2):173–210, April 1997

[108] Rothermel, K., Dürr, Fr., *Location-based Services: Auf dem Weg zu kontextbezogenen Informations- und Kommunikationssystemen*, In: ITG (ed.): Jubiläumsfachtagung 50 Jahre ITG: Zukunft durch Informationstechnik – Schnell-Mobil-Intelligent., Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme)., pp121-128, deutsch, VDE-Verlag, ISBN: 3-8007-2825-7, 2004

[109] Open Services Gateway Initiative (OSGi) Aliance, http://osgi.org

[110] Oh, S., J., Sang Ho Lee, *Resource Selection for Autonomic Database Tuning,* ICDEW archive, Proceedings of the 21st International Conference on Data Engineering Workshops, IEEE Computer Society, Washington, DC, USA, pp 1218, 2005

[111] Schmidt, D., Stal, M., Rohnert, H., Buschmann, Fr., *Pattern Oriented Software Architectures, Volume 2 : Patterns for Concurrent and Networked Objects*, John, Wiley & Sons, 2000

[112] Shannon, C. E., *The Mathematical Theory of Communication*, University of Illinois Press, Chicago, 1963

[113] Sicard, S., Boyer, F., and De Palma, *Using components for architecture-based management: the self-repair case*, In Proceedings of the 30th international Conference on Software Engineering (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 101-110, 2008

[114] Namjoshi, K. Sh., *Ameliorating the State Space Explosion Problem*, PhD thesis, UT Austin, 1998

[115] Sobr, L., Tuma, P., SOFAnet: *Middleware for Software Distribution over Internet*, In proceedings of Symposium on Applications and the Internet (SAINT 2005), Trento, Italy, Copyright (C) IEEE, Piscataway, New Jersey, USA, ISBN 0-7695-2262-9, pp. 48-53, Feb 2005

[116] Sridharan, B., Mathur, A. P., *Infrastructure for the Management of Smart Homes*, Whitepapaper, Spring SERC Showcase 2001

[117] Stevens, G., Quaisser, G., Klann, M., *Breaking it up: an Industrial Case Study of Component-Based Tailorable Software Design*, Springer, 2005

[118] Stoyanov, E., Roller, D., *Robust Software Architecture Design of Distributed Product Development System*, in: D.; Roller, S. Opletal. (Hrsg.): ELEKTROTECHNIK CAD – Intelligente Genetische Algorithmen Aktuelle Entwicklungen, Aachen: Shaker, ISBN 3-8322-5568-0, pp. 21-32, 2006

[119] Stoyanov, E., Roller, D., Wischy, M., *Using Managed Communication Channels in Software Components*, 20th International Conference on Advanced Information Networking and Applications, Volume 2 (AINA'06), IEEE Computer Society, Vienna, Austria, ISBN: 0-7965-2466-4, pp. 499-503, 2006

[120] Stoyanov, E., MacWilliams, A., Wischy, M., Roller, D., *Distributed Software Maintenance Using an Autonomic System Management Approach based on the Viable System Model*, in P.Dini, D. Ayed (Eds), Proceedings of icas, International Confernce on Autonomic and Autonomous Systems (ICAS'06), IEEE Computer Society, San Jose, ISBN: 0-7695-2653-5, pp. 58-64 17, 2006,

[121] Stoyanov, E., Wischy, M., Roller, D., *Using Managed Communication Channels in Software Components*, Proceedings of ACM International Conference on Computing Frontiers, Ischia, Italy, 2006, session: Reconference on autonomic computing, ACM Press, New York NY, USA, ISBN: 1-59593-302-2, pp. 177-186, 33 NS

[122] Stoyanov, E., Wischy, M., Roller, D., *Cybernetics and General Systems Theory (GST) Principles for Autonomic Computing Design*, S. Kawada (Ed), Proceedings of icac, Second International Conference on Autonomic Computing (ICA'05), IEEE Computer Society, Los Alamitos USA, ISBN 0-7695-2276-9, pp. 389-390, 2005

[123] Stoyanov, E., Dalakakis, S., Roller, D., Wischy, M., *Supporting the Rapid Product Development Requirements by a viable Software Architecture*. In: Engineering Design and the Global Economy. Eds.: Samuel, A., Lewis, W., Institution of Engineers Australia, Barton, ISBN 0-85825-788-2, pp. 636-637, 2005

[124] Sventek, J., Badr, N., Dulay, N., Heeps, S., Lupu, E., Sloman, M., *Self-Managed Cells and their Federation*, Proceedings of the 17th Conference on Advanced Information Systems Engineering, 2005

[125] Takemiya, H., Shudo, K., Tanaka,Y., *Constructing Grid Applications Using Standard Grid Middleware*, Journal of Grid Computing, Kluwer Academic Publishers 117-131, 2004

[126] W3C Technical Reports and Publications, http://www.w3.org/TR/

[127] Thiel, S., Dalakakis, S., Roller, D., *A Learning Agent for Knowledge Extraction from an Active Semantic Network*. IEC (Prague) 2005: 217-220, 2005

[128] Trumler, W., Bagci, F., Petzold, J., Ungerer, T., *Smart Doorplates - toward an autonomic computing*, Autonomic Computing Workshop, 2003, Page: 42 – 47, 2003

[129] Trumler, W., Klaus, R. and Ungerer, T., *Self-configuration via Cooperative Social Behavior*, Third International Conference, ATC 2006, Wuhan, China, 2006

[130] Trumler, W., Bagci, F., Petzold, J., Theo Ungerer, *AMUN - autonomic middleware for ubiquitous environments applied to the smart doorplate*, ELSEVIER Advanced Engineering Informatics, Volume 19 Issue 3, Pages 243-252, 2005

[131] Valetto, G., Kaiser, G., Phung, D., *A Uniform Programming Abstraction for Effecting Autonomic Adaptations onto Software Systems*, Proceedings from Second International Conference on Autonomic Computing (ICAC'05), IEEE Press, 2005, pp. 286-297

[132] van Moorsel, A.P.A. and Wolter, K., *Analysis of Restart Mechanisms in Software Systems*, IEEE Transactions on Software Engineering, Vol. 32, Issue 8, pp. 547-558

[133] van Moorsel, A.P.A., *Grid, Management and Self-Management*, The Computer Journal, Vol. 48, Issue 3, pp. 325-332

[134] Want, R., Pering, T. and Tennenhouse, D., *Comparing Autonomic and Proactive Computing*, IBM Systems Journal 42, No. 1, 129-135 2003

[135] Whiteson, S., Stone, P., *Towards autonomic computing: adaptive network routing and scheduling*, Proceedings from International Conference on Autonomic Computing, 2004, 17-18 May 2004, pp 286 – 287

[136] Witten, I. H., Frank, E., *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*, Morgan Kaufmann; 2 edition (June 10, 2005), ISBN-13: 978-0120884070

[137] Yan, Ch., Jiang, Y., Juster, N. P. (eds.), *Perspectives from Europe and Asia on Engineering Design and Manufacture: A Comparison of Engineering Design and Manufacture in Europe and Asia*, Kluwer Academic Publishers, Dordrecht Boston London, ISBN 1-40202211-5, pp. 41-58, 2004

[138] Yergeau, F., Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation, 4th February 2004

[139] Zao, F., *Intelligent Computing, About Complex Systems,* Mathematics and Computers Simulation, 36: 423-432, 1994

[140] "N1: Revolutionary IT Architecture for Business" www.sun.com/software/solutions/n1/overview.html, visited Feb. 13, pp. 1-3, 2003

Curriculum Vitae

Emil Stoyanov was born in the town of Ruse, Bulgaria. In the summer of 1997, three months after graduating the highschool he was enrolled into the course "Computer Systems and Technologies" at the University of Ruse. During his studies for the degree of “Master of Science” he participated in a series of international academic projects related to distributed control, measurement and simulation. In December of year 2002 Emil Stoyanov was assigned a helping function with the research team of Institut für Rechnergestützte Ingenieursysteme (IRIS) at Universität Stuttgart where he assisted in the development of a distributed knowledge and communication platform. An year later he started his PhD studies in the frame of a cooperation between IRIS and Siemens Corporate Technology, Munich. Main topic of his research is development of distributed software components and system management. As of 2008 Emil Stoyanov is leading a team of software engineers at a Bulgarian provider of research and development services in the field of telecommunications and mobile applications.