2006

A framework for self-configuration

Fadil Mesic
San Jose State University

A FRAMEWORK FOR SELF-CONFIGURATION

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by Fadil Mesic

August 2006

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

G Pour

Dr. Gilda Pour, Committee Chair

Al Alaverdi

Al Alaverdi, SigmaQuest, Inc., Committee Member

Dr. Donald Hung, Committee Member

APPROVED FOR THE UNIVERSITY

Rhea L. Williamson 07/17/06

ABSTRACT

A FRAMEWORK FOR SELF-CONFIGURATION

By Fadil Mesic

This thesis explores autonomic computing, and its role in coping with increasing complexity and costs of IT infrastructure. Fast-paced development, population increase, and demand on resources and information, as well as globalization processes and interconnectivity of infrastructure networks, pose new challenges in maintenance, security, optimization, problem detection, and repair.

The first part of this thesis presents the current state in the area of autonomic computing. Aspects of self-configuration, self-optimization, self-healing, and self-protection are explored. Problems and challenges in the building autonomic systems are examined as well. The second part of the thesis proposes a framework for self-configuration based on stable, pattern-based architecture. Web services and software agent technologies are utilized in the design of the proposed framework. Design of a sample application is described to demonstrate use and applicability of the framework. The thesis concludes with proposals for future work in this area and possible enhancements of the proposed framework.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and appreciation to my thesis advisor, Dr. Gilda Pour, for the valuable assistance, support, constructive criticism, encouragement, and patience she provided during my graduate coursework and preparation of my thesis.

I would also like to thank my committee members, Al Alaverdi and Dr. Donald Hung, for their valuable insight and feedback.

I would like to express my deepest gratitude and appreciation to my wife, Binnur Kurtlar-Mesic, who provided tremendous support and motivation for completion of my study and this thesis.

I would also like to express my thanks to my undergraduate professors from my alma mater at University of Sarajevo, Bosnia & Herzegovina, for building strong foundations in my education.

At the end, I would like to thank my family for their unrelenting encouragement and unconditional support throughout my education.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER: 1 Introduction

1.1 Subject

The subject of this thesis is self-configuring systems and a self-configuration framework. The thesis introduces the methodology for evaluation of autonomic systems and examines their role in dealing with the complexity and costs of infrastructure. Problems of reliability, maintainability, and supportability are also concerns of this work.

1.2 Statement of the Need

The constant growth of the population, increased demand for resources, energy, new products, information, and the movement of people and materials drive expansion of the infrastructure necessary to respond to these demands. The infrastructure is not only getting larger, but more interconnected and complex, with numerous dependencies between nodes.The national power grid, transportation networks, and telecommunication networks are all more difficult to control and manage as new nodes and connections are added to the systems.

In no other infrastructure are these changes more profoundly seen as in the "information highway." The Internet is growing and getting more complex at an astonishing rate. Figure 1 presents the historical growth of the Internet based on host number.

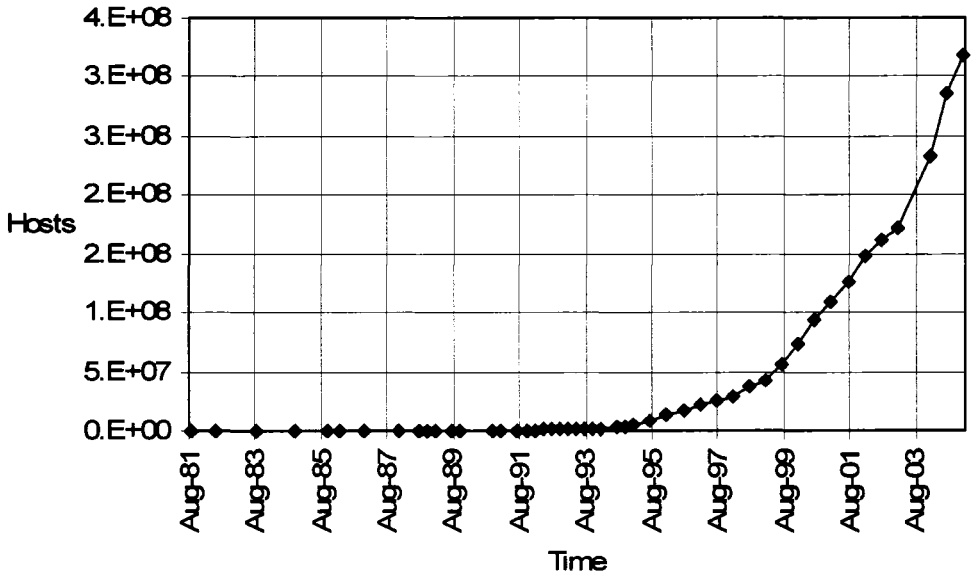

Figure 1. Internet growth.

From Internet Systems Consortium, Inc.

Other indicators of growth, such as the number of users and the number of registered domain names, follow similar patterns.

Another significant problem is the constant increase in software complexity. Operating systems, system software, and applications are getting bigger and bigger. The number of programs running on a single system is constantly rising. Figure 2 shows the increase in the size of Microsoft Windows OS from its Windows 3.1 release in 1992, to Windows XP release in 2001. The Line of Code (LOC) is a very simple metric, but it clearly shows that software is getting bigger and bigger, and bigger software often means more complex software.

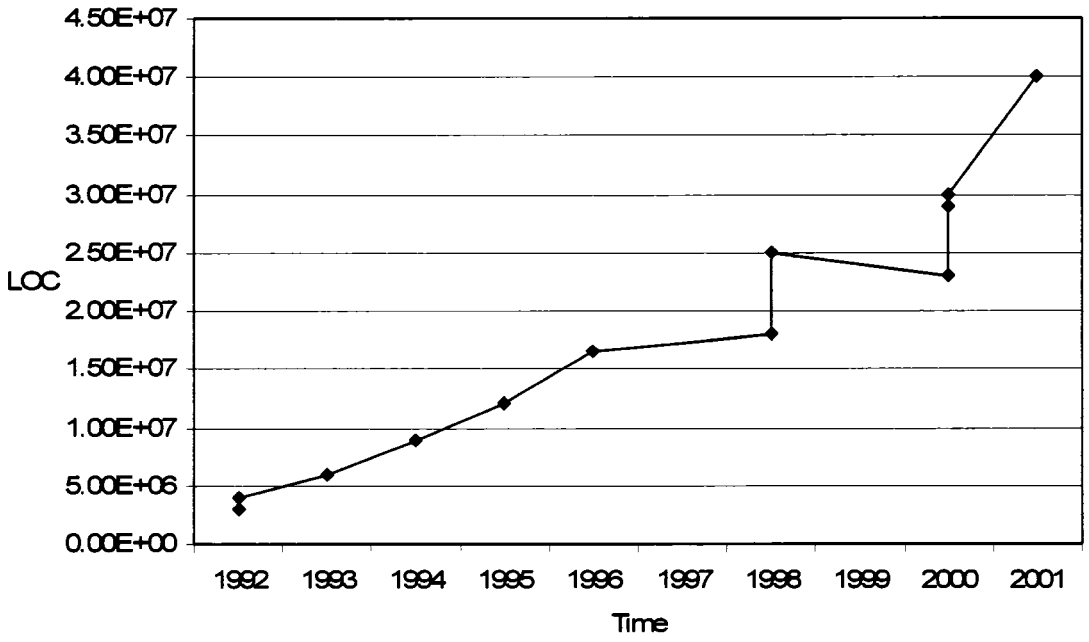

Figure 2. Microsoft Windows OS size increase.

Source of data from *Modern Operating Systems* (p.763), by A. S. Tanenbaum, 2001, Prentice Hall and "Software Complexity and Security," by B. Schneier, March 2000, *Crypto-Gram Newsleeter*. Retrieved May 5, 2006, from http://www.counterpane.com/crypto-gram-0003.html

Space exploration, undersea research, and equipment maintenance in remote and inhospitable environments put special demands on the systems that operate in those conditions. A degree of autonomy and adaptability is required to make appropriate decisions and respond to the unanticipated changes in the environment. Such systems require heuristics and reasoning to respond to the emergent situations.

Another acute problem is the increase in operational costs of the infrastructure and all systems that support it. It becomes increasingly difficult, yet cost-effective, to maintain and support a complex IT infrastructure, comprised of numerous hosts running different operating systems and applications.

Some of the systems of vital importance require uninterruptible and very reliable operation. Responses to these requirements are costly, redundant, and hot-swappable systems. Even those systems still require regular maintenance and attention.

We already depend in many ways on a variety of systems which routinely operate in a ubiquitous way, and we only know that they are there when a catastrophic failure occurs. We use electric power everyday and take its availability for granted. Yet, when a big outage in 2003 occurred, most of the Northeast of the United States felt its consequences deeply. Rapid responses are required to address failures in the power grid, telecommunication networks, and transportation. Prevention of "ripple" effects when a failure occurs is one of the most important and often very difficult tasks to achieve (Amin, 2000). Humans lack speed required to act in those situations, and an automatic response is necessary. Yet appropriate knowledge has to be built into the system to make the right decision. In the 2003 outage, a wrong response was the cause of the situation. The electric grid's control system acted as designed, yet overall the result was wrong due to the system's inability to reason and foresees consequences of the action taken.

It is safe to argue that we have already reached a point where a smarter and more efficient way is necessary to address problems like this. Autonomic computing has the objective of freeing people from the need to manage, support, and maintain computing systems by having computing systems take care of themselves.

1.3 Objective

The goal of this thesis is to give a profile of the current state in the area of autonomic computing and explore methods for evaluation, analysis, and design of

autonomic computing systems. This thesis also addresses the main challenges and problems in building autonomic computing systems and possible ways for their resolution.

This work examines a set of methods for evaluation, analysis, and design of autonomic computing systems and evaluates different approaches in how to deal with some of the challenges in designing and building autonomic computing systems. Results of this research are a proposal for a systematic approach in analysis and design of autonomic computing systems, a guideline for selecting appropriate architecture, and strategies to cope with problems and challenges. The goal is to create a basic framework which is generic enough to be applied in various autonomic computing problems. The stable core of the framework tries to capture fundamental domain knowledge, and additional work is required to adapt framework to a particular problem.

1.4 Scope

The scope of this work includes evaluation, applicability, design methodology, and challenges in building autonomic systems with specific focus on self-configuring aspects of autonomic computing. The scope also includes a research of the current state in the domain of autonomic computing and a comparison of architectures proposed by different authors. An architecture for self-configuring systems is presented.

1.5 Research Methodology

This thesis is based on work of other authors and reflects on their achievements and proposed solutions. Research is conducted through (a) initial research; (b) a review of the current status and the relevant work in the area of autonomic systems; (c) a comparison

of different methods and solutions proposed so far; (d) an analysis of the problem; (e) a proposal of a systematic approach for analysis, design, and selection of architecture; (f) a proposal of the strategies and guidelines to deal with the challenges; and (g) experiments and examples to support proposed methodology. Examples will focus primarily on self-configuring aspects of autonomic computing.

1.6 Thesis Stratification

Chapter 1 of this thesis gives an introduction to the problem and explains scope and goals of this work.

Chapter 2 presents a review of the relevant work and literature in the area of autonomic computing and particularly to the self-configuring aspect of autonomic computing.

Chapter 3 explores different aspects of autonomic computing, their role, and interaction.

Chapter 4 explores different methodologies and architectures suitable for building autonomic computing systems, their advantages and disadvantages and applicability of architecture to particular feature of an autonomic system.

Chapter 5 examines various challenges and problems in designing and building autonomic computing systems and possible strategies and directions taken to solve those problems.

Chapter 6 focuses on domain of self-configuration.

Chapter 7 presents a framework for self-configuration. Detail analysis and design is given, with rationale for employing specific methodology or technique. This chapter

focuses on self-configuring aspect of autonomic computing. A comprehensive domain analysis is given, with identification of the self-configuring elements. Top-level entities (self-configuring modules) are identified, and each module is further analyzed and designed, with elements of reusability in mind. Analysis patterns were created and combined in a stable architecture which can be varied for use in different self-configuring problems. Software agents are mapped to elements of the framework and a software agents-based implementation is described.

Chapter 8 utilizes the proposed framework for designing a sample application.

Chapter 9 gives a conclusion and a perspective for future work. A summary of thesis contributions and a conclusion is given. Future directions and work are presented as well.

Cited literature and references are presented in the References section. Each chapter starts with an introduction and ends with final remarks.

1.7 Chapter Last Remarks

In this chapter a brief introduction of the research area problem and the scope of the research were introduced. Objectives and research methodology were discussed as well. The next chapter will explore the current state of research and relevant work in the area of autonomic computing.

CHAPTER: 2 Literature Review

2.1 Introduction

This chapter focuses on the current state of autonomic computing research and advances as reported in the scientific literature. A selection of literature and related work on the topic of autonomic computing is reviewed.

2.2 Literature Review

A number of authors have explored the problem of autonomic computing since its first appearance in IBM manifesto (2001). One group of authors approach problems of autonomic computing systems (self-optimization, self-protection, self-healing, and self-configuring) on a more formal and more generic level, while other authors focus more on a particular problem inside the domain of autonomic computing. These problems include Knowledge Management, Autonomic Database Systems, GRID and Peer to Peer (P2P) computing, High Reliability and Fault Tolerance, Adaptive Defense and Security, Adaptive Storage, Dynamic Routing, Service Discovery, Resource Allocation, Autonomic Economics, e-commerce, and more.

Bantz et al. (2003) focus on the problem of autonomic personal computing, the current state in that area, and propose an architecture for autonomic personal computing. McCann (2003) explores autonomic computing for database management systems proposing a DBMS architecture based on fine-grained, self-managed components instead of the usual, monolithic architecture. McCann introduces an adaptation framework and presents experimental results and experiences with "Patia" web-data architecture.

Kephart and Chess (2003) describe the structure of an autonomic element, a building block in an autonomic architecture. They also discuss a number of engineering and scientific challenges in building autonomic computing systems. Blair et al. (2002) focus on self-healing aspects of autonomic computing, elaborating particular suitability of reflective technologies for building autonomic computing systems with self-healing abilities. In their work, Open ORB reflective middleware technology is described in the detail and used as an example of reflective technologies with applicability in self-healing autonomic computing systems.

Another group of authors explore problems which have clear implications in the domain of autonomic computing but are not directly aimed at building an autonomic computing system. Georgiadis, Magee, and Kramer (2002) describe a self-organizing software architecture for distributed systems, with elements of self-configuration and reconfiguration in a dynamic environment. They present experiments with "Darwin Component Model" and a number of software tools used in the study. Jennings (2000) explores the agent-based software engineering, presenting an example of a flexible manufacturing system built upon software agents. He expresses "the need for flexible management of changing organizational structures" (p. 279). Although his work deals with agent technology, it clearly shows a similarity of problems in the areas other then computing, with the problems of self-management investigated in the area of autonomic computing. It is a good example of why agent technology is particularly suitable for addressing these problems. Whisnant, Kalbarczyk, and Iyer (2003) deal with dynamically reconfigurable software. They propose a formal system model for

dynamically reconfigurable software and then describe a concrete reconfigurable software-implemented fault-tolerant (SIFT) environment, based on premises given in the formal system. Amin (2000) gives an overview of self-healing autonomic systems in infrastructure networks and presents an example of the software agents based on the solution implemented in the national power grid. He also points to a number of problems that face designers of such large scale systems. He stresses the importance of a distributed solution and localization of failures, either through isolation or through repair. Cofino et al. (2003) focus on knowledge management in autonomic systems. They propose elements of Self-Learning and an adaptable Knowledge Base in conjunction with typical elements of Monitoring, Detection, Diagnostic, and Problem Resolution.

Adding autonomic features to existing software is explored by various authors. Sadjadi and McKinley (2005) propose a general programming model called transparent shaping to facilitate dynamic adaptation of existing software. Their approach combines four key software development techniques: aspect-oriented programming enabling separation of concerns at development time, behavioral reflection to support software reconfiguration at run time, component-based design to facilitate independent development and deployment of adaptive code, and adaptive middleware to encapsulate the adaptive functionality.

A number of authors focus on Web Services technology and its adaptation and use in achieving autonomic behavior. Sadjadi and McKinley (2005) focus on the blending of transparent shaping mechanism and web services to achieve self-managing of composite systems.

2.3 Chapter Last Remarks

In this chapter a selection of scientific literature on the topic of autonomic computing is presented. Autonomic computing is a hot, contemporary topic being actively researched by a number of scientists and industries and will continue to generate much interest. The scientific and technical communities are pursuing the goal of autonomic computing on multiple tracks, using different methodologies and techniques. New methodologies and technologies are being constantly proposed and developed. Since its inception in 2001, a number of available articles and work created on the topic of autonomic computing show significant continuous interest in this field.

CHAPTER: 3 Aspects of Autonomic Computing

3.1 Introduction

The term and basic principles of autonomic computing have been coined in the IBM Manifesto (2001). The paper introduces "the next grand challenge," designing and building self-managing computing systems. Paradoxically, to reduce the complexity in managing computing systems, systems must become more complex. The paper draws an analogy between autonomic computing and the autonomic behavior of the human body and central nervous system. The operations of both should be oblivious and effortless, acting on a sub-conscious level. Complexities are built into the system, but the user does not need to know about them since the system is able to take care of itself.

3.2 Self-Managing Systems

Kephart and Chess (2003) present four aspects of autonomic behavior: self-healing, self-configuration, self-optimization, and self-protection. These four aspects of autonomic computing system are defined as:

1. Self-Configuration – system is able to adapt to the changes through automatic configuration or reconfiguration of its parts or components.
2. Self-Healing – system is capable of automatic discovery and recovery from errors.
3. Self-Optimization – system is able to continuously improve its performance through constant monitoring, and resource management under a given set of constraints.
4. Self-Protection – system is able to proactively identify and protect itself from intrusions and attacks.

In their work, Kephart and Chess describe typical autonomic elements (Figure 3), consisting of Monitor, Analyze, Plan, and Execute (MAPE) modules, and an Autonomic Manager.

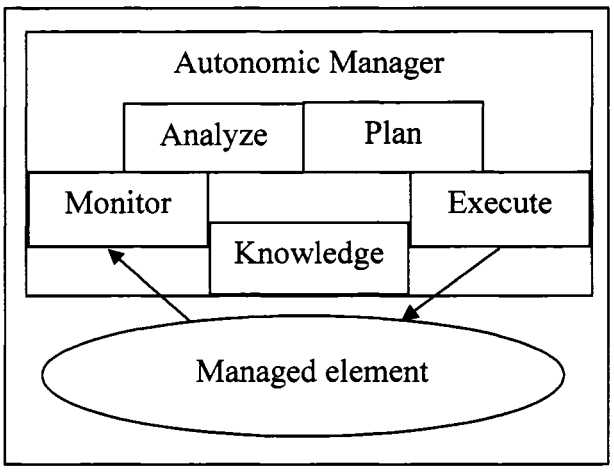

Figure 3. An autonomic element.

From "The Vision of Autonomic Computing," by J. O Kephart and D.M.Chess 2003, *IEEE Computer, 36*(1), 41-50.

The managed element is constantly being monitored, collected data is being analyzed, and a plan for action is being prepared and then executed. System environment is monitored for a number of reasons (performance, system health, system optimization, and security). In addition, other terms such as self-awareness, self-learning, self-monitoring, self-repair, etc. are in use by different authors and all relate to autonomic computing systems behavior. Sterritt and Bustard (2003) classify general properties of autonomic systems in goals, objectives, attributes, and approaches, where each class consists of:

1. Goal – self management
2. Objectives – self-configuring, self-healing, self-optimizing, self-protecting
3. Attributes – self-aware, environment aware, self-monitoring, self-adjusting
4. Approaches – make (systems engineering), achieve (adaptive learning)

In this classification, objectives reflect broad system requirements, while the attributes represent basic implementation mechanisms.

IBM introduces five evolutionary steps in achieving the ultimate goal of autonomic computing initiative, with level one representing the current situation (manual management of the systems), levels two to four having increased self-management capabilities, and level five being the realization of the ultimate goal of self-managing systems.

3.3 Characteristics of Autonomic Systems

IBM suggests eight basic characteristics that an autonomic computing system should possess:

1. An autonomic computing system needs to "know itself." This includes detailed knowledge about its components, resources, status, capacity, and relation to other systems.
2. An autonomic system must be able to configure and reconfigure itself under varying conditions.
3. An autonomic computing system must be able to perform constant improvement and self-optimization.

4. An autonomic computing system must be able to recover from failures and continue to operate. It must exhibit elements of self-healing.
5. An autonomic computing system must interact with the environment and protect itself and its resources from malicious or inadvertent threats and attacks. It must preserve security and integrity.
6. An autonomic computing system must adapt to the environment, interacting with other systems and even sharing/leasing its own resources with other systems.
7. An autonomic computing system must be open, based on open standards.
8. An autonomic computing system must optimize the resources management, distribution, and allocation to respond to user needs, all while keeping complexity hidden from the user.

The first characteristic is basically self-awareness of the system--its place and role in the environment in which it operates. Knowing its own structure, state, and relation to outer world allows a system to make proper operational decisions without involvement of humans. The second, third, fourth, and fifth characteristics relate to four aspects of autonomic computing, explored in the detail later in the text.

The sixth characteristic relates to the ability of a system to change its behavior dynamically during the run time, based on the environmental conditions. Adaptation is an important characteristic required by autonomic computing systems. The ability of a computing system to adapt to changes in the environment can be incorporated in the design of the system (adaptation by design) or can be added after by modifying the system behavior by building wrappers and adaptive layers around the system. The first

approach places additional requirements and constraints on the system leading to a flexible, adaptive design, whereas the second one is suitable for application on already deployed, legacy applications. Openness is an important characteristic, allowing an autonomic computing system to progress faster, being based, and developed on open standards and platforms.

Hiding complexity of the inner workings and optimizing its response based on user needs and requirements makes the autonomic computing system ubiquitous and available, resembling technologies such as electric power or telephone.

3.4 Self-Configuration

Companies are creating new products and product releases with an ever increasing speed. As a consequence, customer service, network operation, and support centers as well as system and network administrators are being overwhelmed and having a hard time in coping with delivery, support, and maintenance.

Lack of adequate design reviews and testing, a consequence of the fast-paced cycle of competition, demand, and desire to provide new products and services all result in frequent updates and numerous fixes. Propagation of software updates, patches, fixes and new functionality were historically done manually. Some of the major companies still prefer to do it this "traditional" way. For example, Sun Microsystems releases security updates as individual patches and "patch clusters" (equivalent of service packs in the Windows world) which have to be downloaded and installed manually. Installing a major database system, and its upgrades are still tasks which are done manually. A

number of prerequisites, which must be satisfied, present a nontrivial task even for a skilled system administrator.

Recently, software vendors have started to realize that the process of delivering upgrades and fixes must be improved. Today, there are a number of software vendors that include an automatic updater that usually runs as a service or a daemon, checking for updates on a predefined schedule, downloading and installing updates in the background. They give different levels of flexibility and control what and when will be downloaded and installed, when and how often to check for updates, and whether to inform the user about the whole process or not. Examples of software supporting automatic updates include:

1. Microsoft's Windows Automatic Update
2. Linux RedHat's up2date and the RHN Panel Applet
3. Linux Debian's apt
4. Antivirus and security software updaters (almost all major security software vendors include now an automatic updater).

There are a number of issues still remaining, which are not addressed or are only partially addressed with these tools. Problems like "dependency hell," where each installed component or package depends on a number of others, which in turn depend on others, and so on, are not yet successfully resolved, although some of the tolls, such as apt, Yum, Urpmi, try to alleviate this problem through use of online software repositories.

Yet, automating delivery of new fixes, patches, and updates is just one part of the problem – ability to reconfigure the system on the fly, without interrupting service, or at

the time when such interruption will cause a minimal damage is another important factor in achieving self-configuring behavior. Ability to test a new configuration, and automatically roll-back changes (revert to the previous state), is another requirement, which must be included in an autonomic system implementing the self-configuring aspect. Keeping the system synchronized and preserving the state of components being replaced is another problem that must be dealt with while attempting dynamic reconfiguration of the system.

Self-configuration plays an important role and is closely related to other aspects of autonomic computing, since they rely on it. Self-healing, self-optimization, and self-protecting aspects, all depend on self-configuration. Self-configuration sits at the end of an autonomic process triggered by needs for system healing, optimization, or protection.

3.5 Self-Healing

A true autonomic computing system should be able to recover from failures and errors. This can be achieved either through a graceful degradation of services or more often, through detection, identification, and repair from a failure. Tasks involved in this process may include removing or replacing a faulty component, a module, or a node, shutting down or disabling applications or processes, isolating or disconnecting parts of the system, as well as such drastic measures as restarting or rebooting the parts of the system or the whole system.

Some examples of a simple self-healing behavior can be found in contemporary operating systems. For example, under MS Windows, OS service recovery mechanism includes the ability to instruct the service controller to attempt to restart the service, run a

specific program, or restart the computer on the first, second, or subsequent service failures.

More sophisticated mechanisms would need to include diagnostics, service adjustments, replacement of the faulty service or its component, all while the user is continuing to use the system oblivious to actions conducted by the system.

Self-healing behavior is especially important in mission critical systems and applications, such as the national infrastructure (power grid, communications, traffic control, etc.), space and undersea explorations, airplanes, ships, anywhere any error can result in a loss of lives and significant monetary damage.

Fault-tolerance and high-availability have been available in high-end systems for quite some time. However, there is still a great need for more sophisticated and more affordable solutions, applicable in the dynamic and heterogeneous environments. For example, Candea, Kicman, Zhang, Keyani, and Fox (2003) describe "JAGR – JBoss with Application-Generic Recovery," an autonomous self-recovering application server, based on application-generic failure path inference (AFPI), path-based failure detection, and micro-reboots on a sub-application level.

Sterritt (2003) explores pulse monitoring as a health-check for the autonomic grid. The same concept is also used in Sterritt and Chung (2004) applied on personal computing in peer-to-peer (P2P) environments. A number of other authors explore detection, failure diagnostics, and recovery mechanisms, all required elements for building a fully self-healing system.

3.6 Self-Optimization

Self-optimization may not look that important as self-healing and self-protection, but it ultimately speaks about the system responsiveness and ability of the system to perform its mission satisfactorily and satisfy its user needs. Self-optimization is closely related to quality attributes of the system: availability, accessibility, response time, performance, but also resource utilization and effectiveness. These attributes may be affected by demand, software or hardware failures, deadlocks or strained processes, and inadequate use of resources.

Currently, there are a number of applications that exhibit self-optimizing behavior. Load-balancers, bandwidth optimizers, virtual machines, operating systems, database management systems (DBMS) -- all have some elements of self-optimizing behavior. However, this behavior is generally based on a set of specific, narrow criteria and operates only with a limited number of local parameters and knowledge. In some cases, self-optimizing behavior of one application or a system may be harmful to other applications and systems and their users.

Self-optimizing techniques are generally based on cycles of monitoring and data collection, analysis, and recommendations for system adjustment. This is basically a "closed control loop technique," a concept originating from process control theory, which is also used in other aspects of autonomic computing.

Sadjadi, McKinley, Stirewalt, and Cheng (2004) describe a design of self-optimizing wireless network applications based on adaptation, behavioral reflection, and aspect-oriented programming. They use transparent reflective aspect programming

(TRAP), and TRAP/J, a programming model and language designed by the authors to support dynamic adaptation during the run time. Similarly, Martin, Powley and Benoit (2004) use principles of reflective programming as a mechanism for self-tuning of DBMSs. Their solution can be applied to any DBMS which supports dynamically adjustable configuration parameters. The approach taken by authors includes a method for automatically diagnosing performance problems and the incorporation of this method into current DBMSs using the concept of reflection.

Bennani and Menasce (2004) deal with the problem of autonomic performance management. They describe design of a quality of service (QoS) controller for highly variable workloads in terms of the inter-arrival time and service times of requests. In the design of their QoS controller, they incorporate workload forecasting techniques. Fontana (2004) focuses on optimization of component-based applications. He describes "The Component Balancer," software that statistically analyzes the relation between component methods invocations and how they affect each other, and then dynamically adjusts methods executions by introducing delays calculated via fuzzy logic. Loeser, Ditze, Altenbernd, and Rammig (2004) describe "GRUSEL," a self-optimizing bandwidth aware video on demand P2P application, based on the distribution factor and nodes characteristics as optimizing criteria and a rules tree used to execute optimization algorithm. Mahabhashyam and Gautam (2004) explore dynamic resource allocation of shared data centers. In their work they consider multiple classes of requests with varying needs and build an analytical model to calculate and compare the two performance

measures identified (loss probability of one class of requests and average delay of other class of requests).

Sadjadi and McKinley (2004) explore transparent self-optimization in existing CORBA applications. Their approach is based on the use of a generic proxy incorporated in "ACT," an adaptive middleware framework developed by authors. They use their approach in self-optimization of an existing image retrieval application, executed in a heterogeneous wireless environment.

Aiber et al. (2004) talk about "self-optimization according to business objectives," focusing on high-level business goals, such as maximizing revenues. This approach differs from more traditional optimization of computing environments that focuses primarily on service and resource availability, neglecting costs as its optimizing criteria. Similarly, Yiyu, Das, Gautam, Qian, and Sivasubramaniam (2004) explore the cost-effective autonomic control of web servers under time-varying request patterns.

3.7 Self-Protection

An autonomic computing system must be able to respond appropriately to protect itself against attacks, whether they are malicious or inadvertent (the result of a mistake or a miss-configuration). In order to do that the system must be able to recognize and identify threats and plan for a response to counter them. The system must be able to reason about unknown threats based on the knowledge of its environment and itself and on the past experience. Ultimately, the system must be able to learn from its past experience and build proactive measures and actions to achieve ultimate goals of self-

sufficient protection, providing security and integrity to its components, data, and resources.

Stojanovic, Abecker, Stojanovic, and Studer (2004) describe ontology-based correlation engines as a way to achieve self-protection mechanism. Based on user-defined rules, analysis of events, and the data collected from the environment, correlation engines play a significant role in detecting threats and intrusions and protecting a system from them. Correlation engines follow the basic Monitor-Analyze-Plan-Execute (MAPE) model outlined in Kephart and Chess (2003).

Sterritt and Hinchey (2005) focus on issues of self-protection in autonomic agents for space exploration missions. Based on the biological principles of self-protection in the human body, they first explore the possible scenarios in a system with mobile agents identifying the misuse of mobile agents by hosts, misuse of hosts by agents, and misuse of agents by other agents as potential problems. In the remainder of their work, they focus on a multiagent system for space exploration missions with agent destruction as an ultimate protection against "strained" agents operating outside of the correct context or agents failing to show emergent behavior within acceptable range of parameters.

It is important to note that the current state of the art focuses mostly on firewalls, intrusion detection systems, antivirus software, and less on building the elements of self-protection into the fabric of the system.

3.8 Dependency between Aspects of Autonomic Computing

The ultimate goal of autonomic computing of having computing systems take care of themselves is envisioned through fulfillment of four autonomic aspects: self-healing,

self-protecting, self-optimizing, and self-configuration. These aspects will require that a great deal of flexibility and sophistication be built into the system. These four autonomic aspects are interconnected and dependent on each other. The ability of the system to self-heal cannot be achieved without the ability of the system to reconfigure itself; the self-optimization of the system may require changes in the inner configuration and behavior of the system, and self-protection may also need elements of self-configuration (dynamically changing network structure, adding new nodes to strengthen defenses against an attack, shutting down or disconnecting affected parts of the network, etc.).

3.9 Chapter Last Remarks

In this chapter aspects and characteristics of the autonomic computing systems are explored. Examples of autonomic behavior in current computing systems, such as contemporary operating systems and applications, are analyzed. Each aspect is explored in more detail and some of the approaches used in their realization, as attempted by different authors, are presented.

CHAPTER: 4 Analysis and Design Methodology

4.1 Introduction

New concepts and methodology are desired to deal with problems posed by the autonomic computing challenge. This chapter explores some of those techniques and methodologies and their applicability in the design of autonomic computing systems, particularly those related to self-configuration. Roles of traditional object-oriented analysis and design, use of stable analysis patterns and frameworks, components, agent-oriented programming, event-driven programming techniques, aspect-oriented programming, adaptive knowledge bases and rules based systems, and adaptive middleware are explored in the context of autonomic computing.

In addition, technologies such as grid computing environment, service-oriented architecture (SOA), intelligent systems, artificial intelligence technologies (genetic algorithms, neural nets, fuzzy logic), petri nets, graph theory, ontologies, discovery mechanisms, web services, web services orchestration--all have promising roles in building autonomic computing systems.

Two different aspects are explored: designing software to support autonomic features and designing autonomic computing modules to adjust the behavior of current systems, which were not designed in the first place with autonomic features in mind. The first aspect looks into ways of how to build new software to support its self-management, and particularly, self-configuration, while the later aspect concentrates on ways to adapt existing software to include self-managing capabilities.

4.2 Traditional Object-Oriented Analysis and Design

Traditional Object-Oriented Analysis and Design (OOAD) play a major role in the design of today's software. Analysis and design processes traditionally include: defining problem statement, requirements elicitation, analysis, system design, object design, implementation, and testing. Artifacts produced by each phase include:

Analysis model:

1. Functional model (high level use cases, use case diagram)
2. Conceptual model, a.k.a. analysis objects model (conceptual class diagram)
3. Dynamic model, including system behavior model (system sequence diagrams, contracts for system operations), and analysis state model (conceptual state diagram)

Design model:

1. Design use case model (real use cases, use cases diagrams)
2. Architecture model - subsystem decomposition (package diagrams, deployment diagrams)
3. Object behavior model (interaction diagrams, contracts for methods and operations)
4. Class model (design class diagrams)
5. Design state model (state diagrams for classes)

These artifacts, including a test plan and implementation, are well suited for development of software which will be deployed and maintained by humans.

However, autonomic computing brings new challenges in designing and modeling software. Asynchronous messaging, events, control loops, decision making, adaptation,

exceptions handling and failure detections all include new elements which are not modeled in the best way using the traditional modeling notation.

Additional problems with using only traditional OOA & D in designing autonomic computing systems reside in the fact that a software architect can not anticipate all possible deployment scenarios and use of the software. Complexity of enterprise software environments, component interaction, dependencies, and maintenance issues appear later in the design, at the time of deployment, or at run time. Another problem is that in most of the cases, clear requirements will not be known at the design time of a system with autonomic features.

4.3 Stable Analysis Patterns and Frameworks

What is the relation of stable analysis patterns and frameworks and autonomic computing? Good programming techniques and principles which reduce dependency between components help design software, which is easer to maintain and update. Reducing coupling between classes by a careful design or through the application of principles such as Inversion of Control, increases quality of the software and makes it easier to adapt or reconfigure.

Stable analysis patterns and frameworks promote reuse and allow for building different systems, based on the stable core elements that capture fundamental knowledge about a specific domain. A software system based on stable analysis patterns and frameworks allows for easy modification and variation of "outer" layers of the system. Replacing components in the outer layers has no impact on the core functionality of the software as long as contracts between components are being maintained.

4.4 Agent-Oriented Programming

Agent-oriented Programming (AOP) is another promising technology that has found use in many aspects of system control, management, and distributed computing. Software agents naturally blend into the autonomic computing paradigm. Their characteristics are inherently suitable for building elements of autonomic behavior. Software agent properties (Autonomous, Adaptable, Knowledgeable, Persistent, Mobile, Collaborative) draw close parallels with some of the properties and characteristics required from autonomic computing systems. Some of the software agents express some of the aspects in a greater and some in a lesser degree. This entirely depends on their intended role in a system. Multiple software agents may be necessary in order to achieve desired autonomic behavior. A number of authors have explored the role of software agents in autonomic computing. Tesauro et al. (2004) describe an autonomic system based on multiple software agents, where each agent represents an autonomic element. Their system, called Unity, is guided by high-level policies, with optimal allocation of resources performed by calculation of resource-level utility functions and use of a Resource Arbiter element. They show an application of their architecture in managing data centers.

A multiagent system is governed by goals. Constraints in realizing those goals are resources. Achieving goals are ruled by beliefs. Beliefs themselves can be constantly updated and refined, based on the learning and interaction with environment. Which set of agents will be involved in reconfiguring the system depends on the current mission (current goals). In a multiagent system, each of the agents interacts with one or more

other agents in pursuing their tasks. Tasks can be given priorities guided by general policies and changes in the environment. An agent society acts in orchestration toward achieving a common goal dictated by specified policy and rules. Software agents cooperate and coordinate their actions to avoid conflicts and achieve common goals.

Guided by the Belief-Desire-Intention (BDI) principle, a software agent's behavior depends on its inner state and interaction with environment and other agents. This behavior can be quite simplistic, like in reactive agents, or complex, like in proactive agents, involving elements such as planning, learning, and negotiating.

A major problem with modeling multiagent systems is related to the lack of standardized methodology and notation. According to a FIPA modeling technical committee (Odell & Nodine, 2003), which approved a plan for development of a modeling standard for software agents (AUML), a number of different methodologies, developed by various authors are taken in consideration, but no standard is produced yet. Despite all of that, multiagent systems are natural candidates for building autonomic computing systems.

4.5 Adaptive Knowledge Bases and Rules Based Systems

Autonomic computing systems must make decisions with little or no human involvement. In order to make the right decisions, an autonomic computing system must operate with knowledge about the system itself, as well as about its environment. Detecting the problem, finding the cause, and choosing the right solution to fix it require significant amounts of operational knowledge and ways to infer conclusions based on the collected facts. Optimizing the system operation often requires consideration of different

criteria and calculation of optimization functions, or execution of optimization algorithms. Protecting the system involves, among others tasks, detection and identification of a potential attack or intrusion.

Knowledge bases are designed to represent knowledge and make access to that knowledge easy. An autonomic computing system may use a knowledge base as a way to manage its knowledge about its structure, functioning, anomalies, and its environment. Depending on the purpose and type of the autonomic computing system, a rule engine with production (inference) rules or a reactive rule engine can be also used to store the knowledge represented as facts.

It is important to note that the knowledge the autonomic computing system relies on is not static--actually quite opposite--it is in a continuous change. This implies that an autonomic computing system would need to constantly update its knowledge reflecting the environmental and internal changes. The strategies and methods for the knowledge acquisition, storage, retrieval and usage in these types of systems are the subjects of much research and study.

Cofino et al. (2003) explore knowledge management in autonomic systems. According to the authors, the initial knowledge about the system can be created based on functional specifications and requirements. Running self-identification tests before deploying the system will extend the initial knowledge. This initial knowledge is used as a "normal" state of the system. Deviations from this state may indicate problems in the system. Cofino et al. describe self-learning, self-monitoring, problem detection, self-diagnosing modules, and an adaptive architecture.

4.6 Event-Driven Programming

Observing the computing environment in which it is situated or examining its inner working is one of the core responsibilities of any autonomic computing system. Actions of an autonomic computing system are, therefore, largely driven by events detected in the environment or inside the system itself. A event-driven programming paradigm is thus naturally related to autonomic computing. Whether by pooling, sampling, or analyzing the data from the environment in an event loop, or reacting to changes through asynchronous interrupt handlers, an autonomic computing system utilizes event-programming techniques to acquire and process external and internal stimuli.

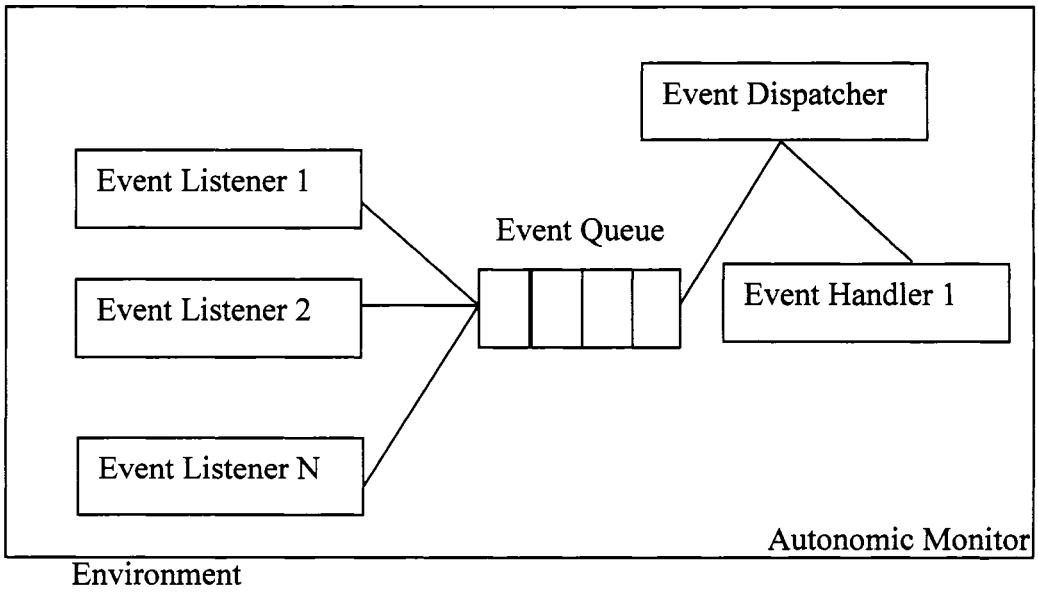

Figure 4. An autonomic monitor module modeled with event-driven architecture

Figure 4 depicts an autonomic computing monitor module modeled with typical event-driven architecture. Event listeners 1, 2,..., N continuously observe the

Environment waiting for a specific Event to occur. As soon as an event is detected, it is placed in an Event Queue, from where is taken by an Event Dispatcher and handed over to responsible Event Handler. Event Handler's task is to process the Event either by passing it to the analysis, or by making immediate decision. Note that event listeners can listen for changes in the environment, but can also be designed to detect events in the autonomic system itself.

4.7 Aspect-oriented Programming

Aspect-oriented Programming (AOP), coupled with reflection, is a promising technique that can be used in building autonomic computing systems. Better separation of concerns through modularization and encapsulation of cross-cutting concerns allow for easier adaptation of the software.

Several authors explore applications of AOP in the design of autonomic computing systems, and software adaptation to support autonomic features. Sadjadi, McKinley, and Cheng (2005), and Sadjadi, and McKinley (2005) use AOP as a tool for transparent shaping of existing software to support pervasive and autonomic computing. They propose a general programming model called "transparent shaping," based on AOP to promote adaptation of existing applications in dynamic environments. Engel and Freisleben (2005) use AOP aspects in a dynamic operating system kernel to introduce autonomic computing functionality. They argue that in most cases, autonomic computing aspects are cross-cutting concerns, and therefore use of AOP is a natural approach for their realization. They further introduce the TOSKANA toolkit for deploying dynamic aspects into an OS kernel.

4.8 Chapter Last Remarks

This chapter focused on various ways and techniques that can be utilized in the design of autonomic computing systems or realization of some of the aspects of autonomic computing. A number of promising technologies are already being explored and used by research community, and some of them are finding a way in "autonomic-like" features of software products already available on the market.

CHAPTER: 5 Challenges in Building Autonomic Systems

5.1 Introduction

There are a number of obstacles that have to be solved in analysis, design, and building of truly autonomic systems. Some of the challenges are of technical nature; some are scientific problems, and some are related to lack of standards in the area. Some of the problems in achieving autonomic computing goal are

1. Lack of a systematic approach to analysis, design, and architecture. The methodology is still in its infancy
2. Lack of tools (OS, compilers, program languages, etc.)
3. Lack of standards
4. Security issues (issues of trust, privacy, safety)
5. Heterogeneity of the environment
6. Scale of the problem

A formal, generic approach to the problem is required. Applicability of the current solution is limited, and some of the problems have been partially resolved. Encouraging is the fact that although the topic is very young, it has generated a significant interest, resulting in a wide spectrum of proposed solutions and ideas.

5.2 Dealing with Complexity

Paradoxically, response to the ever-increasing complexity of computing systems seems to be to make them even more complex. Autonomic computing introduces a new level of complexity into the computing system design and operation; however, this

complexity is ideally, hidden from end users. Autonomic computing allows for ubiquity of services provided by computing systems.

5.3 Dealing with Dynamics

An autonomic computing system must respond to a constantly changing environment. Adaptation is the basic principle used by autonomic computing systems to allow for dynamic adjustments of the system's operation and behavior. Operating on a set of dynamically adjustable parameters, using behavioral reflection, monitoring, analysis, and decision making process allows for adaptation of the system to the stimuli from environment.

5.4 Dealing with Heterogeneity

Today systems usually include many different platforms and configurations, dispersed across vast geographical areas or computing domains. Unifying protocols, such as TCP/IP, and high-level protocols, such as HTTP, RMI, and JMS, allow these systems to communicate and share data. Conforming to common protocol of interaction is a foundation for achieving autonomic computing goals in a heterogeneous environment. The problem is that the common protocols related to domain of autonomic computing are not yet established and a lot remains to be done on their standardization and acceptance. Some efforts focus on currently available technologies, such as web services, ontologies, semantic web, etc. as a way to increase interoperability between different systems.

5.5 Dealing with Legacy Code and Applications

Another challenge is due to an abundance of legacy code and applications. Legacy code and applications often depend on proprietary or outdated technology and are in many cases embedded deep in the cores of systems where any changes could result in costly downtimes. Dealing with this problem is usually done through adaptation or gradual replacement of the parts of the system. Adaptation is done through an injection of code and wrapping autonomic behavior around legacy code functionality.

5.6 Lack of Standards and Methodology

There are no currently available widely accepted standards for the design and building of autonomic computing systems. Various approaches and technologies are used, with varying success. There is a strong need for a formal, generic approach and unification of methodology in a standard, or a set of standards.

Methodology for designing and building autonomic computing systems is still in its early phase. There are many good ideas and promising approaches, but most of them focus on a narrow application or a particular aspect of the autonomic computing.

5.7 Chapter Last Remarks

In this chapter some of the problems and challenges facing designers and developers of autonomic computing systems were analyzed. The biggest obstacles are the lack of standardization and a systematic approach to the design and building of autonomic computing systems.

CHAPTER: 6 Self-configuration – Domain Analysis

6.1 Introduction

Complexity of the software is constantly on the rise. Applications, systems, and networks are growing bigger and bigger, and soon nobody will be able to grasp all the elements, monitor them, and respond to emergencies and changes in the environment. Tracking problems and recovering from them will require a great deal of self-configuration and ability of the system to reconfigure itself.

6.2 Need for Self-configuration

There is a range of systems and applications that need the ability to dynamically change their configuration as a response to events from the environment. The extent of this configuration may span anything from a few components to a number of nodes in a network. For example:

1. Reconfiguring a power grid to save the network from chain failures (ripple effect) by shutting down nodes or isolating the parts of the network affected. Dynamically re-balancing load and redirecting demand.
2. Reconfiguring a computer network under a heavy load, different demands on the parts of the network, or failure of equipment (hubs, routers, switches).
3. Reconfiguring a computer network to deal with security problems (security breach, worm, or virus infection) by shutting down or isolating network nodes, or locking down portions of the system or the network.
4. Adaptive (self-configuring) firewalls, adjusting their configuration and policies according to appearance of new nodes discovered on the network.

5. Updating an application with new components, replacing faulty components or introducing new versions of components providing new features or better performance.
6. Adding new worker threads in an application, based on the dynamically changing demand.

6.3 Different Aspects of Self-configuration

Self-configuration can be analyzed from different aspects. The size, scope, extent, and context all play a role and define different aspects of self-configuration. Concepts, such as assembly, organization, negotiation, and awareness, all play roles in self-configuration. However, some of the aspects are at the core of the self-configuration domain. The following text attempts to identify and analyze those aspects.

6.3.1 Size and complexity

The size and complexity of a system plays a major role in the manageability of the system. A software component can be composed of one or more classes or modules; an application can be built from a few or thousands of components; a computing system may be running a few dozen or hundreds of applications; a network can be comprised of a few or thousands nodes. Based on the size and complexity, self-configuring systems can be categorized in self-configuring:

1. Components
2. Applications
3. Devices
4. Computing Systems

5. Networks

The size and complexity of the system have implications on the selection of an approach used for the implementation of autonomic aspects.

6.3.2 Type

Different levels and types of self-configuration are possible. The following problems include elements of self-configuration:

1. Automatic software setup and installation
2. Automatic configuration, without or with minimal involvement of administrator
3. Reconfiguration triggered by changes in the environment
4. Automatic updates

6.3.3 Level

Self-configuration can be done on different levels. The following levels have been identified:

1. Configuration through configuration parameters
2. Configuration through changes in the inner structure of the configured system
3. Configuration through a mix of configuration parameters that change and a change in the inner structure of the configured system

6.3.4 Static versus Dynamic Configuration

Most devices and applications receive events and changes from the environment. Applications need to be implemented in such a way to dynamically respond to changes and reconfigure themselves. The idea is to build applications which will be self-

sustainable, able to react to any significant events in the environment and adapt to changes without or with minimal human involvement.

Self-configuration will often require reboot, or reloading of the configured system or application. However, in some cases, this is undesired behavior. This is especially true for systems of vital importance performing critical jobs that must be available all the time. These types of systems cannot afford to be reloaded or rebooted. Their configuration must be done while they perform their tasks without interruption of services they provide. Requirements of continuity of operation and service make the design of such systems much more difficult to implement. Usually, this is done through redundancy and implementation of "hot-swapping" components and elements which can be replaced on a live system. However, to be fully autonomic, such systems also need to include component monitoring, failure detection, and identification and planning of adequate response (component replacement, parameters change, etc).

6.3.5 Discovery

Service discovery is an important part in the dynamic environments. New services and capabilities appear and vanish during this time. New network nodes, devices, web-services, and applications appear and disappear. A number of standards were proposed to deal with the problem of service discovery. In the core, most of them are based on a simple architecture of dictionaries and services advertising their capabilities.

In order to be found and offer its capabilities, each new application, service, or device has to conform to the process of registration with a directory service.

Applications or devices that do not subscribe to this model are still able to offer their services by providing a way for interested parties to query their services on a peer-to-peer basis. However, this process usually involves scanning (searching) for devices or applications and would take a considerable more time and resources. Most of the current wireless systems work exactly on that basis. For example, cell phone devices search for a signal of the providing station, establishing a connection after the initial protocol is executed and a device has registered with the network. A wireless card's software instructs the card to scan for a wireless signal, enumerating all available networks and allowing the card to connect to one based on the user preferences or on a predefined policy. Here, a policy could be to "connect to an unsecured network with the strongest signal which offers the highest possible speed that a card can use." Obviously, this policy has a complex criterion that is composed of three distinctive elements: unsecured network, strongest signal, and the highest possible speed. To summarize, the following ways of discovery can be employed:

1. Monitoring services by checking the environment (scanning or pooling).
2. Device or service is signaling its presence (e.g., "pulse") or finding a service to subscribe or register itself, advertising its capabilities.
3. Attached RFIDs or other wireless-based technology can be utilized to signal a device's presence and provide additional information.
4. At the time of plug-in (current plug and play can be improved) a device does not need to carry drivers and software for its use, just necessary information (instructions) where those can be obtained (downloaded).

6.4 Installations

A vast majority of contemporary software is currently installed through some sort of offline or online installer. In most cases, software installers require the user's attention and response and will install selected options in specified locations and configure the software according to the user's responses. Some of the tools used to create application installers are

1. Macrovision InstallShield
2. Microsoft Installer
3. Zero G InstallAnywhere
4. Open source Nullsoft Scriptable Install System (NSIS)
5. Shell scripts

Installers play an important role in easing the process of deployment and the initial configuration of the software on a target system. A typical installation may involve a number of tasks performed on the software being installed or on the environment under which the software will be running. Some of more common installation tasks include:

1. Verifying hardware and software prerequisites
2. Detecting previously installed versions of the software
3. Unpacking the software distribution
4. Copying files in the designated locations
5. Registering files, services, or DLLs
6. Setting up environment variables
7. Creating the desktop and menu shortcuts

8. Executing installed executables
9. Restarting the system
10. Uninstalling the software

Other possible packaging and installation of software includes creation of unattended or silent installations, often used by administrators for deploying software on a large number of computer systems. The same method is also used as a preferred method of installing service packs, patches and fixes, and freeing a user from interaction with the installer. In this type of installation, most of the configurable parameters or user responses have been already pre-selected by an administrator or a software vendor delivering a software update.

6.5 Configuration of the Software

Modern software often includes the ability to modify its behavior based on the set of configurable parameters. Software may include different levels of customization, personalization, and tuning. The software design will usually externalize configuration parameters in one of the following ways:

1. External configuration files (XML, text files, Windows.ini files or other)
2. Registry
3. Environment variables
4. Command switches
5. Proprietary, binary files, and structures
6. Configuration databases
7. Directory services.

For example, most of the web servers include a number of configuration parameters, such as configurable port number, number of threads (minimum and maximum), connection timeout, session expiration, memory usage, etc. Contemporary database systems include tens, even hundreds of configurable parameters for fine tuning and configuration. This allows for adjusting to various uses and purposes and better performance. It is important to note that most of these parameters are manually configured or selected and, therefore, involve human intervention. However, some of the newer servers and database systems include a level of the dynamic self-adaptation based on traffic or load monitoring or statistics of usage or query collection.

6.6 Reconfiguration

Reconfiguration is based on a set of rules, which can be defined in a rule-database or implicitly embedded in the nodes. Decisions made are a part of a deliberation process, or in the case where a fast response is required, the decision is often a reactive response. Changes in the environment are detected through sensors, which are attached to event listeners. These elements, which are often implemented as software agents, are programmed to track changes of specific environmental parameters and react on a breach of preset thresholds. Based on the input from these elements an appropriate action is taken. Further data collection might be ordered; a planned procedure might be executed, or in the case when immediate action is needed, a reactive response can be chosen. There should always be a default rule about what to do in the case everything else fails. This would be a bottom-line strategy in the case all other responses or paths are exhausted. An analogy in programming would be a top level exception handling – a code invoked

when all other exception handlers fail to catch and handle an exception. This whole process is in its core designed after a control feedback loop model.

The feedback control or regulation loop is a well known concept in automation and control theory, allowing for corrective actions applied on a system based on the measured outputs and reference values. Its basic principle is very simple and is based on continuous correction of an observed parameter value. Figure 5 shows a typical control feedback loop. A control feedback loop includes input value x(t) which is usually observed through sensors and which changes over time and reference value y(t) which is fed into a comparator C, Δ which is the difference between reference value y(t) and input value x(t) (usually called an error signal). This value is used by actuator A to adjust the observed value.

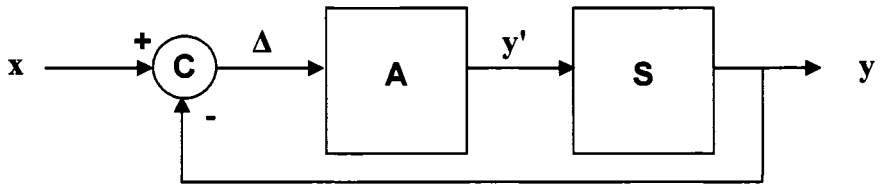

Figure 5. Control feedback loop.

The control feedback loop is used in a number of control and automation systems, for regulation of flight, car speed, water level, room temperature control, etc.

The same control feedback loop principle is used in autonomic computing where it is called autonomic control loop (Figure 6). The managed system is observed through one or more sensors and changed through effectors.

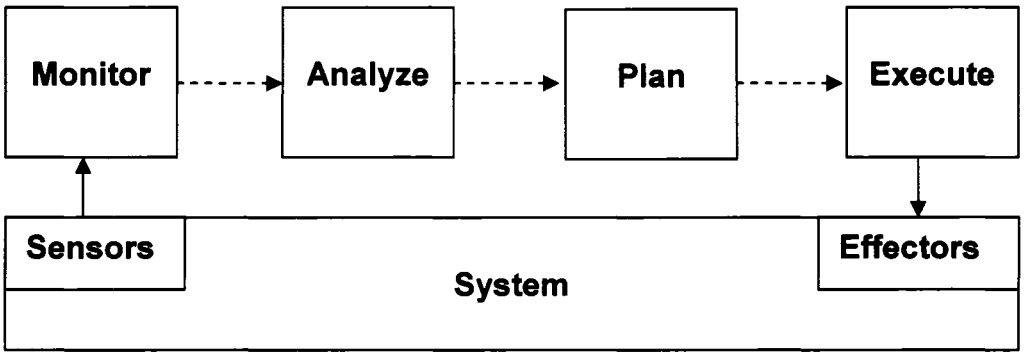

Figure 6. Autonomic control loop.

An autonomic control loop usually includes modules for system monitoring, analysis of the events or property values, response planning, and response execution. When looking into what type of monitors can be included in a typical computing system one may recognize:

1. Environment Monitor
    a) Resources Monitor--monitors server's disk space, free memory, CPU usage.
    b) Network Monitor--monitors computing system's wider space – network traffic, connections, communication with other systems.
    c) Process Monitor--monitors behaviors of other processes sharing resources in the system.
2. Application Monitor--monitors application up-time, availability, response time, application logs, errors, exceptions, and other messages (debug, info, etc.).

3. DB Monitor–a specialized kind of an application monitor–in addition to the above parameters, it also monitors database parameters: table space, CPU usage, memory usage, and DB logs.

Monitoring tools provide input to a decision engine, which runs collected events through a set of rules and decides whether a problem is in progress, whether it might occur or has already happened, and what has to be done to solve a particular problem.

This loop can be used to reconfigure a system to adjust to the environment. Reconfiguration may involve a number of different tasks, such as loading or unloading software modules, shutting down parts of the system, disconnecting or re-connecting parts of the system, adding new parts or modules, applying new policy, making new resources available, stopping or starting application, etc.

6.7 Chapter Last Remarks

Self-configuration involves different aspects of configuration, installation, updates, system control and design. This chapter reflected on some of those aspects and tried to identify some of the major concepts that will be used in the following chapter.

CHAPTER: 7 A Framework for Self-configuration

7.1 Introduction

This chapter focuses on the design of a framework for self-configuration. Different design approaches are explored, and a design based on software agents is proposed. A methodology based on the Gaia approach for analysis and design of multiagent systems is utilized in identification of software agents, their roles, and interactions.

7.2 Self-configuring Applications Design

The ability of a computing system to reconfigure its inner structure implicates that the system has to be built out of parts which can be removed, added, or replaced without affecting the system's functionality. This can be achieved through designing the system or application as an assembly of reusable elements (blocks), which can be replaced with other elements providing the same functionality and conforming to the same "gluing." Using "hooks," configurable links, to "glue" elements in an application creates a flexible architecture which can be easily reconfigured. Thinking of applications as constructions built of "LEGO" elements implies the ability to arrange elements in many different ways to build many different applications or different versions of the same application. Java interfaces, COM components, CORBA bus, and other similar technologies facilitate creation of reusable software elements, used by many different applications (see Figure 7). In networking, TCP/IP acts as a gluing mechanism for very different devices and systems. What is important for all of these technologies is that they are defined and adopted standards, allowing for the interoperability between all systems or applications that are conforming to the same standard or protocol.

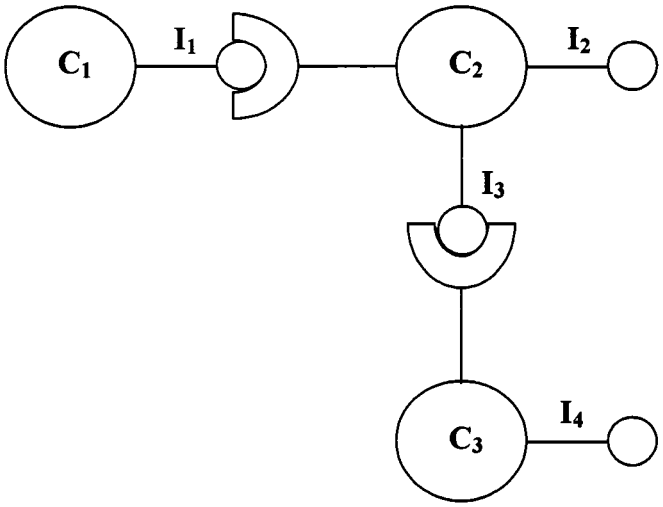

Figure 7. A system comprised of components, glued through interfaces.

Ideally, every application is created in such a way that is composed of a number of building blocks, replaceable components, interacting with each other. Each component can be a "client" or a "server," providing services (capabilities) to other components, or using services (capabilities) of other components. Components create a network of dependency, which can be represented in a graph or a table. A desirable characteristic of an application is to minimize dependency between components–a degree of coupling. The effects of replacing a single component or a single misbehaving component would then have little or no impact on overall application, or would render only a part of the system or application unusable instead of bringing down the whole system or application. Creating a stable application core resembles creation of a stable building foundation in civil engineering. Important questions are how much is the rest of the system dependent on the foundation, and should the foundation also be flexible and to what extent? Another analogy from the civil engineering field is buildings built on a

rubber support, which allows them to move in the case of an earthquake, preventing their crumbling.

With a stable core in place doing an "interior design" of remodeling an application or its façade is much easier. Windows and doors can be replaced; rooms can change the purpose, and generally, an application can be modified without affecting its core functionality or purpose.

7.3 Possible Architectures

In Chapter 4 different technologies that can be used in building autonomic computing systems were identified. This section looks into different possible architectures for designing a framework for self-configuration. A self-configuration framework could be created as:

1. A middleware totally decoupled from application. Applications only implement interfaces toward self-configuring middleware, allowing them to use the services of the middleware. These services can be implemented as a set of API calls or a web-services API.
2. Through a layer that applications would subscribe to–using observer pattern, event notification, or interrupts.
3. Through a middleware built with aspect oriented programming (AOP).
4. As an interface implemented by application.
5. Embedded inside the application (the last flexible).
6. Multiagent system with hooks into the system or application.

The proposed framework would create a middleware that applications and components would use to configure, reconfigure, and update themselves. The layer itself is responsible for tasks such as:

1. Context-awareness
2. Service discovery
3. Protocol negotiation
4. Interacting with the environment on behalf of the device or application
5. Security
6. Providing self-configuration capabilities to devices or applications

7.4 Design of a Framework for Self-configuration

The main aim of the design is to create a framework which can be applied in a variety of self-configuration scenarios. The framework tries to capture core knowledge related to configuration, reconfiguration, and self-configuration, by identifying enduring business themes (EBTs), and corresponding business objects (BOs). A stable, pattern-based architecture is proposed, and several scenarios of framework applications based on real problems are explored by varying Industrial objects (IOs).

An implementation of the framework with software agents is proposed, by defining the roles of each agent, and inter-agent communication. Underlying infrastructure and utility modules are also defined. In the process of definition and design of the framework, the following design and analysis steps are used:

1. Perform domain analysis–create a set of requirements (high level)
2. Create use cases (high level)

3. Build a conceptual model (static)
4. Transfer the model to a set of CRC cards
5. Create a static model--class diagram
6. Identify roles
7. Map roles to agents
8. Create a dynamic model. Define agents' communication
9. Create a deployment diagram

7.4.1 Description of the system that is wanted

A framework for self-configuration is a software framework, consisting of multiple agents, the purpose of which is to provide a base for building applications with self-configuring capabilities.

7.4.2 Requirements

Based on the domain analysis and the scope of this project, a set of requirements is identified:

R01: The self-configuration framework shall allow an application to dynamically load a new configuration.

R02: The self-configuration framework shall allow for managing configuration policies (Add/Remove/Modify).

R03: The self-configuration framework shall allow for managing an application's or a system's configurations (Add/Remove/Modify).

R04: The self-configuration framework shall provide facilities for control actions on a system or application (stop, start, resume, load, unload, reset, reboot).

R05: The self-configuration framework shall allow for reconfiguration of an application or a system by adding new capabilities or resources.

R06: The self-configuration framework shall allow for reconfiguration of an application or a system by removing its capabilities or resources.

R07: The self-configuration framework shall allow for reconfiguration of an application or a system by modifying its configuration (either through the configuration parameters or by replacing its components).

R08: The self-configuration framework shall allow for tracking of changes (configurations) of an application or a system.

7.4.3 Use Cases

The following use cases describe some of the configuration scenarios. Note the commonalities between use cases–there are only slight variations, depending on the interpretation of the ConfigurationPlan.

UC01: The Monitor detects a change in the environment and informs the Analyzer about the change. The Analyzer consults the Policy and prepares a ConfigurationPlan, then initiates the Configuration. The Configuration interprets the ConfigurationPlan and directs the Configurator to add a new Component to the Entity. The Configurator adds the Component and requests from the Controller a reload of the Entity.

UC02: The Monitor detects a change in the environment and informs the Analyzer about the change. The Analyzer consults the Policy and prepares a ConfigurationPlan, then

initiates the Configuration. The Configuration interprets the ConfigurationPlan and directs the Configurator to remove the Component from the Entity. The Configurator removes the Component and requests from the Controller a reload of the Entity.

UC03: The Monitor detects a change in the environment and informs the Analyzer about the change. The Analyzer consults the Policy and prepares a ConfigurationPlan, then initiates the Configuration. The Configuration interprets the ConfigurationPlan and directs the Configurator to replace the Entity's Component. The Configurator replaces the Component and requests from the Controller a reload of the Entity.

UC04: The Monitor detects a change in the environment and informs the Analyzer about the change. The Analyzer consults the Policy and prepares a ConfigurationPlan, then initiates the Configuration. The Configuration interprets the ConfigurationPlan and directs the Configurator to set ConfigurationParameter of the Entity. The Configurator sets the ConfigurationParameter and requests from the Controller reload of the Entity.

7.4.4 Conceptual Model

The process of domain analysis involves the creation of a dictionary that includes domain concepts and their definitions. Table 1 represents the domain concepts that have been identified for the domain of self-configuration. These concepts help the creation of a conceptual model, which includes main concepts and relations between them. Some of the concepts captured are identified as entities in the model while others represent attributes. The creation of a conceptual model follows the basic steps of software analysis phase in traditional OOA & D (Larman, 2004).

Table 1. Configuration Concepts

Concept	**Description**
Configuration	Describes the configuration process
Configurator	Role which executes the configuration
Entity	An entity to configure
Component	A part of the entity
Interface	A component's contract with the environment
Configuration Plan	A plan that guides the configuration
Configuration Step	Part of the configuration plan
Configuration Parameter	A value that influences behavior of the entity
Controller	Role which controls the entity
Downloader	Retrieves components and configurations
Monitor	Monitors the environment and entity for changes
Analyzer	Analyzes changes and creates configuration plans
Policy	Sets rules of the configuration
Policy Manager	Manages configuration policies
Repository	Holds components and configurations
Updater	Role that updates the entity

The relation between the main concepts is established and presented in a conceptual model (Figure 8).

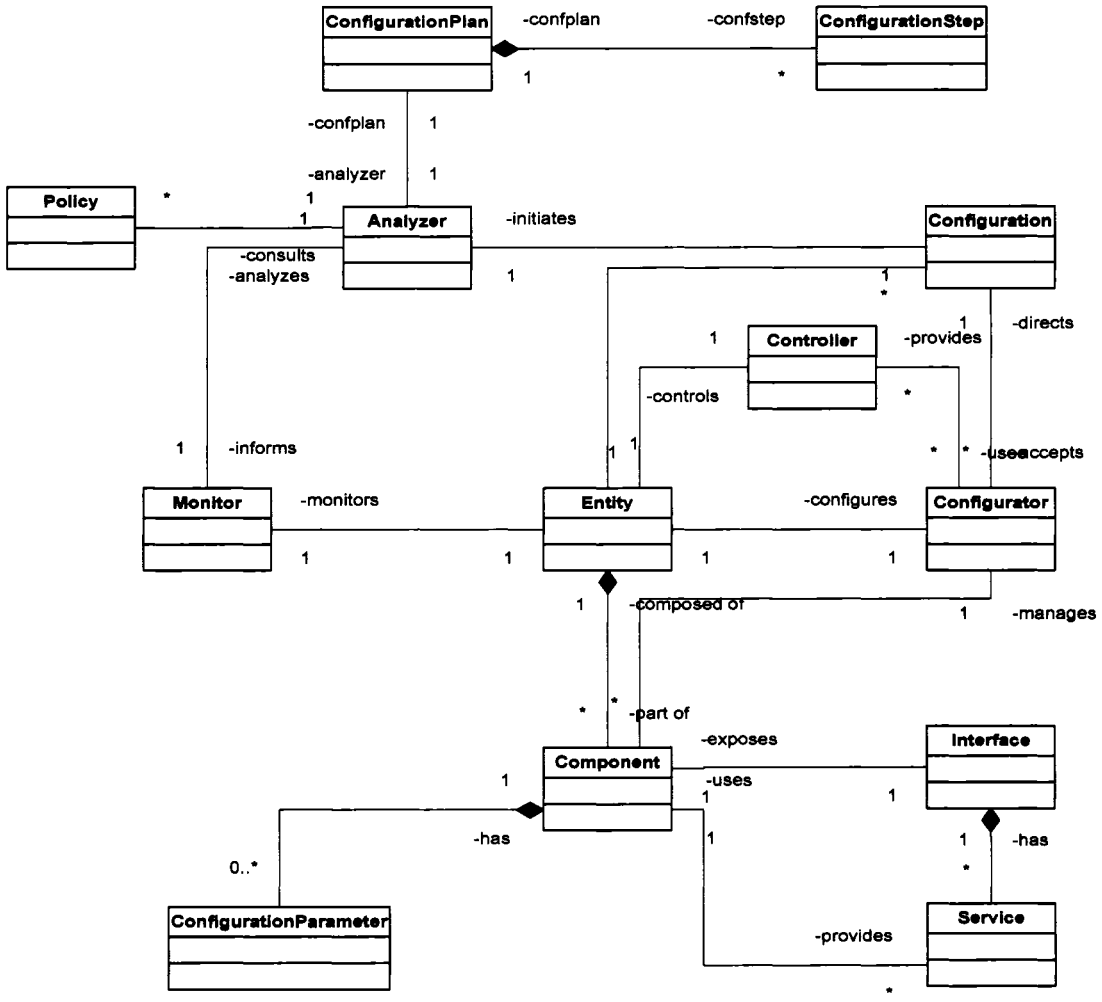

Figure 8. Conceptual model.

7.4.5 CRC Cards

The conceptual model is further expanded by the creation of CRC cards for each defined concept. Each concept in the model is analyzed and its responsibilities and collaborations are established. Table 2 represents the CRC card for the Entity. The Entity is an abstraction which represents a system, an application, or a device – an entity to be configured. The Entity is comprised of components and may have associated configuration parameters used to modify its behavior.

Table 2. CRC Card for Entity

Entity		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for providing status information and operation indicators.	Monitor Configurator Controller	setup() assign() query() apply()

The Configuration initiates, guides, and manages the whole configuration process. To accomplish these tasks the Configuration establishes collaboration with the Analyzer and the Configurator (Table 3).

Table 3. CRC Card for Configuration

Configuration		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for handling a configuration process.	Analyzer Configurator	initiate() activate() suspend()

The Monitor is responsible for observing an Entity and registering changes which may require Entity's reconfiguration. The Monitor collaborates with the Analyzer in the process of initiating a configuration. Table 4 represents the Monitor's CRC card.

Table 4. CRC Card for Monitor

Monitor		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for monitoring of the Entity.	Entity Analyzer	initiate() monitor()

Table 5. CRC Card for Policy

Policy		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for managing a configuration policy.	Entity	initiate() add() remove() activate() suspend()

The Policy guides a configuration process. In general, a goal policy is based on specified goals. The goal policy is on a higher level of abstraction, since it is not concerned about how a goal will be attained. Its focus is on the goal without having knowledge about inner workings of the system. Since the system can not directly operate with goal policies, there is a need for translating goal policies in a set of action policies that are based on condition-action rules.

For example, a policy can be set to update all systems as soon as a new update is available. This could translate into a sequence of actions (Figure 9). The Configurator is responsible for the execution of a ConfigurationStep, adding, removing, or replacing a part (Component) of an Entity, or setting an Entity's or a Component's configuration parameter.

```
IF new_update_available(Entity) THEN
    update(Entity)
    IF !run_test(Entity) THEN
        rollback_update(Entity)
    ENDIF
ENDIF
```

Figure 9. Example of execution rules.

Table 6 depicts the Configurator's CRC card.

Table 6. CRC Card for Configurator

Configurator		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for configuration of the Entity.	Entity Component	add(Component) remove(Component) replace(Component, Component) set(Component, Parameter) verify()

The Component represents abstraction of a component of an application or a system that can be added, removed, replaced, or configured. The Component may have sub-components and defines its interaction with the environment via its interface.

Table 7 shows a CRC card for the Component.

Table 7. CRC Card for Component

Component		
	Collaborations	
Responsibility	Clients (Role)	Server
Represents a component of the Entity and provides services to other components.	Configurator Entity	dependencyList() getInterface()

Each component or module should identify itself by a unique identifier and provide the ability for querying version information. It should also specify which services are provided to prevent the situation of rendering inoperable a part of the system by reducing the services previously offered by a similar component or a different version of the same component. Ability to query and test the component or module is essential for successful management.

Table 8 depicts a CRC card for the Interface, which defines a set of services that are provided by the Component.

Table 8. CRC Card for Interface

Interface		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for describing set of services provided by the Component	Component	getServices()

Table 9 depicts a CRC card for the Controller which is responsible for execution of control actions on the Entity.

Table 9. CRC Card for Controller

Controller		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for control of the Entity	Configurator	stop(Entity) start(Entity) pause(Entity) resume(Entity) restart(Entity)

The Configuration consists of one or more configuration steps. Configuration steps execute in a sequence; they may branch based on conditions, loop, etc. The execution of configuration steps is usually done through administration scripts using various scripting languages, like shell scripts, batch scripts, Perl, awk, etc. Table 10 depicts a CRC card for the ConfigurationStep, a part of the self-configuring framework which defines a basic element of configuration.

Table 10. CRC Card for ConfigurationStep

ConfigurationStep		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for handling a basic unit of configuration.	Configurator	initiate() apply()

Typical configuration steps may involve following:

1. Replace a component or module (replace a part of the system)
2. Add (install) a new component or module (a part of the system)
3. Remove or uninstall a component or module (remove or isolate a part of the system)
4. Stop a service or application
5. Start a service or application
6. Pause or resume a service or application
7. Disconnect or connect a part of the system
8. Acceptance process
   a) Testing
   b) Evaluating results
   c) Accepting process
   d) Roll-back

Although not a part of the framework design (conceptual model) per se, some additional elements related to the framework are identified and analyzed. These elements are necessary for the functioning of the system proposed in this work. Those elements are

related to the tasks of retrieving configurations and components from a storage and the scheduling of configuration tasks. A repository of components and configurations is required to keep physical files and information about components, their versions, and relations. Accessing this repository for the purpose of retrieving the files is assigned to the role of the Downloader. Table 11 depicts the CRC card for the Downloader.

Table 11. CRC Card for Downloader

Downloader		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for retrieving components and configurations.	Configurator	download(Component)

Scheduled configuration or execution of specific configuration steps or tasks is needed in those cases when execution of configuration steps would have a negative impact on the operation of the Entity (an application, a device, or a system).

Table 12. CRC Card for Scheduler

Scheduler		
	Collaborations	
Responsibility	Clients (Role)	Server
Responsible for scheduling execution of the Task.	Task	schedule(Task) run(Task) cancel(Task) delete(Task)

7.4.6 Class Diagram

Figure 10 depicts the framework's class diagram. All major elements of the framework are shown with some of the supporting classes. There is also a number of

other support implementation classes that would need to be included in the full design but that are not shown here for the purpose of brevity.

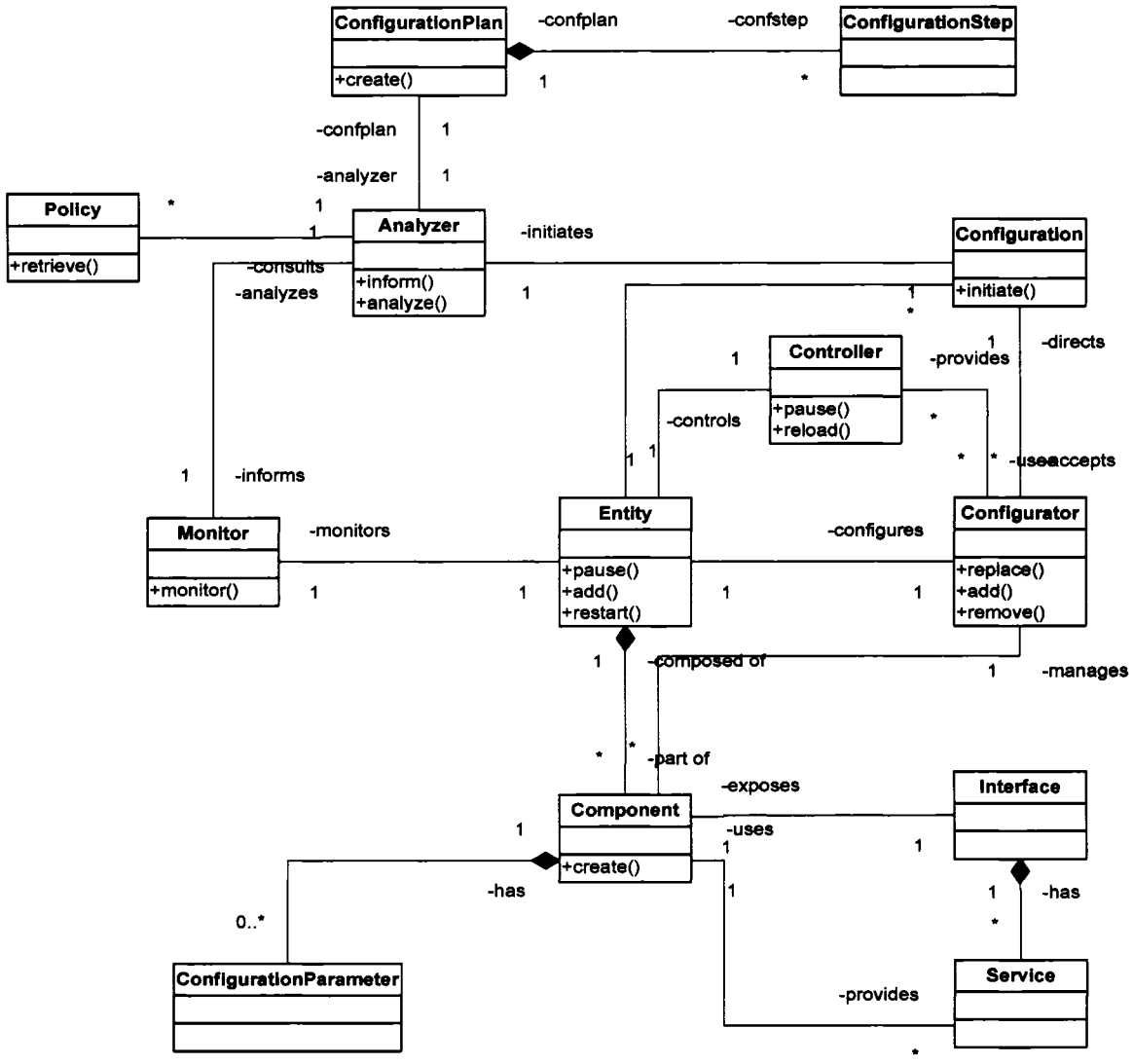

Figure 10. Class diagram.

7.4.7 Mapping Roles to Agents (MAS System for Self-configuration)

Different methodologies and languages are proposed for designing and modeling agent systems. Most of them are based on an extension of standard Unified Modeling

Language (UML) or Rumbaugh's Object Modeling Technique (OMT). This work uses elements of Gaia AOSE methodology, which is based on roles and described in Wolldridge, Jennings, and Kinny (2000) and Zambonelli, Jennings, and Wooldrige (2003). A general schema of Gaia methodology is presented in Figure 11.

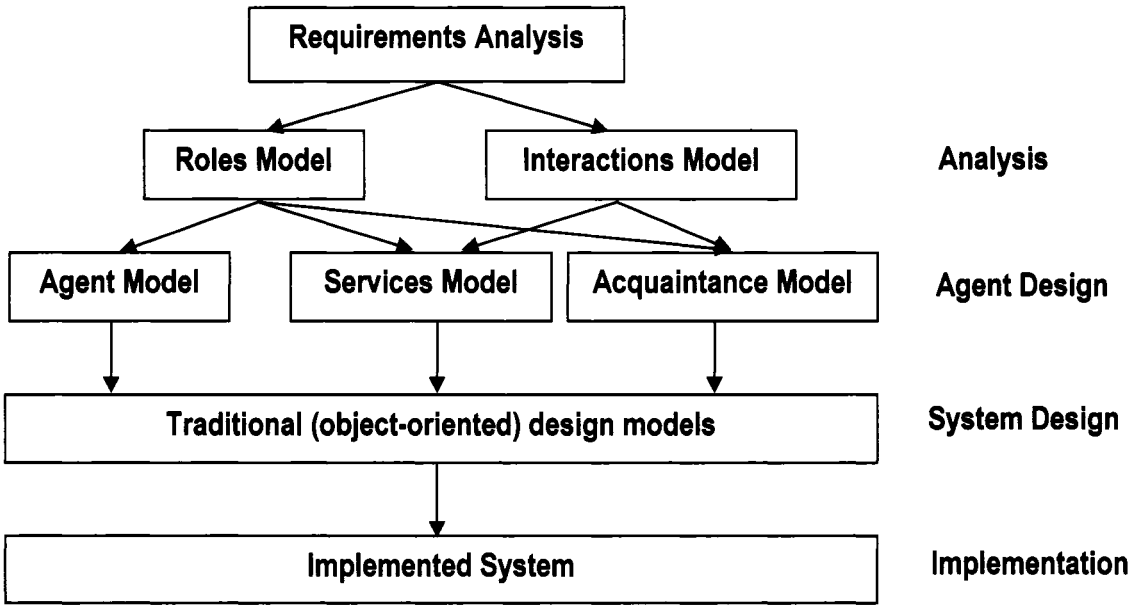

Figure 11. Gaia methodology for AOSE.

The conceptual design and roles identified are further mapped to software agents. Each software agent is responsible for a single role. The end result is an Organizational Model (the abstract description and analysis of all roles involved in a system). The model is expressed as a textual description of how the use cases are realized in terms of collaborating objects (agents).

Each software agent must, on its startup, register with a directory service and announce its capabilities, name, and address. The capabilities of an agent are expressed as a set of services that it can perform or provide. Services provided can be used by other

agents, external systems, or humans. To use services, a requester must contact the register to learn about available agents, their addresses, and capabilities. The register (a.k.a. Yellow pages) service is not modeled and is provided by the agents' platform.

Table 13 through Table 17 describe all the major roles (Monitor, Analyzer, Configuration, Configurator, Controller) in the framework. Each role is described according to the Gaia notation and specifies all protocols and activities that each role participates in, as well as its responsibilities (liveness and safety) and its permissions.

Table 13. Schema for Role Monitor

Role Schema: MONITOR
Description: This role involves monitoring of a change in assigned environmental parameter.
Protocols and Activities: Accept, Monitor, InformChange
Permissions: reads: *Parameter* *// parameter to monitor*
Responsibilities
Liveness: MONITOR = (Accept.Monitor.InformChange)$^{\omega}$
Safety: • true

Table 14. Schema for Role Analyzer

Role Schema: ANALYZER
Description: This role involves receiving a notification from the Monitor, consulting the Policy, analyzing the change, creating the ConfigurationPlan and initiating the Configuration of the Entity.
Protocols and Activities: ReceiveNotification, ConsultPolicy, AnalyzeChange, CreateConfigurationPlan, InitiateConfiguration
Permissions: reads: *Property, Value, Policy // for monitored Entity* creates: *ConfigurationPlan*
Responsibilities
Liveness: ANALYZER = (ReceiveNotification.ConsultPolicy.AnalyzeChange.CreateConfigurationPlan. InitiateConfiguration)$^{\omega}$
Safety: • true

In the Gaia methodology, protocols establish the relation between a role and other roles in the system, while activities define actions performed by the role which do not require collaborative participation of other roles. Note the similarity of roles in the Gaia methodology with CRC cards defined earlier. Both, CRC cards and the roles in the Gaia

methodology capture essential information about a concept in the system, a collaboration, and responsibilities.

Table 15. Schema for Role Configuration

Role Schema: CONFIGURATION
Description: This role directs a configuration process. It receives the ConfigurationPlan from the Analyzer and directs its execution.
Protocols and Activities: ReceiveConfigurationPlan, DirectConfiguration, Notify
Permissions: reads: *ConfigurationPlan*
Responsibilities
Liveness: CONFIGURATION=(ReceiveConfigurationPlan.DirectConfiguration.Notify)$^{\omega}$
Safety: • true

Although in the core very similar, by its intent and information captured in them, CRC cards and roles in the Gaia methodology have some significant differences. Roles in the Gaia methodology introduce permissions attribute--a new element used to describe what a role is allowed to do and what is not, which is different from CRC cards. In addition, responsibilities are expressed not in a narrative form, but through specific notation via two categories: liveliness and safety responsibilities. While liveliness responsibilities state that "something good happens" (express that "something will be

done," and consequently, that the agent representing the role is still alive), safety responsibilities involve requirements imposed on a role to satisfy and maintain specific invariants.

To express liviness responsibilities, Wolldridge et al. (2000) use a notation similar to the life-cycle expression of the fusion method (Coleman et al., 1994), adding an additional operator ώ, which describes infinite repetition. At the same time, safety responsibilities are expressed as a list of predicates applied on the variables specified in a role's permissions.

Table 16. Schema for Role Configurator

Role Schema: CONFIGURATOR
Description: This role involves execution of configuration activities.
Protocols and Activities: ReceiveActivity, ExecuteActivity, Inform, RequestControlAction
Permissions:
Responsibilities
Liveness: CONFIGURATOR=(ReceiveActivity.ExecuteActivity. [RequestControlAction].Inform)+ ^{ώ}
Safety: • true

Table 17. Schema for Role Controller

Role Schema: CONTROLLER
Description: This role involves execution of control activities on the Entity.
Protocols and Activities: ReceiveActivity, ExecuteActivity, Inform
Permissions:
Responsibilities
Liveness: CONTROLLER=(ReceiveActivity.ExecuteActivity.Inform)+ $^{\omega}$
Safety: • true

The next step of analysis in the Gaia methodology is to define the protocols of interaction between the software agents. The protocol of interaction focuses on high level purpose of the interaction, rather than on the precise ordering of particular message exchanges (Zambonelli et al., 2003).

Table 18 and Table 19 depict the InformChange and InitiateConfiguration protocols of interaction of the Monitor and the Analyzer role, respectively. The remaining interaction protocols between software agents are easy to define; however, for the purpose of brevity, they are not shown here.

Table 18. InformChange Interaction Protocol

Protocol Name:		
Inform Change		
Initiator:	Partner:	Inputs:
Monitor	Analyzer	Parameter, Value
Description:		Outputs:
When the Monitor detects a change in the environment, it has to start a protocol to inform the Analyzer.		

Table 19. InitiateConfiguration Interaction Protocol

Protocol Name:		
InitiateConfiguration		
Initiator:	Partner:	Inputs:
Analyzer	Configuration	ConfigurationPlan
Description:		Outputs:
When the Analyzer initiates the configuration process, it has to start a protocol to inform the Configuration about the ConfigurationPlan.		

The design of an agent system in the Gaia methodology (Figure 11) is defined with Agents, Services, and Acquaintances models. Figure 12 shows the agents model which maps the roles to agents. In this case, each role is mapped to one agent.

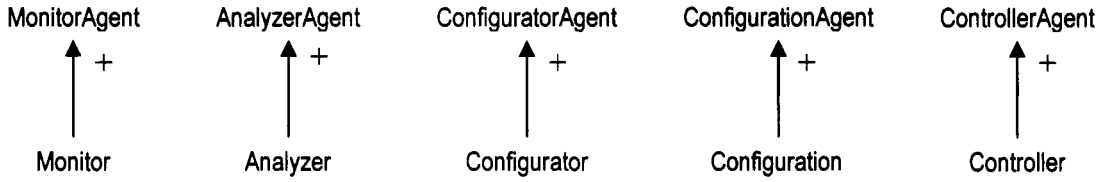

Figure 12. Agent model.

Table 20 through Table 24 depict the services model. The services model defines all services, inputs, outputs, pre-conditions, and post-conditions for each agent in the system. The services model contains the elements found in class and the sequence diagrams of traditional OOA & D.

Table 20. The Services Model for MonitorAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
accept value to monitor		confirmation	true	true
inform about change		*property, value*	*property* $\neq$ *nil* *value* < *threshold*	Analyzer informed *value* > *threshold*

Table 21. The Services Model for AnalyzerAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
accept notification	*property, value*	confirmation	*property* $\neq$ *nil* *value* > *threshold*	true
retrieve policy	*monitoredValue*	*policy*	policy available	policy retrieved
create configuration plan	*policy*	*configurationPlan*	*policy* $\neq$ *nil*	*configurationPlan* $\neq$ *nil*

Table 22. The Services Model for ConfiguratorAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
apply configuration step	*configurationStep*	*result*	*configurationStep* ≠ *nil and* ¬ *executing another configurationStep*	true

Table 23. The Services Model for ConfigurationAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
initiate configuration	*configurationPlan*	*configurationStep*	*configurationPlan* ≠ *nil*	*configurationStep* accepted

Table 24. The Services Model for ControllerAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
receive activity	*activity*	*activity validated*	*activity is valid* ^ *activity applicable*	true
execute activity	*activity, entity*	*confirmation*	*activity can be executed on entity*	true
inform	*activity result*	*confirmation*	*activity executed*	true

The final part of the design is an acquaintance model, which shows the communication pathways that exist between agents (Figure 13).

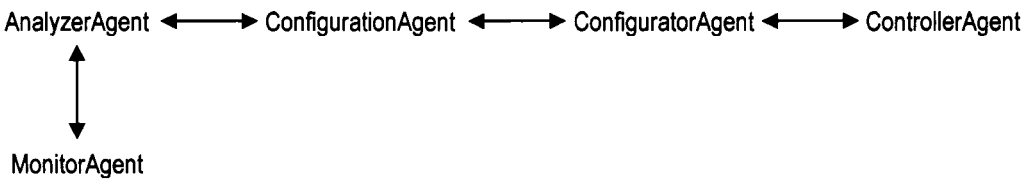

Figure 13. Acquaintance model.

7.4.8 Sequence of Events (Dynamic Model)

Figure 14 depicts the typical sequence of events, triggered by a change in one or more of observing parameters. Information passed by the Monitor indicates a need for reconfiguration. Based on the information obtained from the Monitor and the Policy, the Analyzer creates the ConfigurationPlan and initiates the process of the Configuration. The Configuration directs the process, assigning the task of the addition of the Component to the Configurator, responsible for configuration of the Entity, which executes the task and requests the Controller to restart the Entity. This sequence of events follows closely the execution of a typical Monitor-Analyze-Plan-Execute autonomic loop and involves all major modules comprising an autonomic element.

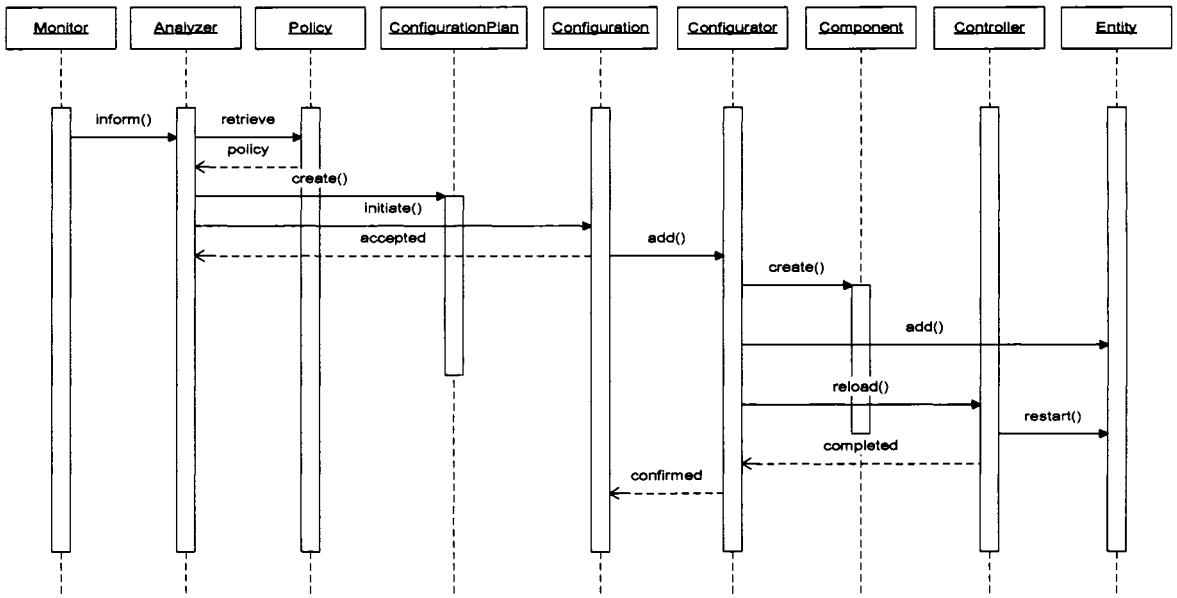

Figure 14. Sequence diagram for use case add new component.

Note that this sequence of events is a simplified one and that additional elements like component verification, version checking, regression testing, roll-back of changes,

and system integrity checking shall be included in a production grade system. Each use case can be expressed as a sequence of events. In a traditional OOA & D there is 1:1 mapping between use cases and sequence diagrams. In AOSE, each use case can be represented as a sequence of events/actions with messages passed between agents. Messages can be passed using ACL or KQML prerogatives. In addition to the design specified through the Gaia methodology, for each agent in the system, presence and a degree of the agent characteristics–if, and in which degree is an agent mobile, collaborative, autonomous, adaptable, persistent, or knowledgeable--can be further defined and shown on an agent properties axis (Griss & Pour, 2001).

7.4.9 Component Diagram

Figure 15 depicts the component diagram of the framework's software agents.

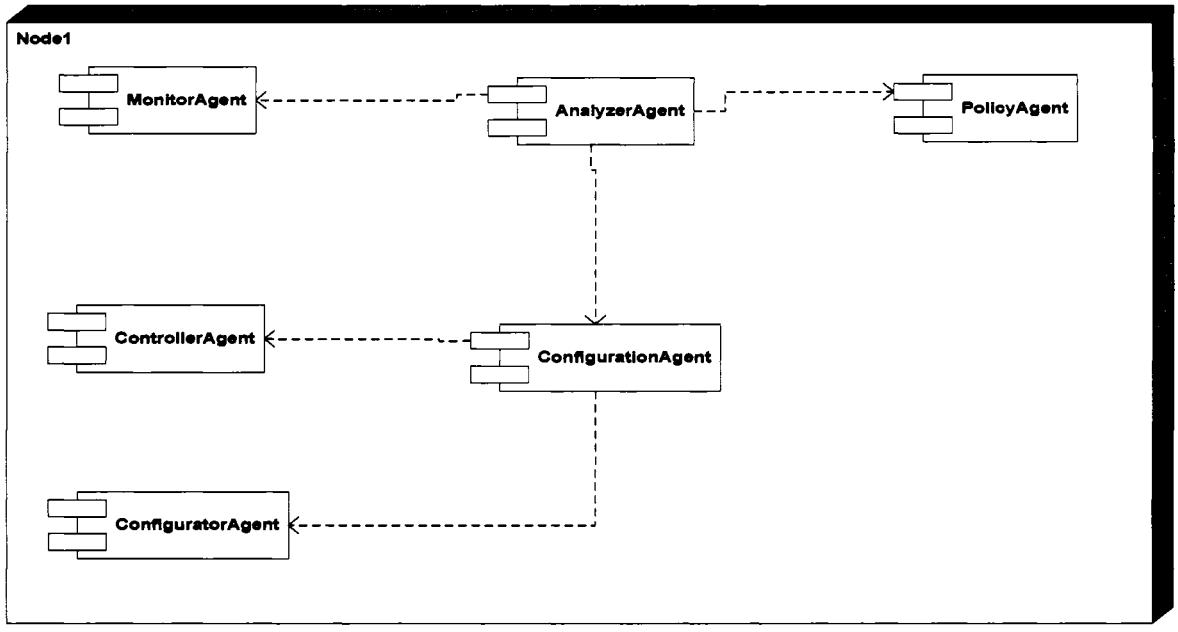

Figure 15. Component diagram.

7.4.10 System Design

To make the system design complete, additional systems, not part of the framework, but necessary for its use are introduced. Those systems provide services and capabilities required for storing and managing components, configurations and policies:

1. Web Services API (Application Server)
2. Repository (Application Server/Database Server)
3. User Interface for adding new configurations, components, and policies (Application Server)

Figure 16 depicts a deployment diagram of the framework, including these additional systems and components.

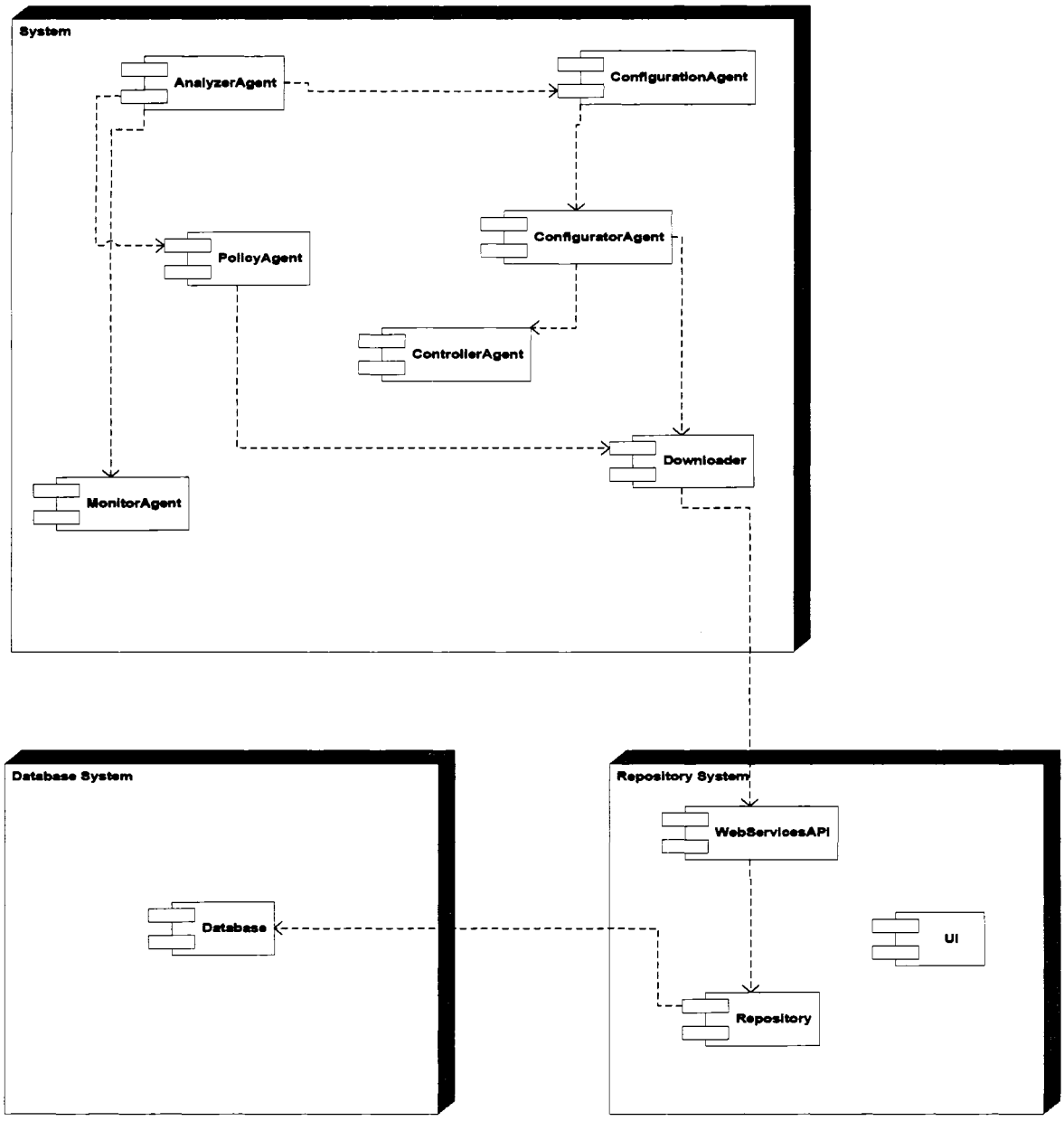

Figure 16. Deployment diagram.

The monitor agent is installed on the observed system. The monitor agent is responsible for monitoring the application and detecting significant events related to the application. "Normal" operation of the application is defined as a set of limits (thresholds) and behavior. Quantification of the normal operation is expressed as a set of rules and values

stored in a database or a configuration file. In the control theory world, the monitor agent plays a sensor role. The monitor agent is given a set of parameters to monitor. Monitoring can be implemented with pooling (measuring or running a specific task every T amount of time), and taking a sample of observed values (variables). These values are then compared with nominal values (or acceptable ranges). A corrective action is executed in the case of detected changes that are outside of predefined limits. A corrective action can consist of one or more tasks that should be executed in the sequence to affect the behavior of the environment or the system. These tasks will depend from application to application and will usually involve some kind of resource management or application reconfiguration.

The analysis consists in finding the root cause of the detected event. There could be multiple possible causes, which are then ranked by probability of occurrence. For example:

1. Verify that the service is up and running (YES/NO).
2. Verify that the application server can connect to the database server (YES/NO).
3. Verify that the service can perform a query in the database (YES/NO).
4. Investigate a type of the thrown exception.
5. Search a knowledge base to find the possible causes.

In the planning stage, an autonomic system must decide how and when to apply a corrective action. This stage involves reasoning about the system's interaction with its environment (e.g., availability of services that it provides to users). The decision when and how to apply a corrective action will depend on inputs from the analysis stage and

restrictions imposed on the execution of corrective actions (e.g., the system restart is required, but it can not be performed while the system performs certain operations or some resources are in use).

A corrective action consists of one or more tasks that have to be executed. These tasks usually transfer in one or more method invocations. A set of tasks that can be executed represent controls (effectors) to be applied on the system in order to change its behavior.

7.4.11 Implementation

An implementation of the framework using Java and JADE (Java Agent Development Framework) is proposed. The JADE framework conforms to FIPA standards and provides packages for ACL, ontology, and protocols. JADE also provides a directory (Yellow Pages) service and runs on top of a JVM. Possible implementation of a repository, based on web services, shall be configurable to run on either Tomcat or JBoss application server, and PostgreSQL as a DBMS. An implementation of a web services layer can be achieved through use of the Axis framework.

7.5 Challenges and Limitations

There are a number of challenges that must be overcome to support true dynamic reconfiguration. Some of the challenges related to dynamic reconfiguration are

1. Maintaining the integrity of the system
2. Synchronization
3. Keeping tab on states of all components
4. Security

The framework described in this work models the core configuration concept. There are a number of other related concepts and elements that would be necessary to include when creating a production grade application.

7.6 Chapter Last Remarks

The analysis and design of a framework for self-configuration based on software agents are described. A deployment of the framework and a support infrastructure is also defined.

CHAPTER: 8 Examples of Self-configuration Based on the Proposed Framework

8.1 Introduction

This chapter explores the design of a sample application which utilize self-configuration framework. Different aspects and elements of self-configuration are analyzed and defined.

8.2 Example 1: An Example of Self-configuring Application

8.2.1 Introduction

A company called Acme, Inc. is a software manufacturing company with two software products: data collection application (DCA) and data aggregation server (DAS). The DCA is running on a number of test or simulation machines (nodes) collecting data in the form of XML files. These files are being sent to a DAS through a Java application issuing calls to a DAS server via a web services API. There are numerous deployments of DASs and DCAs.

8.2.2 Problem Statement

Acme, Inc. is looking for a better way of updating configurations of their DAS and DCA deployments. The following needs of Acme, Inc. and its customers motivate the development of an automatic updater (deployment) tool:

1. Various customers would like to propagate changes in the DCA code to test or simulation stations without reinstalling everything manually. This can be a daunting task in case the customer has a large number of test stations.

2. Some of the customers run remote operations, with lack of skilled and authorized personnel to perform the setup.
3. There is a need for propagation of newest fixes and patches as they are available.
4. Uninterrupted (or minimal interruption) of testing or simulation.

The set of tasks based on these needs include documenting the requirements and proposing a design of the automatic update (deployment) toll of DAS and DCA. The process involves making a decision on a couple of design alternatives, and identification of the changes required in DAS and DCA systems.

8.2.3 Requirements

Automatic update (deployment) tool shall:

R01: Be transparent and induce minimal involvement of Acme's staff or customers.

R02: Allow uploading of new versions of DAC software on a DAS application server.

R03: Allow automatic verification of available updates.

R04: Perform all upgrades and uploads through a web services API. Since web services are running over HTTP, only the requirement for an upgrade would be allowed HTTP/HTTPS access.

R05: Implement an authentication and authorization procedure for upgrade.

8.2.4 Proposed Solution

A customer DAS server periodically and on demand synchronizes its content with the Acme, Inc. deployment server. Similarly, as a DCA verifies and synchronizes the version of software deployed on a test or simulation station, a DAS server can employ a

Synch service based on the designed framework. This service will on its startup or on a scheduled basis try to connect to Acme, Inc. site using a dedicated port, which can be placed as a part of a DAS configuration and exchange information about which version of DCA or DAS software is running on the server. In the case where a newer version is available, the Sync service would download the DCA (DAS) software from the Acme, Inc. site. In the case of the DAC software being downloaded, it will place it in a deployment repository. Note that in the case where the Acme, Inc. site is down or is not available, the Sync service proceeds with loading the DAC software and runs with the version it already has. In any case, no harm is done.

8.2.5 Benefits

Automatic updates decrease the burden on Acme, Inc. employees and on its customer too, and Acme, Inc. has full control over what and when should be made available for each customer. If Acme, Inc. wants to propagate changes, it can do it per each customer and with its discretion, by requiring the customer's DAS server Sync service to authenticate itself to the Acme, Inc. deployment service first.

8.2.6 DCA Updates

The following sequence of steps describes the basic functionality of the application:

1. The DCA Sync service loads the DCA configuration from the DCA configuration file (e.g., an XML configuration file).
2. The DCA configuration contains three attributes:

a) version - version of the DCA (e.g., version="3.2")

b) revision - revision of the DCA (e.g., revision="10")

c) update-service-url – a relative address of an update web service on the DAS

3. The DCA Sync service extracts following information from the configuration file:

   a) A DAS address (which may include a port number)

   b) A station GUID

   c) An update service URL

   d) A software version

   e) A software revision

4. The DCA Sync service connects to the DAS server update service (Web service API) and sends three parameters:

   a) Station GUID

   b) Version

   c) Revision

Note that in the case the DAS server is down or not available, the DCA Sync service proceeds with loading the DCA and runs with the version it already has.

5. The DAS update service processes parameters and:

   a) Invokes lookup in the repository

   b) If a newer version of the software is available, retrieve it from the deployment repository and send it back to the DCA Sync service.

   c) The DCA Sync service receives the update

   d) The DCA Sync service stops the DCA (if it is running)

e) The DCA Sync service installs the update

f) The DCA Sync service starts the DCA

g) The DCA Sync service confirms the update to the DAS update service

e) The DAS update service updates status in the repository

f) If the version and revision received are up-to-date notify the DCA Sync service that the station already has the latest release of the DCA software installed.

Changes required in DCA:

The DCA Sync synchronizer based on the proposed framework is required. Its main tasks are synchronize a station's DCA software with the version available on a DAS, update and start a DCA.

Changes in DAS:

A new, Update web service is required.

8.2.7 DAS Updates

The deployment server (DS) of Acme, Inc. is a service running on a dedicated machine which allows a customer of DAS servers to connect to and check for software upgrades and patches.

The service requires each DAS server connecting to it to authenticate and identify itself and then it sends following information: (a) Version, Revision and release date of the DAS software running on it; (b)Version, Revision and release date of the DCA software in the DAS server deployment repository.

Based on the DAS server identification, the deployment service looks up its customer database and checks if there is any new update for that customer in the

repository. The repository is divided per customer, since not all customers will run on the same level of the Acme, Inc. software.

8.2.8 DAS Upgrade Scenario

1. Acme, Inc. makes an upgrade of its software (either a new release, incremental upgrade, or a patch) and decides to make it available to its customer X.
2. The upgrade is placed in the deployment repository.
3. The customer database is updated for the customer to inform the deployment service that a new upgrade is available.
4. When a customer's DAS server Sync service connects to Acme, Inc.'s deployment server, it first sends its credentials. The credentials can be provided as a server GUID, a key, a password, or an electronic certificate.
5. The Acme, Inc.'s deployment server allows the customer's DAS server Sync service to proceed.
6. The customer's DAS server sends information about:

   a) The DCA software version, revision, release date and a list of patches and upgrades applied.

   b) The DAS software version, revision, release date, and a list of patches and upgrades applied.
7. The Acme, Inc.'s deployment service consults the customer's database and realizes that there is a new upgrade for the customer available and enabled.
8. A new upgrade is sent to the customer's DAS server Sync service through HTTP/SOAP mechanism.

9. The customer's DAS server Sync service installs the upgrade and notifies the Acme, Inc.'s deployment server about success or a failure.
10. In the case of success, the Acme, Inc.'s deployment server updates the customer's database with information that the new update was propagated. Information about what has been deployed and when with all significant timestamps is registered. This way Acme, Inc. always knows which customer is running what software and at which level or a version of the software.

Exception to flow:

3b. If an upgrade is manually installed, an Acme, Inc. employee needs to update the customer's database with the necessary information to keep it up-to-date. This would mostly occur with the very first installation of the software when the software is initially distributed to a customer on a release disk.

Changes in DCA

1. The DCA Sync service based on the framework is required. Its main tasks are to synchronize the station DCA software with a version available on the DAS, update and start the DCA.

Changes in DAS

1. A new, Update web service is required
2. A "Check for Updates" button needs to be added on the Admin page. This button would allow a DAS administrator to manually initiate a check for available updates. Since the DAS server would be running in this case, any updates would be downloaded only and installed on the next reboot. The DAS server administrator

would be informed about new updates availability, about their download, and when they would be installed. It is up to a DAS server administrator to decide whether she wants to reboot the application immediately or schedule a reboot action through a scheduler.

8.2.9 Authentication

Each DAS server must authenticate itself to the Acme, Inc.'s deployment service. Each DAS server can authenticate itself by either one or a combination of the following mechanisms:

1. Unique GUID set in the DAS server during the installation
2. A username or a password set in the DAS server
3. An electronic certificate

8.2.10 Deployment Scenario

Figure 17 shows a deployment of the solution in the described example.

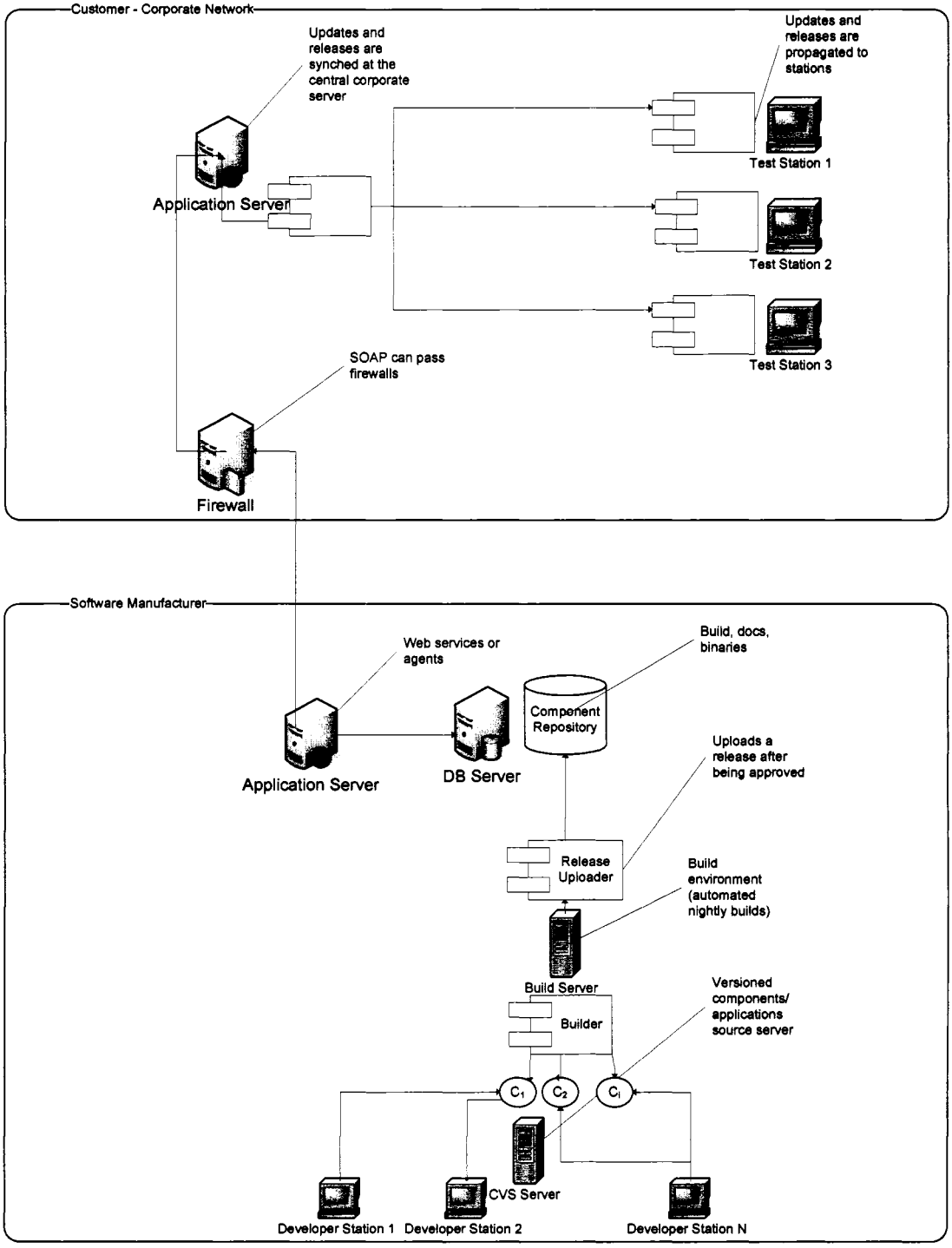

Figure 17. A deployment example.

8.3 Example 2: An Example of Self-Configuring Device

This example explores the relation of context awareness and reconfiguration. The example can be designed with a simulation of "virtual devices." The simulation should include the spatial aspect of an environment in which a device appears in the system as a newly discovered "node" being configured or reconfigured automatically. The device provides information about itself and an URL of a repository server for downloading drivers or configuration instructions.

When a device is being moved to a new environment, the framework recognizes changes through the Monitor, discovers services through the discovery mechanisms provided by Yellow pages of the agent platform, negotiates a protocol with Providers through the Negotiator, downloads necessary code via the Downloader to adapt the device to the environment, installs the code with the Configurator, and reloads the device configuration on the fly through the Controller, without interrupting the device's operation. The Configuration coordinates the whole process and the actions performed.

8.4 Chapter Last Remarks

In this chapter some of the possible examples and uses of the proposed framework are shown. A real-world problem with automatic propagation of software updates and utilization of the framework is described. The second example talks about automatic device configuration. In this hypothetical example, additional context-awareness functionality is required to be connected to the Monitor.

CHAPTER: 9 Conclusions

9.1 Introduction

This thesis explores aspects of autonomic computing (particularly aspect of self-configuration) and the role of autonomic computing in dealing with the increasing complexity and costs of IT infrastructure.

The first part of this work focuses on the current situation in the field of autonomic computing. The challenges and possible solutions are analyzed. Different aspects of autonomic computing are explored in detail as well.

The second part of the thesis focuses on the design of a framework for self-configuration, based on the software agent technology. Use of the framework in building a sample application is described.

The thesis concludes with proposals for future work in this area and possible enhancements of the proposed framework.

9.2 Thesis Contributions

Results of this research are a proposal of a systematic approach in the analysis and design of autonomic computing systems, a guideline for selecting appropriate architecture, and strategies to cope with problems and challenges. A basic framework for self-configuration is created, generic enough to be applied in various autonomic computing problems. The stable core of the framework tries to capture fundamental domain knowledge, and additional work is required to adapt the framework to a particular problem.

9.3 Future Work

Future work will focus on expansion of the proposed framework, refinements of the design methodology, and the model and development of a more comprehensive set of examples.

REFERENCES

Aiber, S., Gilat, D., Landau, A., Razinkov, N., Sela, A., & Wasserkrug, S. (2004). Autonomic self-optimization according to business objectives. In *Proceedings of the First International Conference on Autonomic Computing* (ICAC'04), (pp. 206-213). New York: ACM Press.

Amin M. (2000). Toward self-healing infrastructure systems, *IEEE Computer, 33*(8), 44-53.

Bantz D. F. et al. (2003). Autonomic personal computing. *IBM Systems Journal, 42*(1), 165-176.

Bennani, M.N., & Menasce, D.A. (2004). Assessing the robustness of self-managing computer systems under highly variable workloads. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 62-69). New York: ACM Press.

Blair G. S. et al. (2002). Reflection, self-awareness and self-healing in OpenORB. In *WOSS '02: Proceedings of the first workshop on self-healing systems*, (pp. 9-14). New York: ACM Press.

Candea, G., Kiciman, E., Zhang, S., Keyani, P., & Fox, A. (2003). JAGR: an autonomous self-recovering application server. In *Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS'03)*, (pp. 168-177).

Cofino, T., Doganata, Y., Drissi, Y., Tong F., Kozakov, L., & Laker, M. (2003). Towards knowledge management in autonomic systems. In *Proceedings of Eight IEEE International Symposium on Computers and Communication (ISCC'03)*, (pp. 789-794).New York: ACM Press.

Coleman, D. et al. (1994). *Object-oriented development: The fusion method.* Hemel Hempstead, England Prentice Hall International.

Engel, M., & Freisleben, B. (2005). Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the Fourth International Conference on Aspect Oriented Software Development,* (pp. 51-62). New York: ACM Press.

Fontana, J. (2004). The component balancer: Optimization of component-based applications. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 338-339). New York: ACM Press.

Georgiadis I., Magee J., & Kramer J. (2002). Self-organizing software architectures for distributed systems. In D. Garlan, J. Kramer, & A.L. Wolf (Eds.) *WOSS'02: Proceedings of the first workshop on self-healing systems* (pp. 33-38). New York: ACM Press.

Griss, M.L., & Pour. G. (2001). Accelerating development with agent components. *IEEE Computer, 34*(5), 37-43.

Jennings N.R. (2000). On agent-based software engineering. *Artificial Intelligence, 177*(2), 277-296.

Kephart J. O., & Chess D.M. (2003). The vision of autonomic computing. *IEE Computer, 36*(1), 41-50.

Larman, C. (2004). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.).New York: Prentice Hall.

Loeser, C., Ditze, M., Altenbernd, P., & Rammig, F. (2004). GRUSEL - a self optimizing, bandwidth aware video on demand P2P application. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (p330-331). New York: ACM Press.

Mahabhashyam, S.R., & Gautam, N. (2004). Dynamic resource allocation of shared data centers supporting multiclass requests. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 222-229). New York: ACM Press.

Martin, P., Powley, W., & Benoit, D. (2004). Using reflection to introduce self-tuning technology into DBMSs. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS'04)*, (pp. 429-438). New York: ACM Press.

McCann J .A. (2003). The database machine: Old story, new slant? *Proceedings of the first Biennial Conference on Innovative Data Systems Research, VLDB.*

Odell J., & Nodine M. (2003). *Foundation for intelligent physical agents* (Modeling Work Plan). Retrieved May 6, 2006, from http://www.fipa.org/docs/wps/f-wp-00022/f-wp-00022.html

Sadjadi, S.M., & McKinley, P.K. (2005). Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the Second IEEE International Conference on Autonomic Computing (ICAC'05)*, (pp. 76-87). New York: ACM Press.

Sadjadi, S.M., & McKinley, P.K. (2004). Transparent self-optimization in existing CORBA applications. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 88-95). New York: ACM Press.

Sadjadi, S.M., McKinley, & Cheng, B.H.C. (2005). Transparent shaping of existing software to support pervasive and autonomic computing. In *Proceedings of the International Conference on Software Engineering (ICSE) Workshop on Design and Evolution of Autonomic Application Software (DEAS)*, (pp. 1-7). New York: ACM Press.

Sadjadi, S.M., McKinley, P.K., Stirewalt, R.E.K., & Cheng, B.H.C. (2004). Generation of self-optimizing wireless network applications. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 310-311). New York: ACM Press.

Schneier B. (2000, March). Software complexity and security. *Crypto-Gram Newsletter*. Retrieved May 5, 2006, from http://www.counterpane.com/crypto-gram-0003.html 00022/f-wp-00022.html

Sterritt, R. (2003). Pulse monitoring: Extending the health-check for the autonomic grid. In *Proceedings of IEEE International Conference on Industrial Informatics (INDIN'03)*, (pp. 433-440). New York: ACM Press.

Sterritt, R., & Bustard, D. (2003). Towards an autonomic computing environment. In *Proceedings of 14th International Workshop on Database and Expert Systems Applications (DEXA'03)*, (pp. 694-698). New York: ACM Press.

Sterritt, R., & Chung, S. (2004). Personal autonomic computing self-healing tool. In *Proceedings of 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, (pp. 513-520). New York: ACM Press.

Sterritt, R., & Hinchey, M. (2005). Engineering ultimate self-protection in autonomic agents for space exploration missions. *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, (pp. 506-511). New York: ACM Press.

Stojanovic, L., Abecker, A., Stojanovic, N., & Studer, R. (2004). Ontology-based correlation engines. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 304-305). New York: ACM Press.

Tanenbaum, A. S. (2001). *Modern operating systems* (2nd ed.). Prentice Hall.

Tesauro, G. et al. (2004). A multi-agent systems approach to autonomic computing. In

Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), (pp. 464-471). New York: ACM Press.

Whisnant K., Kalbarczyk Z. T., & Iyer R. K. (2003). A system model for dynamically reconfigurable software. *IBM Systems Journal*, *42*(1), 45-49.

Wooldridge, M., Jennings, N. & Kinny, D. (2000). The Gaia Methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems 3*(3), pp. 285-312.

Yiyu C. Das, A., Gautam, N., Qian W., & Sivasubramaniam, A. (2004). Pricing and autonomic control of Web servers with time-varying request patterns. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (pp. 290-291). New York: ACM Press.

Zambonelli, F., Jennings N. R., & Wooldridge M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Trans. on Software Engineering and Methodology, 12*(3), 317-370.