Quality Criteria and an Analysis Framework for Self-Healing Systems

by

Sangeeta Neti

B.E., Pt. Ravishankar Shukla University, India, 1998

A Thesis Submitted in Partial Fulfilment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Quality Criteria and an Analysis Framework for Self-Healing Systems

by

Sangeeta Neti

B.E., Pt. Ravishankar Shukla University, India 1998

Supervisory Committee

Dr. Hausi A. Müller, (Department of Computer Science)

Supervisor

Dr. Yvonne Coady, (Department of Computer Science)

Departmental Member

Dr. Alex Thomo, (Department of Computer Science)

Departmental Member

Supervisory Committee

Dr. Hausi A. Müller, (Department of Computer Science)

Supervisor

Dr. Yvonne Coady, (Department of Computer Science)

Departmental Member

Dr. Alex Thomo, (Department of Computer Science)

Departmental Member

ABSTRACT

Autonomic computing has become more prevalent and, hence, its evaluation is becoming more important. In this thesis, we address the issue of evaluating the software architecture of self-healing applications with respect to the changes and adaptation over long periods of time. To facilitate this evaluation, we developed an analysis and reasoning framework for the architecture of self-healing systems. The reasoning framework is based on attribute-based architectural styles (ABASs) and is tailored to selected quality attributes. When an autonomic system evolves, the proposed reasoning framework can be used to re-analyze the system and verify certain quality attributes. The explicitly available relationship between architecture and quality attributes not only helps in documenting the current architecture design, but also allows developers to reuse the architectural analysis during long-term

evolution when the original system designers are long gone. Hence, the proposed framework can facilitate both design and maintenance of self-healing systems.

In order to develop the analysis and reasoning framework, we identified key quality attributes for self-healing systems. We have also defined new autonomic-specific quality attributes for the self-healing systems, which includes support for detecting anomalous system behaviour, support for failure diagnosis, support for simulation of expected behaviour, support for differencing between expected and actual behaviour, and support for testing of correct behaviour. Further, we customized the ISO 9126 quality model to the quality requirements of self-healing systems, considering both traditional attributes as well as newly defined autonomic-specific attributes.

Table of Contents

Supervisory Committee ii
Abstract iii
Table of Contents v
List of Tables viii
List of Figures viii
Acknowledgements ix

1 Introduction 1
1.1 Introduction to Autonomic Systems..... 1
1.1.1 Characteristics of Autonomic Computing Systems..... 1
1.1.2 Characteristics of Self-Healing Systems..... 3
1.2 Analysis and Reasoning Frameworks for Computing Systems..... 6
1.3 Motivation and Objectives..... 7
1.4 Thesis Outline..... 9
2 Quality Criteria for Self-Healing Systems 11
2.1 Introduction..... 11
2.2 Software Quality Attributes for Self-Healing Systems..... 12
2.2.1 Traditional Quality Attributes..... 12
2.2.2 Autonomic-Specific Quality Attributes..... 14
2.3 Quality Model for Self-Healing Systems..... 16
2.4 Summary..... 20
3 Architectural Styles for Self-Healing Systems 21
3.1 Introduction..... 21
3.2 Architecture of Self-Healing Systems..... 21
3.3 Mapping of the Conceptual Model of a Self-Healing Architecture to

an Autonomic Element..... 22
3.4 Different Architectural Styles for Self-Healing Systems..... 24
3.5 Summary..... 28
4 Attribute Based Architectural Styles for Self-Healing Systems with Respect to Traditional Qualities 29
4.1 Introduction..... 29
4.2 Attribute Based Architectural Styles (ABASs)..... 29
4.3 Traditional Quality ABASs..... 32
4.3.1 Modifiability ABASs..... 32
4.3.1.1 Peer-to-Peer Modifiability ABAS..... 33
4.3.1.2 Aggregator-Escalator-Peer Modifiability ABAS..... 35
4.3.1.3 Chain-of-Configurators Modifiability ABAS..... 37
4.4 Reliability ABAS..... 39
4.5 Summary..... 43
5 Attribute Based Architectural Styles for Self-Healing Systems with Respect to Autonomic-Specific Qualities 44
5.1 Introduction..... 44
5.2 Autonomic-Specific Quality ABASs..... 44
5.2.1 Support for Detecting Anomalous System Behaviour ABASs..... 45
5.2.1.1 Peer-to-Peer Support for Detecting Anomalous System Behaviour ABAS..... 45
5.2.1.2 Aggregator-Escalator-Peer Support for Detecting Anomalous System Behaviour Sub ABAS..... 50
5.2.1.3 Chain-of-Configurators Support for Detecting Anomalous System Behaviour ABAS..... 52
5.2.2 Support for Simulation of Expected or Predicted Behaviour ABAS..... 53

5.2.3 Support for Differencing between Expected and Actual Behaviour ABAS..... 57
5.2.4 Support for Failure Diagnosis ABAS..... 58
5.2.5 Support for Testing of Correct Behaviour ABAS..... 61
5.3 Summary..... 61
6 Analysis of Self-Healing Systems using Attribute Based Architectural Styles 62
6.1 Introduction..... 62
6.2 Using ABASs in the Analysis of Self-Healing Systems..... 62
6.3 Case Study: Model for a Self-Managing Java Server..... 63
6.3.1 Analysis..... 65
6.3.2 Results of the Analysis..... 67
6.4 Summary..... 70
7 Conclusions and Future Work 72
7.1 Conclusions..... 72
7.2 Contributions..... 73
7.3 Future Work..... 73
Bibliography 75

List of Tables

Table 2.1. Quality Characteristics According to IS0 9126..... 18

List of Figures

Figure 2.1. Relationships between autonomic characteristics and quality factors..... 13

Figure 2.2. Quality model for self-healing systems based on ISO 9126..... 19

Figure 3.1. Conceptual architectural model of a self-healing system..... 23

Figure 3.2. Conceptual model of a self-healing architecture mapped to an autonomic element..... 24

Figure 3.3. Aspect peer-to-peer architectural style..... 25

Figure 3.4. Aggregator-escalator-peer architectural style..... 26

Figure 3.5. Chain-of-configurators architectural style..... 28

Figure 4.1. Characterization of the modifiability ABAS..... 32

Figure 4.2. Reliability architectural style..... 41

Figure 4.3. Self-healing reliability model..... 42

Figure 5.1. Characterization of the support for detecting anomalous system behaviour ABAS..... 46

Figure 6.1. Message flow in the Java server model..... 65

Acknowledgments

I am deeply indebted to my supervisor, Dr. Hausi Müller, for his valuable time, guidance, encouragement, and patience throughout my research. In the first place, I am thankful to him for giving me an opportunity to work under his supervision at the University of Victoria. I would also like to thank him for providing financial assistance throughout my program. He always welcomed my thoughts and responded to them with great enthusiasm and encouragement. Thanks for providing such a friendly and healthy environment. I enjoyed being a part of this research group where the environment was friendly, flexible, encouraging, and challenging.

I would like to thank the other members of my supervisory committee, Dr. Yvonne Coady and Dr. Alex Thomo, for their suggestions and encouragement.

I would also like to acknowledge the support I received through teaching assistantships from the Department of Computer Science at the University of Victoria. I am also grateful to all my fellow graduate students of the Rigi group for their help and support. In particular, I would like to acknowledge the help I received from Holger Kienle who provided me with valuable comments on the thesis. Further, I would like to thank all my friends and well wishers for their encouragement over the years.

My special thanks go to my husband, Prabhakar, for his love, caring support, patience and constant encouragement without which the completion of this thesis would not have been possible. He instilled a lot of confidence in me and always encouraged me to achieve high goals. He always showed me good directions not only during the program but also in my life. This made my life much easier particularly in this new country. Thanks for your love and caring support. I would also like to thank my lovely daughter, Medha, for bringing love and joy to our entire family. Without

her co-operation, the timely completion of this thesis would not have been possible. I am so thankful to her for giving me a new meaning in life.

I am deeply indebted to my affectionate parents, Sri Gorty Hanuman Narayana and Smt. Gorty Ramalakshmi, whose blessings and unending love gave me all the strength to lead my life. They took all the pain and did their best to make my life comfortable. Thanks for your unending love, support, patience, and understanding. I would also like to express my special gratitude towards my elder sister, Lakshmi, who influenced me during every phase of my life. Without her help and support, I would not have come this far. Thank you for your unending love and support. Thanks to my loving sister, Kavita, who has always been a good friend to me. She always showed special concern to me and stood by me when I was in trouble. Thanks for believing and caring so much. Many thanks go to my brothers-in-law, Shyam Sunder Murthy and Vishwanathan, for their care and support towards me and my family. Finally, I would like to thank all my other family members for their love and affection.

Dedication

This thesis is dedicated to my loving husband without whose caring support this thesis would not have been possible, my beloved parents and my wonderful daughter.

Chapter 1

Introduction

1.1 Introduction to Autonomic Systems

Autonomic computing is a solution proposed by IBM in 2001 to alleviate the complexity of software systems which is one of the most important challenges faced by the IT industry today [1, 2, 3, 6]. As computing systems become more and more complex, the resources needed to manage and administer them, steadily increases. Out of these resources, the cost of human resources devoted to administration constitutes a major fraction of the IT costs. Autonomic computing aims to minimize these costs by making computer systems self-manageable.

1.1.1 Characteristics of Autonomic Computing Systems

Self-managing systems are grouped according to the following four basic properties [1-6]:

Self-Configuration: An autonomic system needs to be able to configure and reconfigure itself under varying and unpredictable conditions. User-based re-configuration and automatic re-configuration, based on monitoring and feedback loops, should be allowed which means the involvement of the end-user needs to be varied. Systems ought to be designed to provide configurability with feature

capabilities such as separation of concerns, levels of indirection, integration mechanisms, scripting layers, plug-and-play, or set-up wizards, etc.

Self-Optimization: The system needs to be able to continually monitor and tune its resources and operations. More generally, to meet the ever-changing needs of the application environment, the system can continually seek to optimize its operations with respect to a set of prioritized non-functional requirements. Capabilities such as re-partitioning, re-clustering, load balancing, or re-routing is incorporated into the system to provide self-optimization.

Self-Healing: Without loss of data or noticeable delays in processing, the system needs to be able to recover from the routine and extraordinary events that might cause some of its parts to malfunction. Self-recovery means that the system can select, possibly with user input, an alternative configuration from the one of its currently used configurations with minimal loss of information or delay.

Self-Protection: The system needs to be capable of protecting itself by detecting and counteracting threats through the use of pattern recognition and other techniques. This means that the design of the system includes an analysis of the vulnerabilities and the inclusion of protective mechanisms that might be employed when a threat is detected. The design must provide capabilities to recognize and handle different kinds of threats easily, thereby reducing the burden of administrators.

1.1.2 Characteristics of Self-Healing Systems

This thesis focuses on the self-healing property of an autonomic system. There are many concepts, approaches and architectural designs for self healing systems [13-22]. Following is a brief description of self-healing systems.

Software Architecture for Self-Healing Systems

The research activities in architecture of autonomic computing systems can be categorized into four areas: monitoring of components, interpretation of monitored data, creation of a repair plan, and execution of a repair plan [1-10]. Based on this, McCann *et al.* identified two approaches to autonomic computing systems: tightly coupled and decoupled systems [11]. Following is a brief description of the two approaches:

Tightly Coupled Autonomic Systems: Tightly coupled autonomic systems are often built using intelligent agents with their own goals. In some systems the autonomic logic is tightly embedded in the main application logic of the agent. In a multi-agent system each component exhibits its own autonomic behavior. However, there is a clean separation between the conventional component that performs the task and the autonomic manager which implements self-management around it. Compared to the decoupled approach, multi-agent systems have the advantage of a distributed architecture which lowers the number of central points of failure. But there are also drawbacks to this architecture. For example, a chain reaction of agents instructing

other agents to change behavior can potentially lead to instabilities of the overall system.

Decoupled Autonomic Systems: A decoupled autonomic system is one in which a separate infrastructure handles the autonomic behavior of the system. Individual components are not autonomic. The adaptivity infrastructure is clearly separated from the running system. This infrastructure uses an architecture description model of the running system to monitor the running system, to reason about it, and to determine appropriate adaptive actions. The advantage of complete separation between autonomic behavior and the running system is that software adaptation can be plugged into a pre-existing system.

Traditional mechanisms that allow a system to detect and recover from errors are wired into applications at the code level. But they suffer from the problem that they are hard to change, reuse, or analyze. Externalized adaptation is used to overcome these problems [14, 15]. The centerpiece of this approach is the use of architectural models, which are used as a basis for run-time monitoring, error detection, and repair. Many researchers have explored this approach in which architectural models are maintained at run time and used as a basis for system reconfiguration and repair [17]. An architectural model is represented as a graph of interacting components [14, 15]. The nodes of a graph are called components and represent the principal computational elements or data stores of the system: clients, servers, databases, etc. Arcs are termed connectors and represent the interaction paths between components. To account for

various behavioral properties of a system, the components and connectors in the graph can be annotated with property lists. A monitoring infrastructure, consisting of probes, gauges, etc., is needed for decoupled autonomic systems. When a property in the architecture model is updated through monitoring, the architecture model is analyzed to verify its performance. If there is a change in the performance, a repair plan is created based on repair strategies that are defined in advance. An architectural style defines a set of formal constraints over architecture. When a system conforms to a style, it has many benefits including support for analysis, reuse, code generation, and system evolution [15].

A lot of research in creating new design techniques for autonomic systems is still going on. Kaminski *et al.* propose a solution called Chain of Adapters, which is a design technique to achieve backward compatibility [39]. The technique is particularly suitable for self-managed systems since it makes many version related reconfiguration tasks safe, and thus subject to automation.

Hawthorne and Perry discuss the following design issues for self-healing systems [18]:

Tight vs. Loose Coupling: This major design issue is the degree of inter-component coupling.

Tightly coupled systems: In this system, the reflection, reasoning, and configuration components have direct explicit interdependencies. These systems are simpler to

implement, but at the cost of flexibility and scalability. It is a risky architectural style because certain types of tight couplings can make the self-healing mechanisms vulnerable to being disabled when system components fail.

Loosely coupled systems: Loosely coupled adaptation frameworks are far more practical for large, complex systems [18].

Instance vs. Intent-based Selection: Another aspect of a self-healing architecture is the specification of the adaptation system or its components. This is important when one component needs to request a service from another component. Specifying a particular implementation of a monitor or configurator leads to logical dependencies or logical tight coupling among the entities. The adaptation mechanisms have to be designed such that the components identify one another by their logical or functional role in the system.

Peer-to-Peer vs. Hierarchical Organization: Peer-to-peer styles are simple. A peer-to-peer approach allocates the self adaptive functionality to symmetrical sets of monitor and configurator components. Each monitor interacts with a peer level configurator. Whereas hierarchical approaches allow monitor messages to reach higher level configurator or configurator manager components. This enables higher level components to make more comprehensive reconfiguration decisions if needed.

1.2 Analysis and Reasoning Frameworks for Computing Systems

Software architecture is one of the important approaches for designing and understanding a software system [24]. Software architecture is the key design artifact that represents all types of requirements to be achieved. Analysis of the software architecture is necessary to understand the implications of a design decision. The analysis helps in early detection of errors. It has been found that the design decisions at the architecture level can affect the qualities of a computing system [26]. To overcome this problem, the architectural styles have been linked to quality attributes. This link is established through analysis techniques that predict the affect of architectural design decisions on the achievement of quality. Hence, it becomes necessary to evaluate the architectures, to determine its fitness over long periods of time, with respect to certain qualities.

Attribute based architectural style (ABAS) is directly related to the evaluation of architectural styles. ABASs provide a basis for reasoning about software architecture's ability to meet its quality attributes [23]. The reasoning is accomplished by explicitly associating an analysis and reasoning framework with an architectural style. ABASs also help in the design of architectures. Further, ABASs help in evolving software systems. The analysis helps in quickly finding architectural risks and tradeoffs associated with a system. A collection of ABASs has been created for

computing systems [23, 38]. These ABASs have been successfully used in architecture evaluations and design exercises with large, complex industrial systems.

1.3 Motivation and Objectives

The brief literature survey revealed the importance of developing an analysis and reasoning framework for computing systems. A similar kind of analysis and reasoning framework is also necessary for autonomic systems to facilitate the evaluation of their architecture.

Autonomic computing has become more prevalent and, hence, its evaluation is becoming important. There are two issues that need attention: 1) the evaluation of an autonomic system and 2) assessment of the autonomicity of an I/T environment [7]. McCann *et al.* have listed some metrics for evaluating and comparing autonomic systems. Further, IBM has created an analysis tool to determine the level of autonomicity of an I/T environment. So far there is little literature on architecture evaluation for autonomic computing applications [25]. An autonomic application operates in a dynamic environment where autonomic elements interact and collaborate with each other [1, 2]. This kind of complexity and dynamicity requires understanding of the evolving software architecture for self-managing systems. Designing a flexible and scalable architecture is essential to the success of autonomic systems.

The goal of this work is to evaluate and document the software architecture of self-managing applications with respect to specific quality attributes. In this context, the evaluation of architecture means to identify the key attributes and to verify these attributes with respect to the changes and adaptation over long periods of time. The technique used in this work for evaluating the architecture is based on specific individual quality attributes.

This thesis focuses on the evaluation of self-healing systems as opposed to self-configuring, self-optimizing, or self-protecting systems. Self-healing systems need to adapt and change over time [13]. Such changes can be due to a change in their designed operating mode, accumulated component and resource faults, adaptations to external environments, component evolution, or changes in system usage. Hence, it is necessary to evaluate the architecture for self-healing applications in the context of software evolution over long periods of time. To facilitate this evaluation, we developed an analysis and reasoning framework for the architecture of self-healing systems.

We used the attribute-based architectural style (ABAS), formulated by Klein *et al.* [23], for this evaluation. ABASs are well suited for the analysis and design. The purpose of this work is to move the notion of architectural styles towards reasoning, either quantitative or qualitative, based on quality attribute specific models.

1.4 Thesis Outline

Chapter 2 discusses quality criteria for self-healing systems. Defining quality criteria is necessary not only for the evaluation of existing self-healing system architectures, but also for the design and implementation of new self-healing systems. Chapter 3 discusses existing architectural styles for self-healing systems. Software architectures provide high-level abstractions for representing the structure, behaviour and key properties of a software system. This chapter forms a foundation for the subsequent chapters.

In Chapter 4, we discuss the evaluation approach and define an analysis and reasoning framework for self-healing systems with respect to traditional qualities. Chapter 5 discusses the analysis and reasoning framework for self-healing systems with respect to autonomic-specific qualities.

In Chapter 6, we apply the analysis and reasoning framework, defined in Chapter 4 and 5, to evaluate a real world system. Conclusions and future work are presented in Chapter 7.

Chapter 2

Quality Criteria for Self-Healing Systems

2.1. Introduction

This chapter focuses on the quality criteria that can affect the architecture of a software system. Defining quality criteria is important for the evaluation of any software system architecture [27, 28]. In this thesis, we identify the key quality attributes for self-healing systems. An investigation is also carried out on characterization of these attributes. Further, an analysis is presented on the key qualities that affect the architecture with respect to changes and adaptation over long periods of time. Quality attributes include not only traditional quality criteria, such as reliability, modifiability, availability, but also autonomicity-specific criteria, such as dynamic adaptation support, dynamic upgrade support, diagnostics support, support for detecting anomalous system behavior, or support for accountability. New autonomic-specific quality attributes for self-healing systems are also defined. These criteria can be used for the evaluation of existing self-healing systems as well as for the design and implementation of new self-healing systems.

2.2. Software Quality Attributes for Self-Healing Systems

"Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability or interoperability)." [IEEE 1061]

Computing systems are used in many critical applications where a failure can have serious consequences. Developing systematic methods to relate the software quality attributes of a system to the system's architecture provides a sound basis for making objective decisions about design trade-offs. This enables engineers to make reasonably accurate predictions about a system's attributes that are free from any assumptions. The ultimate goal is the ability to quantitatively evaluate and trade off multiple software quality attributes to arrive at a better overall system. In this work, quality attributes for self-healing systems are grouped into two categories: traditional and autonomicity-specific quality attributes.

2.2.1. Traditional Quality Attributes

Self-healing is the capability to discover, diagnose, and react to disruptions. The main objective of adding self-healing features to any system is to maximize availability, survivability, maintainability, and reliability of the system [2]. Salehie and Tahvildari describe major and minor characteristics of autonomic systems and their relationship to quality factors as shown in Figure 2.1 [7]. Reliability and maintainability are two traditional quality attributes that are key properties in any self-healing system.

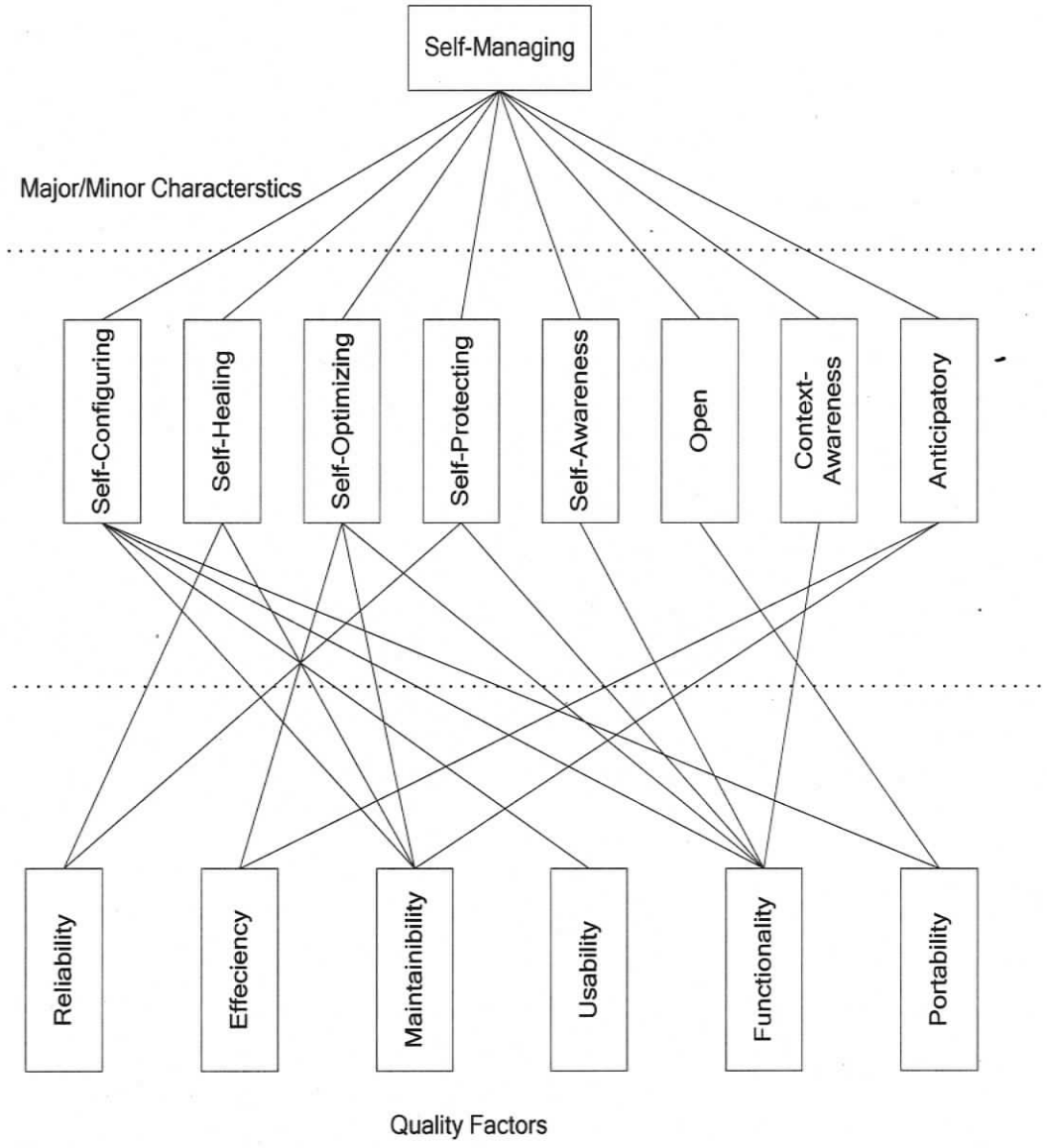

Figure 2.1. Relationships between Autonomic Characteristics and Quality Factors

Reliability in self-healing systems can be further subdivided into fault tolerance and robustness [28]. This means that self-healing systems should be fault-tolerant; delivering specified services despite the presence of faults [30] and robust enough to recover from any problematic situations.

Maintainability can be further subdivided into modifiability and extensibility. Designing a flexible and scalable architecture is essential for modifying self-managing systems without breaking them. Modifiability is an important attribute for self-healing systems since a highly modifiable architecture enhances a highly maintainable final software system [27].

2.2.2. Autonomic-Specific Quality Attributes

Serious problems can arise from the failure of a component in a computing system. Self-healing systems have to predict and take action to prevent these failures from having detrimental impact on applications [1]. The challenge in achieving this objective is to determine the cause of failure of the elements, preferably without shutting down and/or restarting the entire system. Detection of failures is particularly important when autonomic elements or applications are involved in interactions with multiple elements since the system will have accumulated significant state information over time. These failures can be predicted by characterizing and detecting anomalous behavior of the system. Finally, an appropriate recovery technique can bring the system back to its normal state of operation.

Based on the above requirements, the following qualities are identified as important autonomic-specific quality attributes for any self-healing system:

Support for detecting anomalous system behavior—This quality can be defined as the ability to monitor, recognize, and address anomalies with respect to performance of a subject system or its environment. This criterion can be further subdivided into *awareness* and *observability*. Awareness means that the system should provide support to monitor its performance (e.g., state information, behavior, correctness, or reliability) and recognize anomalies. Observability means that the system should provide support to monitor the execution environment of the subject system. Coupling is also considered as a characteristic since it affects complexity and hence the rate and time of anomaly detection. The quality, "support for detecting anomalous system behaviour", helps in early detection of problems and, hence, minimizes the cost involved with disruption of business transactions and applications.

Support for failure diagnosis—This quality can be defined as the ability to locate the source of misbehavior or failure such as component failure, system degradation, change in system usage, etc. This quality is dependent on the complexity of the problem, speed and accuracy of the diagnosis, robustness of the system to faults, etc.

Support for simulation of expected or predicted behavior—This can be defined as the ability to model the system accurately and thereby obtain the expected behavior of the running system. This quality in turn depends on the awareness of the system as

well as the correctness, completeness, consistency, and complexity of the model involved.

Support for differencing between expected and actual behavior—This can be defined as the ability to detect the difference between the current and expected behavior of the system. Simulation serves as a foundation to self-awareness through which differences between predicted and actual behavior can be investigated [34]. So this quality depends on the quality "support for simulation of expected or predicted behaviour."

Support for testing of correct behavior—This can be defined as the ability to test and verify that autonomic elements behave correctly. This quality is important and challenging to achieve since it is harder to predict the behavior of an autonomic system especially when it extends across multiple domains [1]. This quality can be categorized further into testability.

2.3. Quality Model for Self-Healing Systems

Quality modeling is a useful tool for requirements engineering as well as for evaluation, since it explicitly specifies the qualities of the system [28]. Evaluating the quality of software is very important not only from the perspective of a software engineer—for example, to determine the level of provided quality—but also from a business point of view, such as when to make a choice between two similar but

competing products. Several views on quality expressed by quality models have been defined. All these models are based on the idea that a software quality can be decomposed into a number of "ilities"—high level factors, such as usability and maintainability, which can be further decomposed into a number of sub-factors such as accessibility and testability. These sub factors can be measured by quality metrics e.g., directly measurable attributes such as "number of errors" or "degree of testing."

ISO 9126, proposes a generic model to specify and evaluate the quality of a software product from different perspectives such as acquisition, development, or maintenance. As shown in Table 2.1, ISO 9126 defines six characteristics that can be subdivided into the sub-characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. Attributes in the ISO context are the measurable elements of the high level quality characteristics and sub characteristics. The generic ISO 9126 model has to be customized according to the system's non-functional requirements. Figure 2.2 shows the ISO model adapted to the quality requirements of self-healing systems, considering both traditional as well as autonomic-specific attributes. Such a quality model helps with reasoning about the quality characteristics of a self-healing system. Complete analysis of the quality model allows to reason quantitatively about the desirable quality characteristics in the final software system [27]. This quality model can be further extended based on new requirements.

Table 2.1. Quality Characteristics According to IS0 9126

Characteristic	Description
Functionality	A set of attributes that bear on the existence of a set of functions and their special properties
Reliability	A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time
Usability	A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users
Efficiency	A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions
Maintainability	A set of attributes that bear on the effort needed to make specific modifications.
Portability	A set of attributes that bear on the ability of software to be transferred from one environment to another

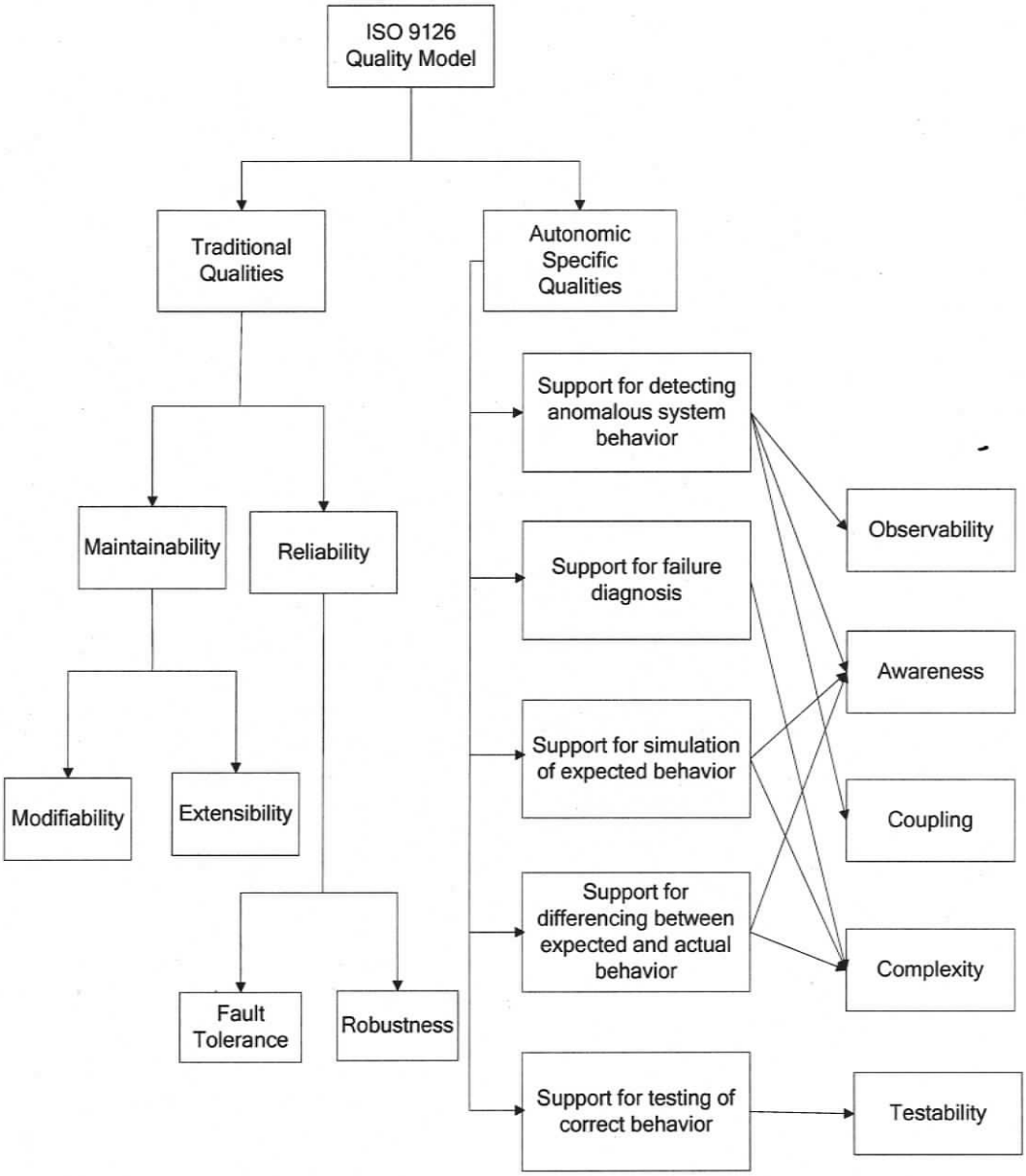

Figure 2.2. Quality Model for Self-Healing Systems Based on ISO 9126

2.4. Summary

It is essential to define the quality criteria for a software system since these criteria are important for the evaluation of any software system. In this work, a set of quality criteria for self-healing systems have been defined that can be further subdivided into two groups: traditional qualities and autonomic-specific qualities. These criteria can be used for evaluating the architectures of existing self-healing systems as well as for the design and implementation of new self-healing systems.

Chapter 3
Architectural Styles for Self-Healing Systems

3.1. Introduction

In this chapter, we analyze various architectural styles for self-healing systems. Software architectures provide high-level abstractions that represent the structure, behaviour and key properties of a software system [19]. These abstractions consist of several components and interactions among various components that build the system. An architectural style defines a set of formal constraints for the architecture. Architectural styles influence architectural evolution by restricting the possible changes an architect can make. Hence, choosing an appropriate architectural style is central to the system's design and evolution and ultimately its success.

In the subsequent chapters, existing architectural styles for self-healing systems are evaluated based on our quality attributes specific model.

3.2. Architecture of Self-Healing Systems

Hawthorne and Perry state the following three basic requirements for architectures of self-healing systems [18]:

- A reflection mechanism to detect internal or external conditions to which the system should respond to.
- A reasoning mechanism to determine what actions to be performed in response to input from the reflection mechanism.
- A configuration mechanism to perform the necessary changes to repair or optimize the system as directed by the reasoning mechanism.

Figure 3.1 depicts the conceptual model of a self-healing architecture. The adaptation mechanism consists of components that perform reflection, reasoning, and configuration of the system. Each component either monitors or configures one or more aspects of the system or its environment. In this model, runtime reflection components and configuration components are called monitors and configurators, respectively. Separate reasoning components are called configuration managers. However, the reasoning functionality can be encapsulated in the configurators themselves.

3.3. Mapping of the Conceptual Model of a Self-Healing Architecture to an Autonomic Element

Autonomic elements are the building blocks of an autonomic computing architecture [1]. An autonomic element consists of an autonomic manager and a set of managed elements such as components or resources. An autonomic manager monitors and

controls the managed elements. At the core, there is a control loop consisting of four components such as monitor, analyzer, planner, and executor. To understand the architecture of a self-healing system in terms of an autonomic element, we have mapped the conceptual model of a self-healing architecture as depicted in Figure 3.1 to an autonomic element as shown in Figure 3.2. In this case, the managed element constitutes a system component or the system's environment and the adaptation mechanism represents the autonomic manager. The reflection component corresponds to a monitor whereas a configuration component corresponds to an executor. Here the analyzer in combination with the planner corresponds to a reasoning component.

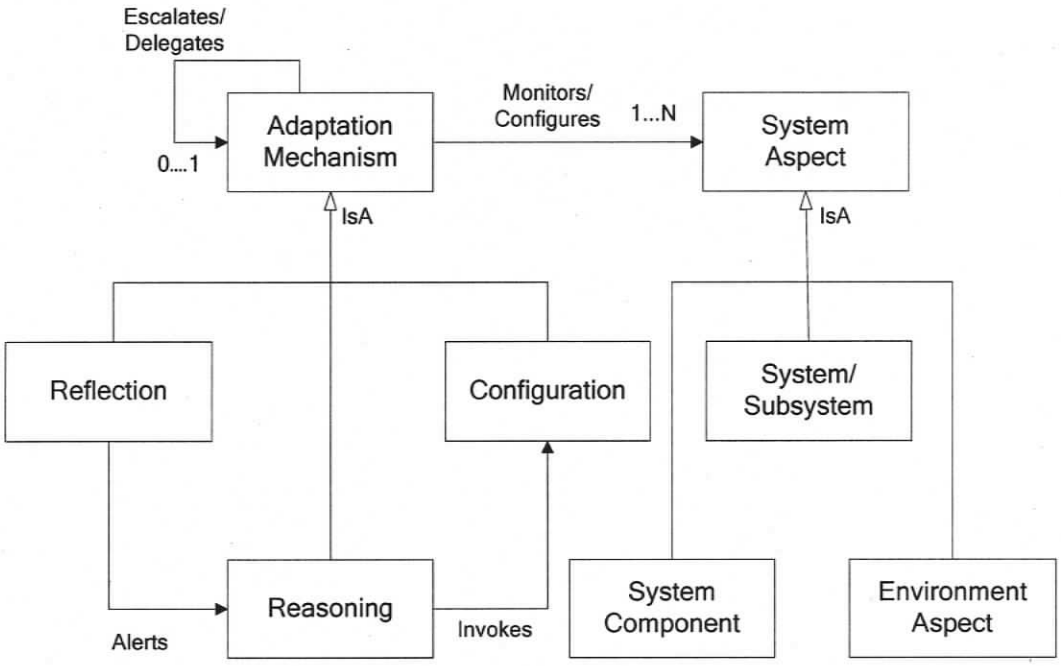

Figure 3.1. Conceptual architectural model of a self-healing system

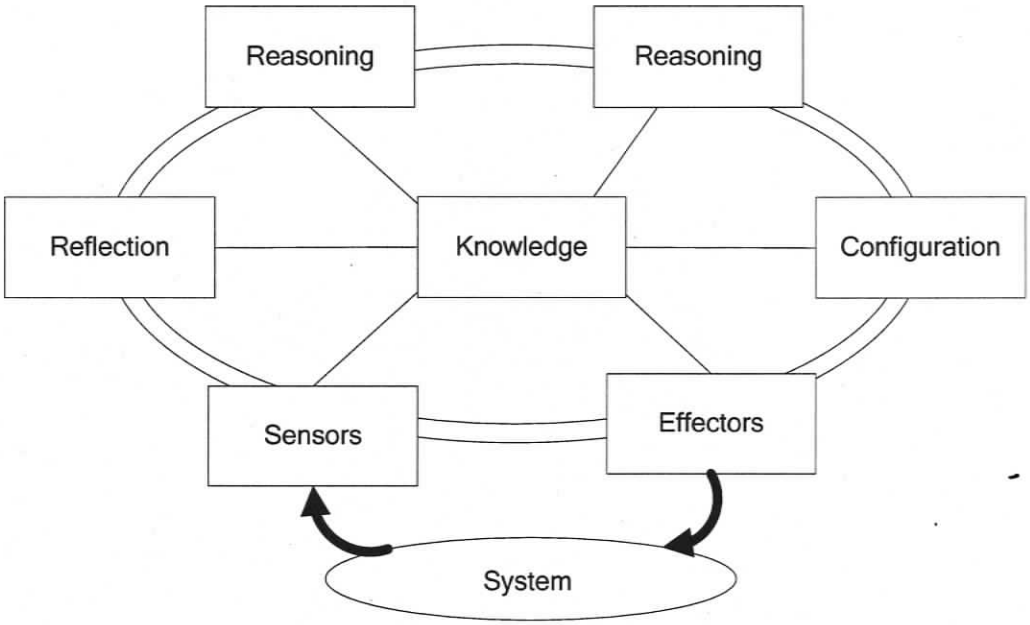

Figure 3.2. Conceptual model of a self-healing architecture mapped to an autonomic element

3.4. Different Architectural Styles for Self-Healing Systems

Hawthorne and Perry have presented and discussed the following architectural styles for self-healing systems [18]. For each style an architecture diagram that describes its working is included along with its advantages and disadvantages. In addition, we discuss for each architectural style a similar view in terms of the notion an autonomic element.

Aspect Peer-to-Peer Architectural Style: This is a simple approach that consists of a monitor component to observe each aspect of the system or the environment. It also consists of a peer configurator component to reconfigure the system. This style is

similar to network protocol architectures where each level in the protocol stack has a peer-level counterpart at the other end of the communication link. The main advantage of this style is that it is conceptually simple. However, the drawback of such a strict peer-to-peer approach is that an actual configurator may require outputs from more than one monitor to make an optimal reconfiguration of the system. Figure 3.3 shows the aspect peer-to-peer architectural style.

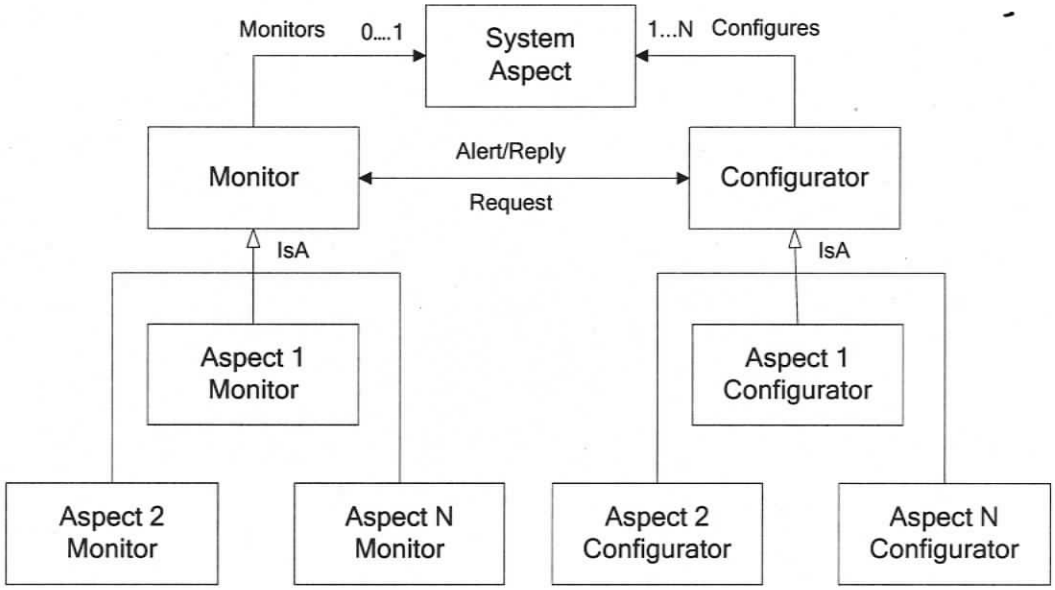

Figure 3.3. Aspect peer-to-peer architectural style

This style can also be viewed as a set of autonomic elements (i.e., one autonomic element exists for each component of the system). Further, each autonomic element in the set is independent and complete on its own.

Aggregator-Escalator-Peer Architectural Style: This style overcomes the limitations of a strict peer-to-peer monitor-configurator approach by allowing monitors to pass their outputs to higher-level monitors. This composite output is then passed to a peer configurator. Therefore, the higher level configurator can make better configuration decisions and thus operate more efficiently than a peer-to-peer monitor-configurator. Figure 3.4 shows the aggregator-escalator-peer architectural style.

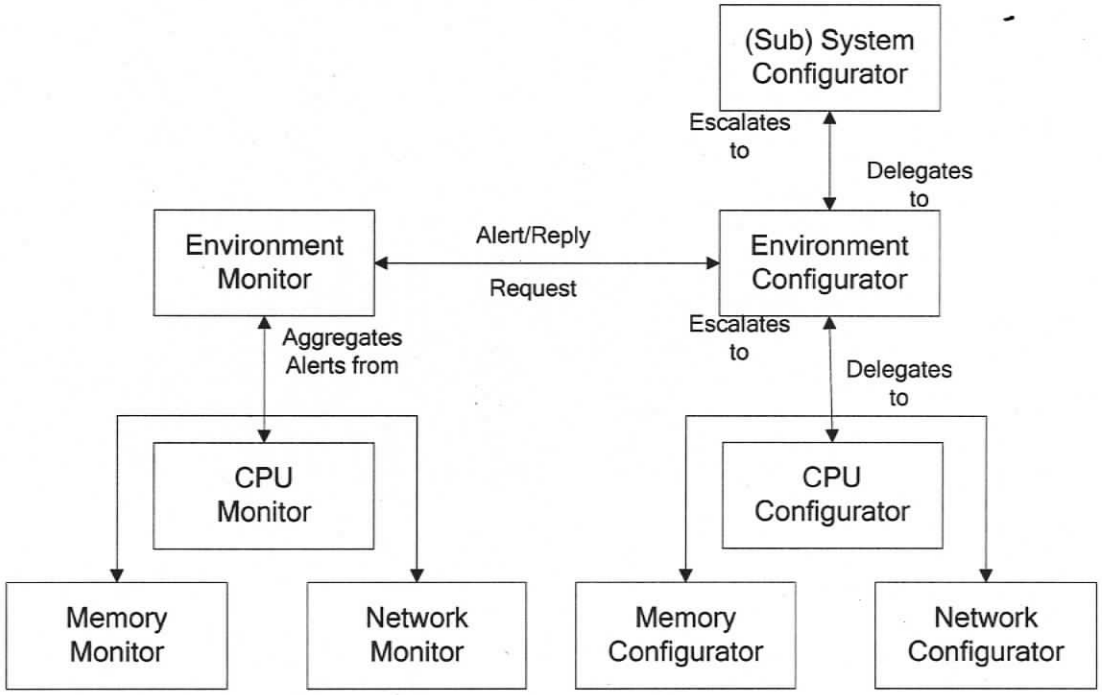

Figure 3.4. Aggregator-escalator-peer architectural style

Similar to the peer-to-peer style, this style can also be viewed as a set of autonomic elements (i.e., one autonomic element exists for each component of the

system). However, in this case each element is not completely independent. Higher level elements need information from lower level elements to function properly.

Chain-of-Configurators Architectural Style: This style comprises of two different design patterns. The first one is similar to the chain-of-responsibility design pattern where multiple configurators are chained together [29]. The configuration request is passed along the chain until it is successfully handled. This enhances loose coupling and also makes runtime modifications easier. This style allows self-healing systems to try all available configuration strategies to repair a given problem. The system can then promote successful strategies and demote less successful strategies while the system is running—just by manipulating its configurator list. The second chain-of-configurators variant uses a visitor pattern in which configurator chaining is used to compose higher order configuration functionality [29]. This style enhances the flexibility of the adaptation mechanism by allowing construction of new configuration solutions from existing lower-level solutions. The combination of the two approaches enhances loose coupling. Figure 3.5 shows the chain-of-configurators architectural style for self-healing systems.

In this style, each configurator is separated from its corresponding autonomic element but all the configurators are chained together. In this case, a single monitor is allocated to the entire system. For each specific problem, an optimum configuration can be chosen as a solution. Thus, the chain-of-configurators architectural style

enhances the flexibility and loose coupling compared to the other styles where the monitor and configurator are fixed.

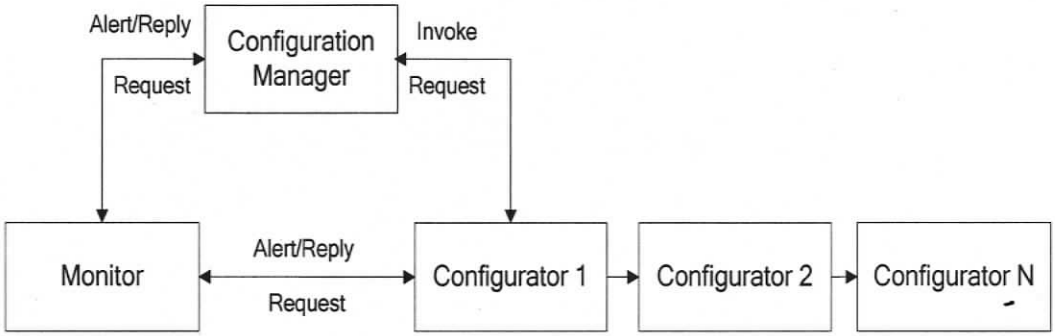

Figure 3.5. Chain-of-configurators architectural style

3.5. Summary

Architectural styles are important for the analysis and evaluation of existing self-healing systems as well as for the design and implementation of new self-healing systems. In this chapter, different architectural styles for self-healing systems have been analyzed. In the following chapters, these architectural styles are evaluated based on quality attribute models.

Chapter 4
Attribute Based Architectural Styles for Self-Healing Systems with Respect to Traditional Qualities

4.1. Introduction

The purpose of this work is to move the notion of architectural styles towards reasoning, either quantitative or qualitative, based on quality attribute specific models. The notion of attribute based architectural style (ABAS) is useful to analyze and document the architecture of self-healing systems. The idea is to collect and document design information, such as tradeoffs, artifacts, diagrams, and design decisions, in a consistent manner that is most relevant for the understanding and evolution of self-healing systems.

In this chapter, we give a brief introduction to attribute based architectural styles, followed by a discussion on the framework developed for traditional qualities.

4.2. Attribute Based Architectural Styles (ABASs)

Attribute-based architectural styles are an extension of the notion of architectural styles [23]. ABAS is a reusable design component in much the same way as a design

pattern [29]. ABASs provide a foundation for more effective reasoning of architectural design by explicitly associating a reasoning framework, either qualitative or quantitative, with an architectural style. It allows a designer to predict the behavior of architecture with respect to a specific quality attribute. For each attribute a separate reasoning framework is developed. The reasoning framework is based on the establishment of a quality model that is specific to a quality characteristic. Such a quality characteristic is called an attribute in the ABAS approach. Only one attribute at a time is considered when ABASs are used in the design or analysis, because ABAS is associated with only one attribute reasoning framework. For example, if an architectural style is interesting from both performance and reliability points of view, ABASs for both performance and reliability should be developed. The design and analysis of software architecture is based on reusable design components such as patterns of software components with predictable properties. The information for characterizing a quality attribute of an ABAS is divided into three categories:

- External stimuli that cause the architecture to respond and/or change.
- Responses such as measured quantities and/or attributes desirable in the architecture.
- Architectural decisions that have a direct impact on achieving attribute responses.

The main purpose of an ABAS is to consistently organize the existing specialized body of knowledge in each of the quality attributes communities. This knowledge can be reused in every ABAS related to a particular quality attribute. Klein *et al.* define the structure of an ABAS to consist of four parts [23]:

Problem description: A description of the design and analysis problem that the ABAS is intended to solve, including the quality attribute of interest, the context of use, constraints, and relevant attribute-specific requirements.

Stimulus/response attribute measures: A characterization of the stimuli to which the ABAS is expected to respond to and the measurement of the quality attribute of that response. A stimulus means the event that causes the system to respond to or change. Responses are the quantities measured or observed in the desirable requirements or attributes of the architecture.

Architectural style: An architectural style consists of the following:

- Components
- Connectors
- Properties of the components and connectors.
- Patterns of data and control interactions.

Analysis framework: It is the description of the formal relation between the quality attribute models and the architectural style as well as the conclusions about architectural behaviour.

4.3. Traditional Quality ABASs

In this section, ABASs are described with respect to the traditional qualities. In order to construct an ABAS for the modifiability quality criterion, we considered the architectural styles mentioned in the previous chapter and used qualitative reasoning to formulate this ABAS [24]. For the reliability quality criterion, we considered the architectural style that belongs to a general family of reliability styles and employed quantitative reasoning to define this ABAS [31].

4.3.1. Modifiability ABASs

Figure 4.1 shows the ABAS characteristics for the modifiability quality criterion. Each of these use the ABAS structure defined in the previous section.

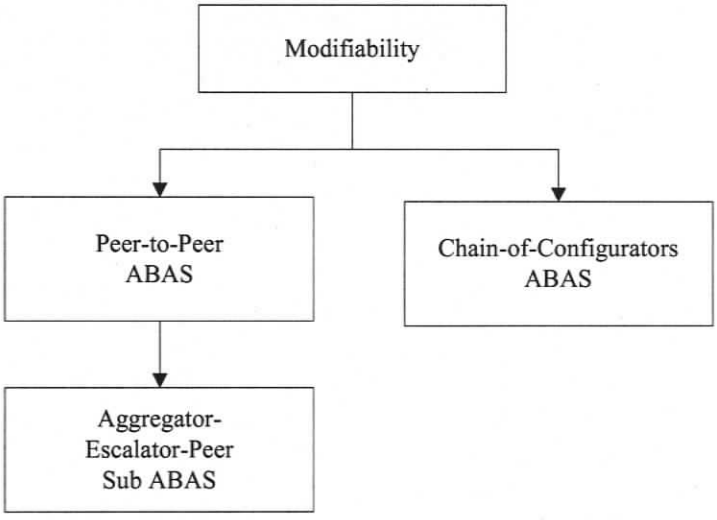

Figure 4.1. Characterization of the modifiability ABAS

4.3.1.1. Peer-to-Peer Modifiability ABAS

1. Problem Description: The peer-to-peer architecture is conceptually simple. A component in this architecture has no knowledge or dependencies on components below it. The architecture hides implementation details and thus provides modifiability at different levels of abstraction.

2. Stimulus/Response Attribute Measures: The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: a change in a peer layer

Response: number of peers affected, number of components or connectors added, number of components or connectors deleted or modified

3. Architectural Style: The peer-to-peer architectural style was described in the preceding chapter. It consists of a monitor component to monitor each aspect of the system or the environment, as well as a peer configurator component to reconfigure the system.

Parameters: The architectural parameters of concern for the ABAS can be described as follows:

Topology: peer to peer

Connectivity: no connectivity between peers.

4. **Analysis Reasoning Framework:** The basic idea of a modifiability ABAS is to describe the effect of changes on other components in the system. Changing a component at a peer level affects only the components at that peer level. This change does not affect or propagate to any other components at different levels. The analysis investigates the response of the ABAS to a set of change scenarios. In particular, it examines how well the architecture supports the following modification scenarios:

   a. **Adding a new monitor:** necessitates adding a new configurator. The addition of a monitor and a configurator together forms a new peer level.

   b. **Adding a new configurator:** requires adding a new monitor. The addition of a monitor and a configurator together forms a new peer level.

   c. **Deleting a monitor:** requires deletion of the peer-level configurator. The deletion of the monitor and the configurator requires the deletion of one peer level.

   d. **Deleting a configurator:** necessitates deleting the peer-level monitor. The deletion of the monitor and the configurator results in the deletion of one peer level.

5. **Design Heuristics Discussion:** The architecture is simple because it constitutes a strictly peer-to-peer approach. Following are some important design considerations to keep in mind when creating a peer-to-peer architecture.

- Choose this ABAS for small systems with a few monitors and configurators since extra resources are needed to respond to each low level monitor alert separately.
- Choose this ABAS if the configurator does not require output from more than one monitor to make a decision.
- A strict peer-to-peer approach decreases coupling which in turn increases system modifiability. But the modifiability comes at the cost of operational efficiency since a better configuration decision cannot be made.
- The architecture does not support run-time modification of the system.

4.3.1.2. Aggregator-Escalator-Peer Modifiability Sub ABAS

Following is the description of aggregator-escalator-peer ABAS. This ABAS is a sub ABAS to the peer-to-peer ABAS.

1. Problem Description: It overcomes limitations of the strictly peer-to-peer approach by delegating the output from lower level components to higher level components.

2. Stimulus/Response Attribute Measures: The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: introduction of a new monitor or configurator; modification of an existing monitor or configurator.

Response: number of components or connectors added, deleted, or modified.

3. Architectural Style: The architectural style of aggregator-escalator-peer was described in the preceding chapter.

Parameters: The architectural parameters of concern for the ABAS can be described as follows:

Topology: Aggregator-escalator-peer

Connectivity: to higher level components

4. Analysis Reasoning Framework: We now examine how the ABAS reacts to some of the scenarios defined in the peer-to-peer ABAS. In this style, introducing a new monitor or configurator affects the components at a higher level. Following are the modification scenarios:

a. **Adding a new monitor:** necessitates change in higher level monitors to be able to use output from the new monitor.

b. **Adding a new configurator:** requires a change in the higher level configurators to be able to use the output from the new configurator.

c. **Deleting a monitor:** requires a change in the higher level monitors such that they do not send or receive messages from this monitor.

d. **Deleting a configurator:** necessitates a change in the higher level configurators so that they do not send or receive messages from this configurator.

5. Design Heuristics Discussion: Following are some important design considerations to keep in mind when creating an aggregator-escalator-peer architecture.

- Gives full monitor output, allowing better configuration decisions and therefore more efficient operation.
- Gives consistent output.
- Increases the coupling between components (by increased interaction) thereby decreasing the system's modifiability.
- Each layer comes at the cost of a performance penalty due to the overhead of the interactions made when traversing the layers.
- This architecture does not support run-time modification of the system.

4.3.1.3. Chain-of-Configurators Modifiability ABAS

This modifiability ABAS features the chain-of-responsibility design pattern and visitor design pattern. The architecture chains the receiving objects and passes the request along the chain until an object handles it.

1. **Problem Description:** The ABAS enhances loose coupling and eases run-time addition and removal of configurator instances.

2. **Stimulus/Response Attribute Measures:** The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: introduction of a new monitor or configurator; modification of an existing monitor or configurator.

Response: number of components or connectors added, deleted or modified.

3. **Architectural Style:** Chain-of-configurators architectural style is described in previous chapter.

Parameters: The architectural parameters of concern for the ABAS can be described as follows:

Topology: data structure such as list

Connectivity: number of configurators (i.e., size of the list)

4. **Analysis Reasoning Framework:** We now examine how the ABAS responds to the change scenarios outlined in the peer-to-peer ABAS. In this style, introducing or deleting a configurator does not affect any other configurators in the chain. Following are the modification scenarios:

a. **Adding a new monitor:** necessitates a change in the adjacent monitors to use output from the new monitor.

b. **Adding a new configurator:** this change does not affect any other component.

c. **Deleting a monitor:** requires a change in the adjacent monitors such that they do not send or receive messages from this monitor.

d. **Deleting a configurator:** this change does not affect any other component.

5. Design Heuristics Discussion: There are some important design considerations to keep in mind when creating a chain-of-configurators architecture.

- This ABAS can be chosen if run-time addition and removal of configurators is needed. This can be achieved by simply manipulating the list at run-time.

- This ABAS can be chosen if it is needed to select an optimal configuration solution from a set of existing ones. The modifiability has a negligible performance cost.

4.4. Reliability ABAS

1. Problem Description: The ABAS focuses on the problem of reliability and availability in self-healing systems. Fault tolerance has long been associated with systems at design time [30]. The correctness of these systems is enforced at design time. But the features of today's systems are consumer-oriented and highly dynamic (e.g., evolving user requirements, evolving environment, quality of service, etc). This self-healing ABAS deals with fault tolerance for dynamic systems. It addresses how

self-healing systems can deliver the functionality, in all situations, possibly with different levels of quality of service (QoS). The architecture of self-healing systems has to support a space of possible behaviours which can be realized through a solution space of possible configurations. Self-healing systems can select an alternative configuration in case of a failure with minimal loss and without compromising reliability and availability. Variability in these systems can be introduced at design time (e.g. by implementing goal graphs [42]) or at run time (e.g. by changing some of the parameters) such that it can support possible behaviours.

2. Stimulus/Response Attribute Measures: The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: introduction of a fault in the system

Response: the levels of degraded service; reliability and availability for each level of service.

3. Self-Healing Fault-Tolerant Architectural Style: The self-healing fault-tolerant architectural style is a derivative of the general family of redundancy style in which different redundant components represent different behaviors or configurations. Analytically, these components are equivalent with respect to their behavior or configuration. Figure 4.2 depicts the reliability architectural style. The components R_1, R_2, ..., Rn represent the similar behavior.

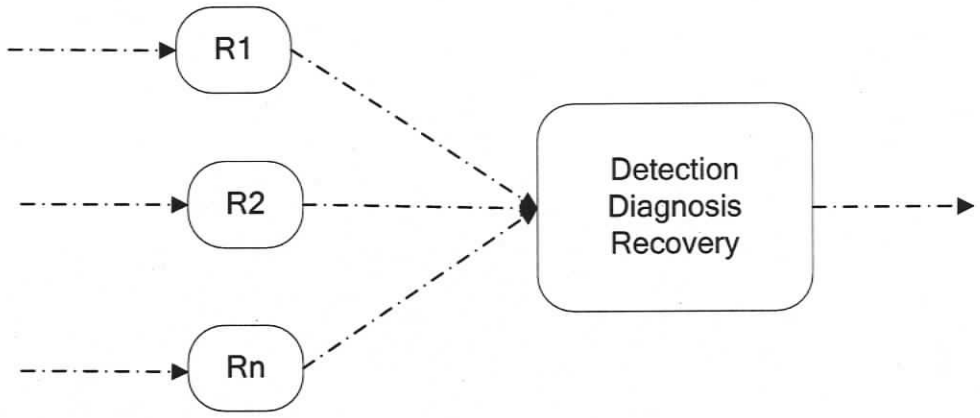

Figure 4.2. Reliability architectural style

Parameters: The architectural parameters of concern for the ABAS can be described as follows:

State transition probabilities λ_1, λ_2, λ_3, λ_4, and λ_5.

4. Analysis Reasoning Framework: The reliability modeling framework proposed by Park *et al.* is used for the analysis of a reliability ABAS [31]. This model incorporates self-healing capability into traditional reliability models. Figure 4.3 depicts the reliability model for self-healing systems. The interpretation of the state transition probabilities are as follows:

- Failure rate: $\lambda 1$
- Failure rate (transition from working state to failure state): $\lambda 2$
- Disease rate: $\lambda 3$
- Healing rate: $\lambda 4$

- Failure rate (transition from healable failure state to failure state): λ5

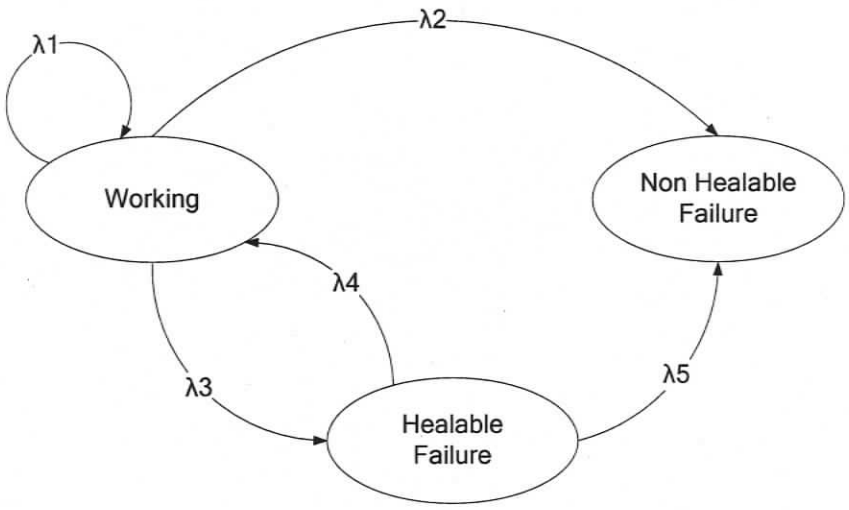

Figure 4.3. Self-healing reliability model

The model has the following three states:

Working State: This is the normal operational state of the system.

Healable Failure: Transient failure due to a change in the designed operating mode, accumulated component or resource faults, component evolution, changes in system usage, etc. These kinds of failures can be predicted by detecting anomalies in the system or environment. The system can then switch to a different configuration and finally recover back to its normal operational behavior.

Non Healable Failure: Permanent failure due to failure of a component. In this case, the component is replaced with a provided redundant component. This is accomplished with traditional fault tolerance techniques [30].

5. Analysis and Design Heuristics Discussion: Repairing the system from a reduced level of service can be more efficient than repairing from a completely failed state. Hence, the reliability and availability of the system increases in the case of healable failure. As the redundancy increases, the reliability increases due to the combination of traditional fault tolerance and self-healing techniques.

4.5. Summary

In this chapter, an analysis and reasoning framework for the self-healing systems has been developed with respect to traditional qualities. One reliability ABAS and three modifiability ABASs have been defined for self-healing systems. ABASs have been used to analyze and document the architecture of self-healing systems. This analysis will likely prove useful for the understanding and evolution of self-healing systems.

Chapter 5

Attribute Based Architectural Styles for Self-Healing Systems with Respect to Autonomic-Specific Qualities

5.1. Introduction

In the previous chapter, we gave a brief introduction to ABASs and discussed the analysis and reasoning framework with respect to traditional qualities. In this chapter, we discuss the analysis and reasoning framework with respect to autonomic-specific qualities. The framework, including traditional and autonomic-specific qualities, can be used together to collect and document design information that is most relevant for the understanding and evolution of self-healing systems.

5.2. Autonomic-Specific Quality ABASs

In order to construct ABASs for the autonomic-specific qualities, the architectural styles mentioned in Chapter 3 are considered. Through this framework, a set of pre-packaged analyses and questions are provided, based upon both known solutions to commonly recurring problems and known difficulties in employing those solutions.

Qualitative reasoning is used to formulate these ABASs. Each of these use the ABAS structure defined in the previous chapter.

5.2.1. Support for Detecting Anomalous System Behavior ABASs

In order to construct an ABAS for this quality, the relevant architectural styles belonging to the family are considered. Figure 5.1 shows the ABAS characteristics for the support for detecting anomalous system behaviour quality criterion. We defined three ABASs for this quality:

1. Peer-to-peer ABAS
2. Aggregator-escalator-peer ABAS and
3. Chain-of-configurators ABAS.

The aggregator-escalator-peer ABAS is considered a sub-ABAS of the peer-to-peer ABAS. Following are the descriptions of these ABASs.

5.2.1.1. Peer-to-Peer Support for Detecting Anomalous System Behaviour ABAS

1. Problem Description: The peer-to-peer style has been described in Chapter 3 on architectural styles for self-healing systems. The problem is to observe whether the architecture is supporting this autonomic quality.

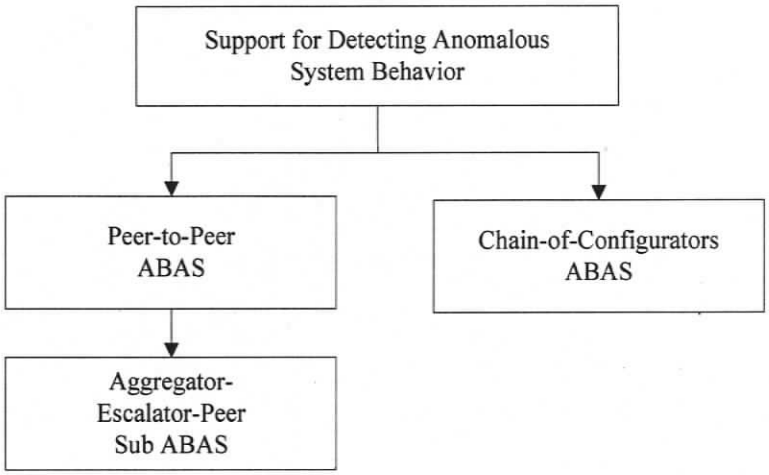

Figure 5.1. Characterization of the Support for Detecting Anomalous System Behaviour ABAS

Criteria for choosing the ABAS:

This ABAS can be relevant if:

- A peer-to-peer topology is used.
- A problem has requirement to support for detecting anomalous system behavior.

2. Stimulus/Response Attribute Measures: The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: A fault in the system.

Response: Is the system detecting the anomaly caused due to this fault?

The response can be obtained using the following measurable quantities [13, 18, 19]:

Detection rate: The fault detection rate needs to be good to satisfy the quality requirements.

Detection time: A fault should be detected as early as possible to avoid system degradation failure.

Coupling: High Coupling causes dependencies among sub-systems which in turn have impact on other factors like complexity etc.

Awareness: This is achieved by internal monitoring of the system. Monitors can be directly bonded to components, connectors, portals or ducts. Design for testability [32] techniques can substantially improve the effectiveness of monitoring the internal states of the system. This addresses the effect of lack of information in automatically managed components.

Observability: This is achieved by external monitoring of the system. External monitoring is critical in resource constrained, mobile environments [12]. There should be few constrained topological rules for attaching to environment facilities.

Fault model: Defining a fault model is necessary for any self-healing system [13]. Self-healing systems must have a fault model describing the faults they are expected to self-heal. Without a fault model, there is no way to assess whether a system actually can heal itself in situations of interest. A fault model is expected to identify the abnormal behavior of the system. It is also expected to decide the fault it can self-heal.

3. Architectural Style: Figure 3.3 depicts the peer-to-peer architectural style.

4. Analysis Reasoning Framework: This analysis investigates the response of the ABAS to a set of scenarios. These scenarios are used to elicit requirements. Following are some of the scenarios in the form of questions identified from the literature:

- What were the architectural decisions during the system design?
- If the system did not recover completely from a fault, how to identify them?
- Does the system know if it crashes?
- Is the architecture allowing instrumentation?
- What are the assumptions in the implementation architecture?
- Is the instrumentation with respect to qualities?
- How to predict normal behavior if the precise specification of the system is not available?
- Is the knowledge module evolved with the system evolution?
- How the system does handle bug fixes?

Reasoning: For the defined scenarios, the architecture is analyzed using the defined measures for its support of this autonomic quality.

Coupling: Loose coupling means no dependencies between components. High coupling increases the complexity and hence can increase the rate and time of anomaly detection. In this case there are no dependencies between peers and hence there is loose coupling. Components at the peer level are dependent on each other. Further, loosely coupled systems can be easily reconfigured.

Awareness: There is a direct binding of components with the monitors. A separate monitor is assigned to each component. But the monitors do not interact with each other and hence they have no knowledge of the other monitors in the system. When autonomic actions are based on local information and limited environmental information, the ability to make accurate decisions may be limited [12].

Observability: There should be few constrained topological rules for attaching to environment facilities [12]. Since there is a loose coupling, the architecture can be easily modified but this cannot be done at run time

Fault model: It is suitable for small systems only.

Design Heuristics Discussion: When designing a system that employs such an ABAS, the designer should ask the following questions:

- What properties of the execution system should be monitored?
  - Properties could be correctness, quality of service, etc.
- What constraints need to be evaluated?
  - Constraints could be reliability or availability.
- What should the system do when constraints are violated?
  - Change in the architecture could be a solution.
- How must the architecture change in order to affect certain properties?
  - It can be done by keeping the architectural decisions intact.
- Is the architecture supporting awareness and observability?
- Is the architecture open to accommodate life-cycle evolution?

To realize this ABAS, the designer should consider the following:

- **Instrumenting software systems to gather information about the changes:** It can be static instrumentation: inserting additional code to the original application or dynamic instrumentation: using probes and gauges [12].
- **Simulations to find the difference between the expected and the actual behavior:** This indicates system abnormality [34].
- **Design for testability techniques** [32]: Checking frequency of raised exceptions and run-time check violations.
    - Monitoring log files
    - Monitoring assertions and variations
    - Monitoring environmental errors
    - Monitor changing environment such as end-user input or external devices and probes
- **Instrumentation to observe hard-wired requirements:** The system requirements itself can change.

5.2.1.2. Aggregator-Escalator-Peer Support for Detecting Anomalous System Behaviour Sub ABAS

This ABAS is a sub ABAS of the peer-to-peer ABAS. Only the aspects that are different from the peer-to-peer ABAS are discussed.

1. Problem Description: This is also a peer-to-peer architectural style with communication between peers. Information from lower level components is aggregated and passed to higher level components. In Chapter 3, the style has been described on architectural styles for self-healing systems. The problem is to observe whether the architecture can support this autonomic quality.

Criteria for choosing the ABAS: This ABAS can be relevant if:

- An aggregator-escalator-peer topology is used
- A problem requires fault detection with efficient system operation

2. Architectural Style: Figure 3.4 depicts the aggregator-escalator-peer architectural style.

3. Analysis Reasoning Framework: The analysis investigates the response of this ABAS to a set of scenarios defined in peer-to-peer ABAS.

Reasoning: The architecture is analyzed using the measures defined in the peer-to-peer ABAS for its support of this autonomic quality.

Coupling: In this case there is coupling between components at the peer level as well as among different peers. Hence, the rate and time of anomaly detection can be affected. Further, coupling makes it difficult to reconfigure.

Awareness: There is a direct binding of components with the monitors. A separate monitor is assigned to each component. Also the monitors interact with each other and hence they have complete knowledge about other monitors in the system.

The received data from multiple data sources is used to make a more informed and accurate decision on what has or might have happened on a particular platform.

Observability: There should be few constrained topological rules for attaching the system to environment facilities. Since there is coupling among the peers, the architecture cannot be easily modified. Also this modification cannot be done at run time.

Fault model: It is suitable for large scale systems.

5.2.1.3. Chain-of-Configurators Support for Detecting Anomalous System Behaviour ABAS

1. Problem description: The chain-of-configurators style has been described in Chapter 3 on architectural styles for self-healing systems. The problem is to observe whether the architecture can support this autonomic quality.

Criteria for choosing the ABAS: This ABAS can be relevant if a chain-of-responsibility design pattern is used.

2. Architectural Style: Figure 3.5 depicts the chain-of-configurators architectural style.

3. Analysis Reasoning Framework: The analysis investigates the response of this ABAS to a set of scenarios defined in peer-to-peer ABAS.

Reasoning: The architecture is analyzed using the measures defined in the peer-to-peer ABAS for its support of this autonomic quality.

Coupling: In this case there is loose coupling between components. Hence, the rate and time of anomaly detection can be better. It can be easy to reconfigure due to the flexible architecture.

Awareness: In this case there is no separate monitor and configurator for each component of the system. There is no direct binding of components with the monitors instead there is a monitor for the whole system. This loose coupling can affect awareness.

Observability: There should be few constrained topological rules for attaching to environment facilities. Since the architecture enhances loose coupling, modifications can be easily done at run time.

Fault model: It is suitable for all kinds of faults. The optimum strategy is chosen for a given problem.

5.2.2. Support for Simulation of Expected or Predicted Behavior ABAS

1. Problem Description: Is the architecture supporting this quality? How can this quality be affected in case of third party components? Lack of software and

environment component models, such as COTS[1] component, legacy components, hardware and operating system, can be a problem.

Criteria for Choosing this ABAS: This ABAS can be relevant if:

- The architecture contains heterogeneous elements, third party elements, etc. System evolution and unexpected behaviors are not controllable in case of third party components.
- The system has a requirement to support the simulation of predicted behavior.

2. Stimulus/Response Attribute Measures: The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: Applying the real stimuli, the actual system is subjected, to the simulatable model.

Response: The behavior should be as predicted from observing the actual system.

The response can be obtained using the following measurable quantities [34]:

Correctness of the model: The model should be correct and able to produce expected results.

Completeness of the model: The model should completely define the underlying executing system.

---

[1] COTS – Commercial off-the-shelf.

Consistency of the model: A model should be consistent to produce same behavior on applying stimuli.

Complexity of the model: Complexity of the model can make analysis of the behavior impractical.

Coupling: In a system, coupling can increase complexity.

Awareness: Complete knowledge of the system is needed.

3. **Architectural Style:** Architectural styles have been discussed in Chapter 3.

4. **Analysis Reasoning Framework:** The analysis investigates the response of the ABAS to a set of scenarios. These scenarios are used to elicit requirements. Following are some examples of the scenarios identified from the literature showing quality requirements:

- Complete knowledge of the system is needed. This quality requires a complete model of the software system and all relevant environmental aspects that affect it.
- The model should evolve with the evolution of the system otherwise the behaviour cannot be analyzed.
- The system should be a reactive system that responds to external stimuli.
- Simulation should support different test scenarios.
- The model is assumed to be in correct state.

Reasoning: For the defined scenarios, the architecture is analyzed using the defined measures for its support of this autonomic quality.

The behavior of a system results from the behaviors of its components and from the interactions and relationships among them. Complexity of the system increases in case of highly coupled systems. This can make behavior prediction difficult. The simulated model should be correct, complete, and consistent. Rigorous testing should be performed beforehand.

Design Heuristics Discussion: When designing a system that employs such an ABAS, the designer should ask the following questions:

- How to observe real stimuli?
- What are the assumptions in simulations? Otherwise it is difficult to compare the responses between the model and the system.
- What are the exceptions or differences between the simulation and the model?
- What are the best suited problems and behaviors?
- What are the system requirements?
- Can simulation be done on any architecture and design models?

To realize this ABAS, the designer should consider the following:

- Systems exhibit many forms of predictable and unpredictable behaviors that are imposed by their environments. It becomes difficult to model every situation. Therefore it becomes important to understand a component's

environment (i.e., the way components can be composed, moved, interchanged, or upgraded).

- The behavior of a system results from the behaviors of its components as well as from the interactions and relationships among various components. In order to create computing systems that manage themselves, it is required to design both the behaviors of the individual elements and the relationships that are formed among them.
- Testing of the model beforehand is necessary.

5.2.3. Support for Differencing between Expected and Actual Behavior ABAS

This ABAS is relevant if the system has a requirement to detect the difference between the expected and actual behavior of the system. During the design time, the purpose of this differencing is to detect inconsistencies between the model and the system [34]. A difference between the model behavior and the system behavior indicates system abnormality (e.g., fault, attack, or misuse). These inconsistencies can be corrected by updating the model or the system. Differencing requires a simulatable, behavioral model of the system. The model simulation is then a reflection of the state and behavior of the executing system. They are both expected to respond equivalently given the same input stimuli. If they do not respond equivalently then the model behavior differs from the system behavior. This requires an accurate model of the

system to obtain the expected behavior of the running system. Simulation serves as a foundation for self-awareness through which differences among expected behaviour and actual behaviour can be investigated [34]. So this quality depends on "support for simulation of predicted behaviour" quality. An ABAS for this quality depends on an another quality ABAS, the support for simulation of predicted behaviour ABAS, for determining the support for differencing between expected and actual behaviour.

5.2.4. Support for Failure Diagnosis ABAS

1. **Problem Description:** The problem is to observe whether the architecture is supporting this autonomic quality.

Criteria for choosing this ABAS: This ABAS can be relevant if the problem has the requirement to support failure diagnosis

2. **Stimulus/Response Attribute Measures:** The primary stimulus and its measurable responses can be characterized as follows:

Stimulus: A fault is detected in the system

Response: How quickly and effectively can a problem be tracked?

The response can be obtained using the following measurable quantities [35-37]:

Complexity of the problem: Complex systems can make diagnosis difficult.

Diagnosis rate: Systems need to be able to localize a large percentage of the errors in order to gain confidence from the system recovery operator.

False positive rate: False positives, which are correct behavior of the system that are mistaken for an error, are costly in terms of time wasted on recovery and re-diagnosis.

Robustness to noise: Few systems are failure-free. It is common to find a small number of failures at any moment on a large system. An ideal algorithm should be able to filter out noise and prioritize faults that are impacting the availability of a system.

Real time turn around: The algorithm needs to be fast enough in order to derive benefits over manual approaches.

3. **Architectural Style:** Architectural styles have been discussed in Chapter 3.

4. **Analysis reasoning framework:** The analysis investigates the response of the ABAS to a set of scenarios. These scenarios are used to elicit requirements. Following are some examples of the scenarios identified from the literature showing quality requirements [35-37]:

- Improve system availability;
- Gather run time information for statistical analysis; and
- Improve diagnosis performance.

Reasoning: For the defined scenarios, the architecture is analyzed using the defined measures for its support of this autonomic quality.

Highly interconnected systems can make diagnosis difficult, inefficient, and expensive. For example, a failure in a single component, such as a database, can quickly cause entire business systems to fail, generating symptomatic alerts from multiple downstream components. The resulting confusion can lead to wasting time and effort in diagnosis. Detection and isolation of failures in large, complex systems is a crucial and challenging task.

Design Heuristics Discussion: When designing a system that employs such an ABAS, the designer should ask the following questions:

- How quickly and effectively can the problem be diagnosed?
- How can diagnosis performance be improved?

To realize this ABAS, the designer should consider the following:

- The system needs to be self-aware.
- The system needs to know when and where an error state occurs.
- The system needs to have adequate knowledge to stabilize them.
- The system needs to be able to analyze the problem situation.
- Assertions can help isolate the locations of bugs. Assertion violations typically report the location of the assertion code.
- The need for sophisticated and systematic methods for the timely and accurate diagnosis of system failures.

5.2.5. Support for Testing of a Correct Behavior ABAS

This ABAS is important since it is harder to predict the behavior of an autonomic system especially when it extends across multiple domains [1]. Autonomic elements have to be thoroughly tested and verified that they behave correctly. Various parts of the system must perform self-tests in order to check whether they still function correctly. This ABAS points to a testability ABAS for the design techniques that can improve this quality in autonomic systems [23].

5.3. Summary

This chapter developed an analysis and reasoning framework for self-healing systems with respect to autonomic-specific qualities. We defined three ABASs for "support for detecting anomalous system behaviour," one ABAS for "support for simulation of expected or predicted behaviour," one ABAS for "support for differencing between expected and actual behaviour," one ABAS for "support for failure diagnosis," and one ABAS for "support for testing of correct behaviour" for self-healing systems. Now that we have developed a collection of ABASs for self-healing systems—traditional as well as autonomic ABASs, we can use these ABASs together to analyze existing architectures for self-healing systems.

Chapter 6
Analysis of Self-Healing Systems using Attribute Based Architectural Styles

6.1. Introduction

In the previous chapters, we defined an analysis and reasoning framework for self-healing systems. In this chapter, the proposed framework is applied to an existing architecture for analysis. This analysis illustrates the usefulness of the notion of an ABAS to analyze and document the architecture of self-healing systems.

6.2. Using ABASs in the Analysis of Self-Healing Systems

Now that we have a collection of ABASs for self-healing systems, traditional as well as autonomic ABASs, how can these ABASs be used together to analyze existing architecture? We introduced a model problem to help illustrate the use of ABASs in analysis for self-healing systems. Every ABAS is associated with an analytical model which is used to understand the impact of architectural decisions that have already been made.

As a first step in the analysis, the quality attributes that are most critical to the system's success are extracted from the architecture. Using this information, the architecture can then be probed for the styles that are used to satisfy the attribute-

specific requirements. Once these ABASs are identified in the architecture, the analytical models associated with them can be applied to understand the ramifications of architectural decisions on the quality-attribute goals.

6.3. Case Study: Model for a Self-Managing Java Server

The model for a self-managing Java server is a working model of a non-stopping, Java-based server (which could be a web server or an application server) [33]. The server has the self-configuration and self-healing capabilities of an autonomic, self-managed server.

In any Java server when an abnormal event occurs during a transaction, the server logs the exception or error and aborts the current transaction. The server is not capable of dynamically managing the problems, and it cannot handle new transactions. The problem has to be analyzed and fixed by the administrator. This leads to a waste of time and labor, which might be very costly for critical businesses. An automated solution can solve this problem. In this model, the Java-based server is converted into a non-stop server by taking a few corrective steps in between the normal flow. These steps are as follows:

1. The server throws an exception or error and logs it.
2. A self-healing module of the autonomic computing system gathers the exception information by parsing the log file.

3. An analyzer module analyzes the cause and decides the necessary action to be taken using the predefined policies.
4. The healing module then takes the appropriate action.

The model of the self-managing Java server has the following components:

Analyzers: Used to analyze the cause of the problem.

HealingManager: Manages the overall function of the model.

PolicyManager: Reads the rules from the policy file and gives instructions to HealingManager to decide which healer should be delegated for a particular operation.

Healers: Cater to different problems. For example, the "ClassNotFoundHealer" fixes the "ClassNotFoundException" and the "FileNotFoundHealer" fixes the "FileNotFoundException". Healers also interact with the PolicyManager to decide on the action taken as part of the healing process.

Policies: A predefined set of rules to fix the problem.

The self-managing Java server model normally works using the following steps:

1. Analyzer is configured to poll a target medium for exceptions or errors. Once an exception or error occurs, it analyzes the cause and delegates it to the HealingManager.
2. From the information it receives from the Analyzer, the HealingManager uses the PolicyManager to decide which Healer to delegate the problem to.

3. Once the control is delegated to a particular Healer, the Healer retrieves rules from the PolicyManager for fixing the problem.
4. The entire process is logged into a file for reference.

Figure 6.1 illustrates the complete message flow in the Java server model.

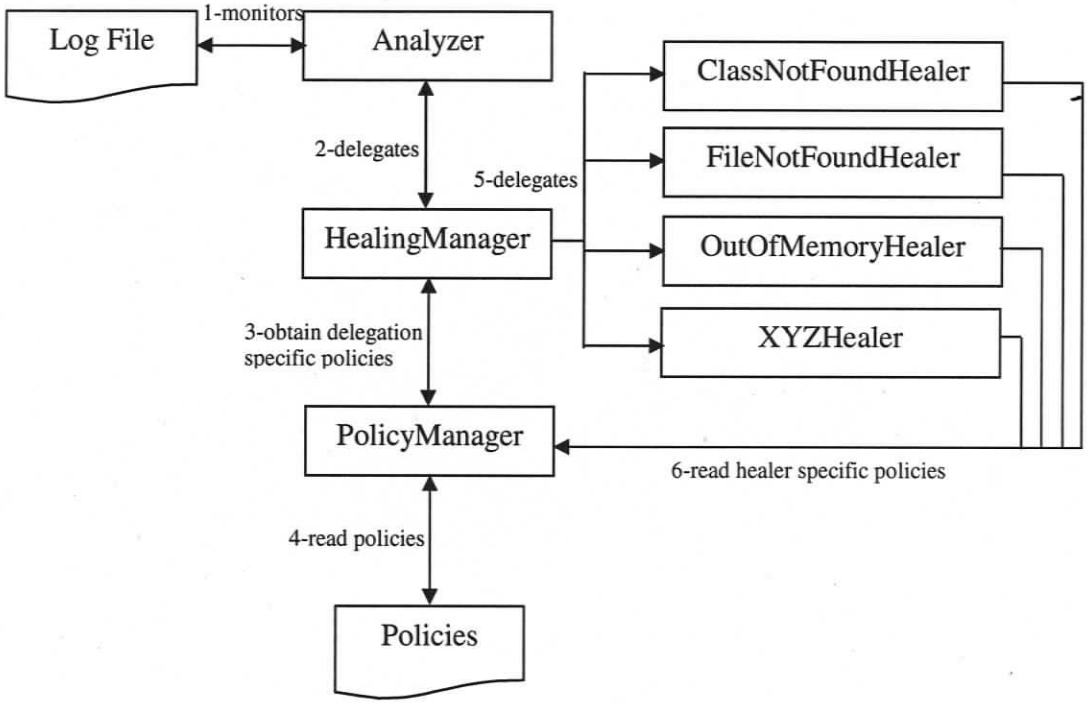

Figure 6.1. Message Flow in the Java server model

6.3.1. Analysis

This section presents the details of the analysis of the problem with the Java server. The following quality attributes are considered critical to the system's success:

Modifiability This quality is an important attribute for any system since a highly modifiable system affords a highly maintainable system.

Support for detecting anomalous system behaviour This quality is important since automated systems should have the ability to monitor, recognize, and address anomalies with respect to its performance or environment.

Support for failure diagnosis Once a problem is detected, the automated system should be able to precisely locate the source of the misbehaviour or failure.

Now, the next step is to probe the architecture for the styles that are used to satisfy the attribute-specific requirements. From the above description of the Java server, it is clear that the chain-of-configurators architectural style is used for this system. Some of the characteristics of the style are as follows:

- Multiple configuration solutions are chained together. In this model, different Healers are assigned to cater to different problems. An appropriate Healer can be selected, from a set of existing ones, for a given problem.

- The HealingManager passes the configuration request along the list of Policies until an appropriate Healer is selected to handle the request successfully.

- The architecture enhances flexibility and loose coupling. There is no direct binding or dependency between the Analyzers and the Healers. An Analyzer and/or Healer can be independently added or deleted. Further, there is no

dependency between different Analyzers. The Analyzer monitors for a particular destination (LogFile, Console, and so on) for any changes and analyzes the cause. For example, LogAnalyzer monitors the log file and analyzes the cause, and DynamicAnalyzer analyzes from exceptions thrown by the JVM.

Based on the above characteristics of this architectural style and the required quality attributes, the chain-of-configurators modifiability ABAS, the chain-of-configurators support for detecting anomalous system behaviour ABAS, and the support for failure diagnosis ABAS are identified.

6.3.2. Results of the Analysis

Finally, the analytical models associated with the ABASs can be applied to understand the ramifications of architectural decisions on the quality-attribute goals.

1. **Applying the chain-of-configurators modifiability ABAS:**

   Design Decisions: The architectural decisions highlighted by the ABAS are:

   - The architecture enhances flexibility and loose coupling.
   - The architecture eases run-time addition and removal of configurator instances.

   Tradeoffs: Overhead in handling the chain of solutions. This can affect performance.

In this model, the HealingManager determines the current exception by going through the Policies (list of rules) in the policy file. Policies are only a property file. We can easily add or delete policies from the property file to handle different types of exceptions. To understand the ramifications of architectural decisions on the quality-attribute goals, we can use the analytical models associated with this ABAS. The following are some of the observed implications:

- The architecture enhances loose coupling. This modifiability comes at the cost of a performance penalty to be paid in terms of the overhead in handling the list of policies.
- The architecture allows run-time addition and removal of a policy as needed.
- The architecture allows selection of an optimal solution from a set of existing ones.

2. Applying the chain-of-configurators support for detecting anomalous system behaviour ABAS:

Design Decisions: The architecture enhances flexibility and loose coupling.

Tradeoffs: Loose coupling can affect awareness of the system.

An automated Java server should have the ability to address anomalies with respect to its performance or environment. The architecture should support this autonomic quality. To understand the ramifications of architectural decisions on the quality-attribute goals, we can use the analytical models associated with this ABAS.

This ABAS provides some measures that can be used to analyze the effect of the design decisions on the quality. The following are the observed implications:

- ***Coupling***: The architecture enhances loose coupling. Due to this isolation, the rate and time of anomaly detection can be better. It is also easy to reconfigure since the architecture is relatively flexible.
- ***Awareness:*** Loose coupling can affect awareness of the system.
- ***Observability:*** Since the architecture enhances loose coupling, it becomes easier to attach to environment facilities.
- ***Fault model:*** The model is able to handle a variety of exceptions since new Healers can be easily added to cater to different types of problems.

3. Applying the support for failure diagnosis ABAS:

Design Decisions: The architecture enhances flexibility and loose coupling.

Tradeoffs: Loose coupling can affect awareness of the system.

In the Java server model, the system should be able to locate the source of the misbehaviour or failure precisely. It has to decide which Healer should be delegated for a particular problem. The architecture should support this autonomic quality. To understand the ramifications of architectural decisions on the quality-attribute goals, we can use the analytical models associated with this ABAS. This ABAS provides some measures that can be used to analyze the effect of the design decisions on the quality. The following are the observed implications:

- ***Complexity of the problem:*** Since the architecture enhances loose coupling it is easier to isolate a problem.

- ***Diagnosis rate:*** Diagnosis becomes easier in case of a loosely coupled system. Hence, the rate and time of diagnosis can be better. But loose coupling can affect awareness of the system which in turn can affect the accuracy of the diagnosis.

- ***False positive rate:*** In highly interconnected systems, a failure in a single component can generate false positives from multiple dependent components. Since in this model the architecture is loosely coupled, false positives can be minimized.

- ***Real time turn around:*** The diagnosis approach is fast enough since the rate and time of diagnosis is better in this model.

In this way, ABASs provide a solid foundation for an improved and accelerated architecture analysis by furnishing a set of pre-packaged analyses and questions for the engineer.

6.4. Summary

In this chapter, we illustrated the usefulness of the notion of an ABAS to analyze and document the architecture of self-healing systems. The analysis showed how to collect and document design information, such as tradeoffs, artifacts, diagrams, and

design decisions, in a consistent manner. In this way, the proposed framework can facilitate the design and maintenance of self-healing systems.

Chapter 7
Conclusions and Future Work

7.1. Conclusions

We developed an analysis and reasoning framework to evaluate and document the architecture of self-healing systems. The ABAS analysis technique was used to tailor the framework to selected quality attributes. This reasoning framework can now be used to analyze the architecture of self-managing systems and to verify certain quality attributes. The key quality attributes for self-healing systems we identified include not only traditional quality attributes, but also autonomicity-specific attributes. Further, we customized the ISO 9126 quality model to the quality requirements of self-healing systems, considering both traditional attributes as well as newly defined autonomic-specific attributes.

The benefit of this framework is that it can be reused to verify certain properties when the environment of the subject system evolves. Moreover, it allows future developers to reuse the architecture analysis not only for the current architecture but also for other self-managing applications. The proposed framework can facilitate the design and maintenance of self-healing systems.

We also illustrated the usefulness of the notion of an ABAS for self-healing systems to analyze and document the architecture of autonomic systems using a case

study. The analysis described in Chapter 6 demonstrates how to collect and document design information, such as tradeoffs, artifacts, diagrams, and design decisions, in a consistent manner.

7.2. Contributions

- Defined quality criteria for self-healing systems. The criteria include newly defined autonomic-specific quality attributes. These criteria can be used not only for the evaluation of existing systems but also for the design and implementation of new self-healing systems.
- Customized the ISO 9126 quality model to the quality requirements of self-healing systems. This quality model can be further extended based on the new requirements.
- Developed an analysis and reasoning framework for self-healing systems with respect to traditional as well as autonomic-specific quality attributes.

7.3. Future Work

The proposed reasoning and analysis framework could be extended to other self-managing applications for a variety of quality attributes and architectural styles. We propose to record the relationship between architecture and quality attributes of self-managed systems using a reverse engineering handbook [44]. This could enhance the flexibility in choosing the right ABAS tailored to specific quality attributes.

Traditionally, applying stimuli and observing responses for architectural analysis is performed during system design and evolution. We predict that the autonomic-specific quality attributes can be analyzed by directly simulating events and observing responses on the running application which is already equipped with sensors/monitors and executors/effectors as an autonomic element.

Bibliography

[1] J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, Vol. 36, No. 1, pp. 41-50, January 2003.

[2] A.G. Ganek and T.A. Corbi, "The Dawning of the Autonomic Computing Era," *IBM Systems Journal*, Vol. 42, No. 1, pp. 5-18, 2003.

[3] IBM Corporation, "An Architectural Blueprint for Autonomic Computing," *Autonomic Computing White Paper*: Fourth Edition, June 2006. http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[4] M. Parashar and S. Hariri, "Autonomic Computing: An Overview," *Unconventional Programming Paradigms (UPP 2004), Lecture Notes in Computer Science, Springer Verlag*, Vol. 3566, pp. 247-259, 2005.

[5] M. Parashar and S. Hariri, "Autonomic Computing: Research Issues, Challenges and Opportunities," *Proceedings ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2003), Autonomic Computing Tutorial*, July 2003. http://automate.rutgers.edu/tutorials/aiccsa03-tutorial-slides/s2-ac-issues.pdf

[6] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," October 2001. http://www-306.ibm.com/autonomic/pdfs/autonomic_computing.pdf

[7] M. Salehie and L. Tahvildari, "Autonomic Computing: Emerging Trends and Open Problems," *Proceedings Design and Evolution of Autonomic Application Software (DEAS 2005),* pp. 1-7, St. Louis Missouri, USA, May 2005.

[8] J. O. Kephart, "Research Challenges of Autonomic Computing," *Proceedings IEEE/ACM International Conference on Software Engineering (ICSE 2005),* pp. 15-22, May 2005.

[9] Microsoft Corporation. *Dynamic Systems Initiative*, 2005. http://www.microsoft.com/dsi/

[10] D. M. Chess, A. Segal, I. Whalley and S. R. White, "Unity: Experiences with a Prototype Autonomic Computing System," *Proceedings IEEE International Conference on Autonomic Computing (ICAC 2004)*, pp.140-147, May 2004.

[11] M. Julie, C. H. Markus, "Evaluation Issues in Autonomic Computing," *Grid and Cooperative Computing (GCC 2004) Workshops, Lecture Notes in Computer Science, Springer Verlag*, Vol. 3252, pp. 597-608, 2004.

[12] D. Tosi, "Research Perspectives in Self-Healing Systems," *Technical Report,* 2004.
http://www.lta.disco.unimib.it/doc/ei/pdf/lta.2004.06.pdf

[13] P. Koopman, "Elements of the Self-healing System Problem Space," *Proceedings IEEE/ACM International Conference on Software Engineering Workshop on Architecting Dependable Systems (WADS 2003)*, pp. 1-6, May 2003.

[14] D. Garlan and B. Schmerl, "Model-based Adaptation for Self-healing Systems," *Proceedings ACM First Workshop on Self-Healing systems (WOSS 2002)*, pp. 27-32, November 2002.

[15] B. Schmerl and D. Garlan, "Exploiting Architectural Design Knowledge to Support Self-repairing Systems," *Proceedings 14th IEEE/ACM International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, pp. 241-248, Italy, July 2002.

[16] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems,* Vol. 14, No. 3, pp. 54-62, May/June 1999.

[17] S. Cheng, D. Garlan, B. Schmerl, J. Sousa, B. Spitznagel and P. Steenkiste, "Using Architectural Style as a Basis for Self-repair," *Proceedings IEEE/IFIP Working Conference on Software Architecture (WICSA 2002),* pp. 45-59, Montreal, August 2002.

[18] M.J. Hawthorne and D.E. Perry, "Architectural Styles for Adaptable Self-Healing Dependable Systems," *Proceedings IEEE/ACM International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA,* May 2005.
http://www.ece.utexas.edu/~perry/work/papers/MH-05-Styles.pdf.

[19] M. Mikic-Rakic, N. Mehta and N. Medvidovic, "Architectural Style Requirements for Self-healing Systems," *Proceedings ACM First Workshop on Self-Healing Systems (WOSS 2002)*, pp. 49-54, November 2002.

[20] E.M. Dashofy, A. van der Hoek and R. N. Taylor, "Towards Architecture-based Self-healing Systems," *Proceedindgs ACM First Workshop on Self-healing Systems (WOSS 2002)*, pp. 21-26, November 2002.

[21] D. S. Wile and A. Egyed, "An Externalized Infrastructure for Self-Healing Systems," *Proceedings IEEE/IFIP Fourth Working Conference on Software Architecture (WICSA 2004),* pp. 285-288, June 2004.

[22] M. Shaw, "Self-Healing: Softening Precision to Avoid Brittleness," *Proceedings IEEE/IFIP First Workshop on Self-Healing Systems (WOSS 2002)*, pp. 111-114, November 2002.

[23] M. Klein, R. Kazman, L. Bass, J. Carrière, M. Barbacci and H. Lipson, "Attribute-Based Architectural Styles," *Proceedings IEEE/IFIP First Working Conference on Software Architecture (WICSA 1999),* pp. 225-243, San Antonio, TX, February 1999.

[24] R. Kazman, G. Abowd, L. Bass and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, Vol. 13, No. 6, pp. 47-55, November 1996.

[25] H. A. Müller, L. O'Brien, M. Klein and B. Wood, "Autonomic Computing," *Technical Note CMU/SEI-2006-TN-006*, April 2006. http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn006.pdf

[26] L. Bass and B. E. John, "Achieving Usability Through Software Architectural Styles," *Proceedings ACM Conference on Human Factors in Computing Systems (CHI 2000)*, pp. 171-172, April 2000.

[27] F. Losavio, L. Chirinos and M. Pérez, "Attribute-Based Techniques to Evaluate Architectural Styles. Case Study for Interactive Systems," *Acta Científica Venezolana*, pp. 130–138, 2002.

[28] F. Losavio, L. Chirinos and M.A. Pérez, "Quality Models to Design Software Architectures," *Proceedings Technology of Object-Oriented Languages and Systems (TOOLS 2001)*, pp. 123-135, March 2001.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995.

[30] D.K. Pradhan (ed.), *Fault-Tolerant Computer System Design*, Prentice-Hall, 1996.

[31] N.J. Park, B. Jin, T. Feng, K.M. George and N. Park, "Reliability Modeling and Analysis of Self-Healing Massively Parallel Computing Systems," *US-Korea Conference (UKC 2004)*, Research Triangle Park, NC, USA, August, 2004.
http://www.ksea.org/ukc2004/en/Proceedings/03ICT/ICT72_okstate.park.pdf

[32] P. Bret, "Design for Testability," 2002.
http://www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf

[33] R. Kumar and P. Rao, "Model for Self-Managing Java Server," 2003.
http://www-128.ibm.com/developerworks/library/ac-alltimeserver/

[34] E. Alexander, "Architecture Differencing for Self Management," *Proceedings ACM SIGSOFT First Workshop on Self-Managed Systems (WOSS 2004)*, pp. 44-48, Newport Beach, CA, November 2004.

[35] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan and E. Brewer, "Failure Diagnosis Using Decision Trees," *Proceedings IEEE First International Conference on Autonomic Computing (ICAC 2004)*, pp. 36-43, New York, May 2004.

[36] A.R. Haydarlou, B.J. Overeinder and F.M.T. Brazier, "A Self-healing Approach for Object-oriented Applications," *Proceedings Sixteenth International Workshop on Database and Expert Systems Applications (DEXA 2005)*, pp. 191-195, Aug. 2005

[37] D.G. Benoit, "Automatic Diagnosis of Performance Problems in Database Management Systems," *Proceedings IEEE Second International Conference on Autonomic Computing (ICAC 2005)*, pp. 326-327, June 2005.

[38] R. Kazman and M. Klein, "Designing and Analyzing Software Architectures using ABASs," *Proceedings IEEE/ACM International Conference on Software Engineering (ICSE 2000)*, p. 820, Ireland, June 2000.

[39] P. Kaminski, H. A. Müller, and M. Litoiu, "A Design for Adaptive Web Service Evolution," *Proceedings IEEE/ACM ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2006)*, pp. 86-92, China, May 2006.

[40] A.B. Brown, J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum and M.P. Yost, "Benchmarking Autonomic Capabilities: Promises and Pitfalls," *Proceedings IEEE International Conference on Autonomic Computing (ICAC 2004)*, pp. 266-267, May 2004.

[41] A.B. Brown and C. Redlin, "Measuring the Effectiveness of Self-Healing Autonomic Systems," *Proceedings IEEE Second International Conference on Autonomic Computing (ICAC 2005)*, pp. 328-329, June 2005.

[42] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, "Towards Requirements-driven Autonomic Systems Design," *Proceedings Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005)*, pp. 1-7 , St. Louis, Missouri, USA, May 2005.

[43] Murch, R. *Autonomic Computing.* Upper Saddle River, NJ: Prentice Hall, 2004 (ISBN 0-13-144025X).

[44] K. Wong. *Reverse Engineering Notebook*, PhD Thesis, Department of Computer Science, University of Victoria, October 1999.

[45] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess and J.O. Kephart, "An Architectural Approach to Autonomic Computing," *Proceedings IEEE International Conference on Autonomic Computing (ICAC 2004),* pp. 2-9, May 2004.