View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

A Quality-Driven Approach to Enable Decision-Making in Self-Adaptive Software

by

Mazeiar Salehie

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2009

© Mazeiar Salehie 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Mazeiar Salehie

Abstract

Self-adaptive software systems are increasingly in demand. The driving forces are changes in the software "self" and "context", particularly in distributed and pervasive applications. These systems provide self-* properties in order to keep requirements satisfied in different situations. Engineering self-adaptive software normally involves building the adaptable software and the adaptation manager. This PhD thesis focuses on the latter, especially on the design and implementation of the deciding process in an adaptation manager.

For this purpose, a *Quality-driven Framework for Engineering an Adaptation Manager (QFeam)* is proposed, in which quality requirements play a key role as adaptation goals. Two major phases of QFeam are building the runtime adaptation model and designing the adaptation mechanism. The modeling phase investigates eliciting and specifying key entities of the adaptation problem space including goals, attributes, and actions. Three composition patterns are discussed to link these entities to build the adaptation model, namely: *goal-centric*, *attribute-action-coupling*, and *hybrid* patterns. In the second phase, the adaptation mechanism is designed according to the adopted pattern in the model. Therefore, three categories of mechanisms are discussed, in which the novel *goal-ensemble* mechanism is introduced. A concrete model and mechanism, the *Goal-Attribute-Action Model (GAAM)*, is proposed based on the goal-centric pattern and the goal-ensemble mechanism. GAAM is implemented based on the StarMX framework for Java-based systems.

Several considerations are taken into account in QFeam: i) the separation of adaptation knowledge from application knowledge, ii) highlighting the role of adaptation goals, and iii) modularity and reusability. Among these, emphasizing goals is the tenet of QFeam, especially in order to address the challenge of addressing several self- * properties in the adaptation manager. Furthermore, QFeam aims at embedding a model in the adaptation manager, particularly in the goal-centric and hybrid patterns.

The proposed framework focuses on mission-critical systems including enterprise and service-oriented applications. Several empirical studies were conducted to put QFeam into practice, and also evaluate GAAM in comparison with other adaptation models and mechanisms. Three case studies were selected for this purpose: the TPC-W bookstore application, a news application, and the CC2 VoIP call controller. Several research questions were set for each case study, and findings indicate that the goal-ensemble mechanism and GAAM can outperform or work as well as a common rule-based approach. The notable difference is that the effort of building an adaptation manager based on a goal-centric pattern is less than building it using an attribute-action-coupling pattern. Moreover, representing goals explicitly leads to better scalability and understandability of the adaptation manager. Overall, the experience of working on these three systems show that QFeam improves the design and development process of the adaptation manager, particularly by highlighting the role of adaptation goals.

Acknowledgements

I would like to thank all the people who made this possible. This thesis would not have been possible unless the help and support of my supervisor, my thesis committee members, and my colleagues in the STAR lab. During my PhD research I learned many new things, and without the people surrounding me I could not enjoy from this period of my life.

First and foremost, I gratefully acknowledge my supervisor, *Professor Ladan Tahvildari*. She made available her support and aid in a number of ways. She always does care about her students, and I had this opportunity to discuss the obstacles hindered me in my study and research openly with her. She did her best to aid me to proceed and I really appreciate her effort. Moreover, she always encouraged and supported me to contribute to the software engineering community.

I would like to show my gratitude to *Professor Kostas Kontogiannis* for his helpful comments and guidelines during this research. He pointed out useful notes about my work in my comprehensive exam, PhD seminar, my defense, and other occasions.

My special thanks go to *Professor Hausi Muller* to accept being my external examiner. His subtle questions and remarks helped me improve this thesis and clarify fuzzy points. I also want to mention that his research contributions and the events he organized, particularly the SEAMS workshop, had tremendous impacts on my PhD research.

I would like to thank *Professor Mohamed Kamel* for his insightful comments and advices in my comprehensive exam. I would also like to thank *Professor Otman Basir* for accepting to be my PhD defense examiner replacement, in spite of his busy schedule. His questions and comments during the defense were quite useful for me to revise some parts of this dissertation. I would also like to thank *Dr. Paulo Alencar* for spending time on reading my thesis carefully and giving many comments on the details. My special thanks go to *Professor Marc Kilgour*. I learned a lot from his course on multi-criteria decision analysis which was the basis of some ideas in this dissertation.

I am indebted to my colleagues for supporting me. Among them, *Sen Li*, *Mehdi Amoui*, and *Reza Asadollahi* are notable. We worked collaboratively in the STAR lab, and we had many meetings to brainstorm and review our work, especially for the case studies.

This work was financially supported by Ontario Graduate Scholarship (OGS), Natural Sciences and Engineering Council of Canada (NSERC), Ministry of Research & Innovation (MRI), University of Waterloo President's Graduate Scholarship (PGS), and Richard & Elizabeth Madter Graduate Scholarship.

I owe my deepest gratitude to my wife, *Sepinood*, who tolerated me during my PhD career, and was always supportive and kind to me. Without her support I could have not finished this job. Also, I owe an immense debt of gratitude to my mother who gave me hope and strength to follow my passions up to this point.

Dedication

To my beloved mother
who taught me to follow my dreams
without disappointment and fatigue

To my beloved wife
who taught me to be hopeful and courageous
without fear and frustration

In memory of my dearest father
who taught me to be patient and to endure difficulties
without complaining and giving up

Contents

Author's Declaration **ii**

Abstract **iii**

Acknowledgments **iv**

Dedication **v**

Table of Contents **x**

List of Tables **xii**

List of Figures **xiv**

1 Introduction **1**

1.1 Motivating Example 1

1.2 Problem Description 2

1.3 Thesis Contributions 4

1.4 Thesis Organization 5

2 Background and Related Work **6**

2.1 Principles 7

2.1.1 Self-Adaptive Software Definition 8

2.1.2 Self-* Properties 9

2.1.3 Adaptation Requirements Elicitation 11

2.1.4 Adaptation Loop 13
2.2 A Taxonomy of Self-Adaptation 16
2.2.1 Object to Adapt 17
2.2.2 Realization Issues 20
2.2.3 Temporal Characteristics 22
2.2.4 Interaction Concerns 23
2.3 Supporting Disciplines 24
2.3.1 Supporting Software Engineering Concepts 24
2.3.2 Supporting Artificial Intelligence Concepts 26
2.3.3 Supporting Control Theory/Engineering Concepts 28
2.3.4 Supporting Network and Distributed Systems Concepts 29
2.4 Research Projects 32
2.5 Research Challenges 38
2.5.1 General Challenges 38
2.5.2 Challenges Addressed by this Thesis 39
2.6 Summary 39

3 QFeam: A Quality-Driven Framework for Engineering an Adaptation Manager 41
3.1 Main Role of Quality Requirements 42
3.2 (Re-)Engineering Adaptable Software 43
3.3 Engineering an Adaptation Manager with QFeam 44
3.3.1 Building Run-Time Adaptation Model 46
3.3.2 Adaptation Mechanism Design 47
3.4 Design Considerations 49
3.4.1 Separation of Concerns 49
3.4.2 Goal-driven Adaptation 49
3.4.3 Modularity and Reusability 50
3.5 Design Metaphors 50
3.5.1 Behavior-Based Robotics 51
3.5.2 Collective Decision-making and Mechanism Design 52
3.6 Summary 53

4 Building Run-Time Adaptation Model **54**
4.1 Notations 54
4.2 Modeling Process 55
4.3 Adaptation Requirements Analysis 57
4.3.1 Eliciting and Analyzing Goals 58
4.3.2 Eliciting and Analyzing Attributes 59
4.3.3 Eliciting and Analyzing Actions 61
4.3.4 Requirements Evaluation 63
4.4 Modeling Adaptation Problem Space Entities 64
4.4.1 Modeling Goal Space Entities 65
4.4.2 Modeling Attribute Space Entities 68
4.4.3 Modeling Action Space Entities 70
4.5 Composing the Adaptation Model 74
4.6 Summary 77

5 Adaptation Mechanism Design and Evaluation **78**
5.1 Adaptation Mechanisms 80
5.1.1 Goal-centric Mechanisms 80
5.1.2 Attribute-Action-Coupling Mechanisms 84
5.1.3 Hybrid Mechanisms 85
5.2 Goal-Attribute-Action Model (GAAM): A Concrete Adaptation Model . . 87
5.2.1 Adaptation Model in GAAM 87
5.2.2 Goal Preferences in GAAM 89
5.2.3 Adaptation Mechanism in GAAM 91
5.3 Evaluating Adaptation Mechanisms 93
5.3.1 Effectiveness 94
5.3.2 Quality of Adaptation 95
5.4 Summary 96

6 Implementation and Empirical Studies **98**
6.1 StarMX Framework for Java-based Systems 99
6.1.1 StarMX Architecture 99
6.1.2 Run-Time Behavior 101
6.1.3 GAAM Implementation 102
6.2 Case Study 1 (CS1): TPC-W Bookstore Application 105
6.2.1 Building Adaptable Software 105
6.2.2 Building the Adaptation Manager 107
6.2.3 Design of Experiments 108
6.2.4 Obtained Results 110
6.2.5 Lessons Learned 113
6.3 Case Study 2 (CS2): News Web Application 114
6.3.1 Building Adaptable Software 114
6.3.2 Building the Adaptation Manager 116
6.3.3 Design of Experiments 119
6.3.4 Obtained Results 121
6.3.5 Lessons Learned 124
6.4 Case Study 3 (CS3): Service-oriented VOIP Call Controller- CC2 125
6.4.1 CC2 VoIP Call Controller 126
6.4.2 Building Adaptable Software 129
6.4.3 Building the Adaptation Manager 131
6.4.4 Design of Experiment 135
6.4.5 Testbed 139
6.4.6 Obtained Results 140
6.4.7 Lessons Learned 143
6.5 Summary 144

7 Conclusions and Future Work **146**
7.1 Thesis Summary 146
7.2 Future Work 149

APPENDICES **151**

A CC2 case study details **152**

A.1 Service Level Agreement (SLA) 152
A.2 Attributes for $CC2_{original}$ 152
A.3 Attributes for $CC2_{modified}$ 153
A.4 Actions for $CC2_{original}$ 154
A.5 Goals for $CC2_{original}$ 154
A.6 Actions for $CC2_{modified}$ 155
A.7 Goals, Weights and Preferences for $CC2_{modified}$ 156
A.8 CC2 experiments detail results 157

References **160**

List of Tables

2.1 Different techniques for realizing sensors and effectors 15
2.2 Some weak and strong adaptation actions in self-adaptive software 19
2.3 Selected projects in the area of self-adaptive software 33
2.4 Comparing projects in terms of self-* properties 34
2.5 Comparing research projects in terms of adaptation processes 35
2.6 Comparing research projects in terms of the taxonomy facets 37

4.1 A sample adaptation activity hierarchy 74

6.1 Evaluating StarMX overload using CC2 system 102
6.2 Evaluation function $E(.)$ in conducted experiments 110
6.3 Summary of ANOVA for high and medium loads separately in TPC-W . . 111
6.4 Summary of ANOVA (4 treatments, 2 blocks, and 3 replications in TPC-W) 112
6.5 Contrasts between T_3 and other treatments - TPC-W case study 113
6.6 Attributes in news application 116
6.7 Data type/quality values for at_1 in news application 117
6.8 Goals in GAAM - News application 118
6.9 Ordinal preferences in the news application 119
6.10 Cardinal preferences in the news application 119
6.11 Friedman ANOVA test for news application 121
6.12 Dunnett test for pairwise comparison of treatments in news application . . 122
6.13 Inter-arrival time for each service in $Exp1$ - CC2 system 138
6.14 Inter-arrival time distribution for each service in $Exp2$- CC2 system 138

6.15 Inter-block ANOVA for CC2 system 140
6.16 Dunnett test for pairwise comparison of treatments - CC2 system 141
6.17 ANOVA test for utility values based on GLM F-test - CC2 system 142
6.18 Dunnett t-test for pairwise comparison of treatments - CC2 case study . . 142
6.19 Experiments for extreme heavy workload (B4)- CC2 system 143

A.1 CC2 Exp1 B1 results (light workload) 157
A.2 CC2 Exp1 B3 results (heavy workload) 158
A.3 CC2 Exp2 B1 results (light workload) 158
A.4 CC2 Exp2 B2 results (medium workload) 158
A.5 CC2 Exp2 B3 results (heavy workload) 159

List of Figures

1.1 Thesis organization 5

2.1 Hierarchy of the self-* properties 9
2.2 Four adaptation processes in self-adaptive software 14
2.3 Taxonomy of self-adaptation 17
2.4 Internal and external approaches for building self-adaptive Software 20
2.5 Classifying self-adaptive software challenges 38

3.1 The big picture of engineering self-adaptive software 42
3.2 Quality-driven framework for engineering an adaptation manager (QFeam) 45
3.3 Adaptation conceptual model 47
3.4 Different approaches of using models in engineering adaptation manager . . 48

4.1 QFeam modeling process 56
4.2 A partial goal hierarchy for the news application 67
4.3 Goal meta-model in quality-driven adaptation framework 68
4.4 Attribute meta-model in quality-driven adaptation framework 69
4.5 Action meta-model in quality-driven adaptation framework 71
4.6 Goal-centric composition pattern 75
4.7 Attribute-action-coupling composition pattern 76
4.8 Hybrid composition pattern 76

5.1 Goal-ensemble mechanism 82
5.2 A general schema of attribute-action-coupling mechanism based on rules . 84

5.3 General schema of a hybrid mechanism using collective decision-making . . 86
5.4 Representing GAAM as a graph 87
5.5 GAAM adaptation mechanism 92

6.1 StarMX high level architecture 99
6.2 StarMX execution chain architecture 100
6.3 Implementing GAAM with StarMX processes 103
6.4 Goal hierarchy in TPC-W 106
6.5 Adaptation action hierarchy in TPC-W 107
6.6 Experimental model of a multi-tier news application 115
6.7 Goal hierarchy of the experimental model in news application 117
6.8 Evaluation function of treatment in Exp1-3 - News application 122
6.9 Evaluation function and loss ratio in Exp4 - News application 123
6.10 JBoss and JSLEE 126
6.11 SLEE services provided by CC2 system 127
6.12 A typical VoIP communication using SIP and RTP 129
6.13 Experiment setting for CC2 system 130
6.14 Goal hierarchy for CC2 system 132
6.15 Capacity of system in terms of response time and throughput graph 136
6.16 Setup environment for evaluating self-adaptive CC2 139

Chapter 1

Introduction

"Adapt or perish, now as ever, is nature's inexorable imperative."
H. G. Wells

Self-adaptive software systems are demanding in various domains nowadays. This thesis aims at engineering such systems, particularly mission-critical applications. In this way, the emphasis is on decision-making by capturing the adaptation requirements and designing a run-time mechanism for adaptation [184].

Engineering self-adaptive software is still an ad hoc process. Requirements engineering, design, development, and testing such systems are different from conventional software systems, and introduce new challenges. As elaborated in Chapter 3, engineering self-adaptive software includes two major phases of engineering *adaptable software* and *adaptation manager*. The former is building or re-engineering an application to be adaptable by exposing necessary sensors and effectors. The latter consists of building an adaptation manager from scratch or from reusable components. This thesis focuses on the second phase, even though the first phase is addressed partially, especially in the empirical studies.

1.1 Motivating Example

One of the specific applications that has motivated this research is a news web application. The example of managing a news application was initially introduced by Cheng *et al.* [41]. Later, in this PhD research, we figured out that the run-time adaptation of a news application, particularly by considering the problems encountered by news web sites in the US after 9/11, provides an interesting case study [185].

The news web sites' usage soared on 9/11, and continued to be high throughout the week. A Los Angeles Times report offers a few numbers[1]: "MSNBC saw a tenfold increase in traffic, with as many as 400,000 hits at any point. CNN.com surged to 162.4 million page views in 24 hours from a 14 million average." By the afternoon of the next day, some web sites reported even heavier congestion than the day before. In this situation, the technical staff of the CNN web site changed the pages in order to remove all of the extra graphics and links, and focus on the breaking news.

After the 9/11 event, administrators and network managers tried to manage the aftereffects in news applications mostly by tuning the parameters, and by applying server or network-level actions. But application-level actions were missing, which were applied manually in the CNN case. Switching between service levels were among these interesting actions. This example is one of the many mission-critical applications (e.g., e-commerce, banking, and communication), that need automation and run-time mechanisms for adaptation. This example is used in this thesis to clarify the abstract discussions.

1.2 Problem Description

Engineering an adaptation manager aims at realizing several key processes, such as *detecting* violations of the requirements and *deciding* the way software needs to be adapted. The focused problem in this thesis is how to capture the essential adaptation requirements, and how to reason based on them to figure out why, how, when, where and what should be adapted. In short, *given a software system, the objective is to engineer an external adaptation manager, which providing domain attributes, adaptation goals and actions, can decide which action is appropriate to be taken.* To this end, the problem is broken down into the following main facets:

- *Modeling problem space key entities*: An important point in engineering an adaptation manager is to capture key entities in the problem space in order to specify adaptation requirements. Because adaptation requirements are strongly linked to software quality factors, quality goals play an important role in the target model. Addressing multiple self-* properties is also significant in this facet, as emphasized by an early work during this research [182].

- *Adaptation mechanism design*: A run-time mechanism is required to manage adaptation, particularly by detecting violating requirements and deciding how to act accordingly. Such a mechanism traces the adaptation requirements and selects an action plan based on the determined adaptation actions. One of the less addressed areas in this facet is the goal-driven adaptation.

[1]http://articles.latimes.com/2001/sep/13/news/mn-45327?pg=1

- *Evaluating an adaptation manager*: One of the challenges in front of building self-adaptive software systems is how to evaluate an adaptation manager and its constituent mechanism. The evaluation considers both the effectiveness and the quality of adaptation.

To address the problem and its facets, the thesis goal is set out to provide a *Quality-driven Framework for Engineering an Adaptation Manger* (QFeam). Since establishing a robust framework for this purpose is the long-term objective and vision of this research, the following limitations are designated to narrow down the scope of this thesis:

- *Focusing on deciding process*: The focus in QFeam is on the deciding process and enabling the adaptation manager to deal with problem space entities at run-time. During this research, it has been turned out that the deciding process can not be isolated completely from other processes, particularly the detecting process. Thus, this thesis partially also addresses the detecting process. But, the monitoring and acting processes are out of the scope of this thesis, though they have been implemented for case studies.

- *Targeting mission-critical application domain*: Although there is no domain-dependent assumption in QFeam, because the experiences and empirical studies are entirely related to mission-critical applications, this domain is selected. For other domains, like safety-critical applications, time constraints should be highlighted in an adaptation model and the corresponding mechanism.

- *Concerning performance and availability quality factors*: Case studies played a significant role in shaping the proposed engineering framework, and due to the fact that in all of them performance and availability have been considered, it makes sense to limit the scope of this thesis by focusing on these quality factors. But, there is no explicit assumption about these quality factors, and extending the framework to other factors seems feasible. For instance, availability is a part of dependability and QFeam appears extensible to cover it as well.

- *Emphasizing application-level adaptation actions*: While there is no explicit assumption related to the adaptation actions, the experiences and empirical studies put stress on application-level actions. The main reason is that application-level adaptation still lacks extensive coverage comparing with adaptation at middleware or network layers.

In order to validate the proposed framework, a series of empirical studies should also be conducted.

1.3 Thesis Contributions

This thesis introduces an engineering framework, QFeam, for modeling adaptation problem space and designing a run-time mechanism relying on the built model. Regarding the target problem and its specified facets the following contributions are notable in this thesis:

- *Designing an adaptation model*: Three concept spaces are identified in a general adaptation problem domain including goals, attributes and actions. Metamodels of these spaces are discussed and some composition patterns are described for linking constituent entities in these three spaces. Several concerns are taken into account in the adaptation model: i) dealing with multiple self-* properties, ii) representing an explicit adaptation logic, and iii) having a run-time embedded model to be used by adaptation mechanism.

- *Adaptation mechanism design*: Several mechanisms are discussed regarding the corresponding composition patterns. A novel concrete model, Goal-Attribute-Action Model (GAAM), and a goal-driven mechanism is introduced based on one of the composition patterns [185]. GAAM is evaluated empirically using three case studies.

- *Evaluating self-adaptive software*: Evaluating the quality of adaptation and benchmarking self-adaptive software systems are still premature and challenging. This thesis discusses this area by benefiting goal-based and utility-based evaluation (See [178, 185]). It is not claimed that this problem has been solved completely, but the discussions tackle the problem considerably and findings provide thought-provoking ideas to pursue.

This research has also led to several valuable by-products, which are not not directly in the scope of the defined problem:

- *Providing a taxonomy of adaptation*: This research started with investigating principles of self-adaptive software and providing a taxonomy of adaptation. This leads to preparing a landscape of this area and spotting research challenges [186].

- *Providing three case studies for run-time adaptation*: Lack of benchmarks and case studies for investigating effectiveness and efficiency of adaptation is one of the main problems in this research and related work. Three case studies are introduced in this thesis that can be used by the research community for future research (see [178, 185, 179]). A remarkable amount of time has been spent for instrumenting these applications with sensors and effectors, and preparing the testbed.

The taxonomy is used to give a big picture of the self-adaptive software domain in the background part of the thesis. Two of the case studies are open source and available for downloading by other researchers.

1.4 Thesis Organization

Figure 1.1 shows the organization of the thesis. Chapter 2 briefly reviews the background concepts and some related work. Chapter 3 introduces QFeam by outlining its objectives and main themes. Two phases of building a run-time adaptation model and an adaptation mechanism are the key processes for this purpose. The former is discussed in Chapter 4, by focusing on three main concept spaces and the composition patterns to link them. Chapter 5 deals with the adaptation mechanism design and a concrete novel model, GAAM. A series of empirical studies are conducted to investigate different issues in engineering an adaptation manager. Chapter 6 reviews the outcomes of the experiments to answer several designated research questions. Chapter 7 summarizes the thesis, draws several conclusions, and suggests ideas for potential future work.

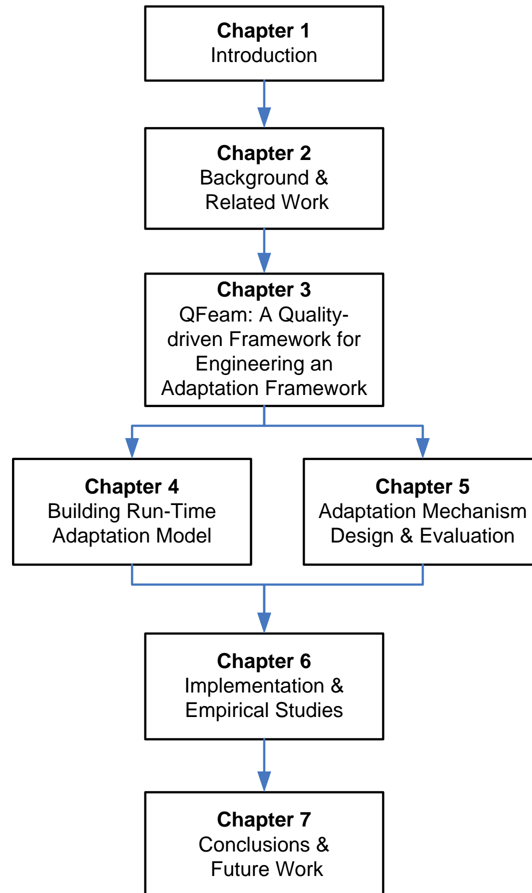

Figure 1.1: Thesis organization

Chapter 2

Background and Related Work

"A perfection of means, and confusion of aims, seems to be our main problem."
Albert Einstein

Scientists and engineers have made significant efforts to design and develop self-adaptive systems. These systems address adaptivity in various concerns including performance, security, and fault management [97, 107]. While self-adaptive systems are used in a number of different areas, this thesis focuses only on their application in the software domain, called *self-adaptive software*. Researchers in this area have proposed various solutions to incorporate adaptation mechanisms into software systems. In this way, a software application which would normally be implemented as an open-loop system, is converted to a closed-loop system using feedback. While adaptivity may be achieved through feed-forward mechanisms as well (e.g., through workload monitoring), the feedback loop takes into account a more holistic view of what happens inside the application and its environment.

Self-adaptive software aims at adjusting various artifacts or attributes in response to changes in the *self* and in the *context* of a software system. *Self* means the whole body of software, mostly implemented in several layers, while the *context* encompasses everything in the operating environment that affects the system properties and its behavior. Therefore, self-adaptive software is a closed-loop system with feedback from the *self* and the *context*.

A fundamental question is why we need self-adaptive software. The primary reason is the increasing cost of handling the complexity of software systems to achieve their goals [108]. Among these goals, some deal with complexity management and robustness in handling unexpected conditions (e.g., failure), changing priorities and policies governing the goals, and changing conditions (e.g., in the context of mobility). Traditionally, a significant part of the research on handling complexity and achieving quality goals has focused on software development and its internal quality attributes (as in ISO 9126-1

quality model [84]). However, in recent years, there has been an increasing demand to deal with these issues at run-time (or operation time). The primary causes of this trend include an increase in the heterogeneity level of software components, more frequent changes in the context/goals/requirements during run-time, and higher security needs. In fact, some of these causes are consequences of the higher demand for ubiquitous, pervasive, embedded, and mobile applications, mostly in the Internet and ad-hoc networks.

Self-adaptive software is expected to fulfill its requirements at run-time in response to changes. To achieve this goal, software should have certain characteristics, known as *self-* properties* [16, 200]. These properties provide some degree of variability, and consequently, help to overcome deviations from expected goals (e.g., reliability). Managing software at run-time is often costly and time-consuming. Therefore, an adaptation mechanism is expected to trace software changes and take appropriate actions at a reasonable cost and in a timely manner. This objective can be achieved through monitoring the software system (*self*) and its environment (*context*) to detect changes, make appropriate decisions, and act accordingly. Required changes in traditional software systems can stem from different categories of maintenance and evolution, as discussed in IEEE Standard for Software Maintenance [82]. This standard discusses corrective maintenance for fixing bugs, adaptive maintenance for adjusting the software according to changing environments, perfective maintenance for updating the software according to changing requirements, and finally, preventive maintenance for improving software maintainability. Although this standard does not explicitly refer to dynamic/run-time changes (dynamic evolution) in conjunction with these four categories, these changes are part of what is needed to deal with bugs and new or changing requirements. Dynamic and run-time changes are the basis for adaptation in self-adaptive software.

This chapter presents an overview of basic principles, properties, and background behind self-adaptive software. It proposes a taxonomy of adaptation [186], relying on the questions of *when*, *what*, *how*, and *where*. Using this taxonomy, a landscape is presented after reviewing a number of disciplines dealing with self-adaptive software, as well as some selected research projects.

2.1 Principles

This section presents a general review of the basic concepts in self-adaptive software. The objective is to provide a unified set of definitions, goals, and requirements that are used in the rest of the thesis.

2.1.1 Self-Adaptive Software Definition

Among several existing definitions for self-adaptive software, one is provided in a DARPA Broad Agency Announcement (BAA) [106]: "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible." A similar definition is given in [149]: "Self-adaptive software modifies its own behavior in response to changes in its operating environment. Here operating environment means anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation."

Prior to formalizing the concept of self-adaptive software, there has been a related point of view regarding the adaptive programming principle as an extension of object-oriented programming [118]: "A program should be designed so that the representation of an object can be changed within certain constraints without affecting the program at all." According to this view point, an adaptive program is considered as: "A generic process model parameterized by graph constraints which define compatible structural models (customizers) as parameters of the process model." This view on adaptation is similar to reflection and meta-programming techniques.

In another point of view, adaptation is mapped to evolution. Buckley *et al.* provide a taxonomy of evolution based on the object of change (*where*), system properties (*what*), temporal properties (*when*), and change support (*how*) [30]. Static and dynamic adaptation, related to the temporal dimension of this view, are mapped to compile-time evolution and load-time/run-time evolution, respectively. For this reason, dynamic adaptation is sometimes called *dynamic evolution*. In fact, self-adaptivity is linked to what Lehman has discussed on feedback and feedback control in the context of the software process for evolution [115]. According to this article, the essence of self-adaptive software is aligned with the laws of evolution (described by Lehman and his colleagues in FEAST/1 and FEAST/2 projects [116]).

Self-adaptive software systems are strongly related to other types of systems. The notable ones are autonomic and self-managing systems [97]. However, it is difficult, if not impossible, to draw a distinction between these different terminologies. Many researchers use the terms self-adaptive (not specifically self-adaptive software), autonomic computing, and self-managing interchangeably; for instance in the survey provided by Huebscher and McCann [76]. In comparing self-adaptive software to autonomic computing, there are some similarities and some differences. From one point of view, the self-adaptive software domain is more limited, while autonomic computing has emerged in a broader context. This means self-adaptive software has less coverage and falls under the umbrella of autonomic computing. From another point of view, we can consider a layered model for a software-intensive system that consists of: application(s), middleware, network, operating system,

hardware [136], and sub-layers of middleware [187]. According to this view, self-adaptive software primarily covers the application and the middleware layers, while its coverage fades in the layers below middleware. On the other hand, autonomic computing covers lower layers too, even down to the network and operating system (e.g., see reincarnation server in Minix 3.0 [204]). However, the concepts of these domains are strongly related and in many cases can be used interchangeably.

The key point in self-adaptive software is that its life-cycle should not be stopped after its development and initial set up. This cycle should be continued in an appropriate form after installation in order to evaluate the system and respond to changes at all time. Such a closed loop deals with different changes in user requirements, configuration, security, and a number of other issues.

2.1.2 Self-* Properties

Adaptation properties are often known as self-* properties (among others, see [16, 200]). One of the initial well-known set of self-* properties, introduced by IBM, include eight properties of self-configuring, self-healing, self-optimizing, self-protecting, openness, anticipatory, self-awareness and context-awareness [79]. This section discusses these properties, along with some other related ones, towards providing a unified hierarchical set, which will be used in the rest of the thesis.

A Hierarchical View

Figure 2.1 illustrates a hierarchy of self-* properties in three levels. In this hierarchy, self-adaptiveness and self-organizing are general properties, which are decomposed into major and primitive properties at two different levels.

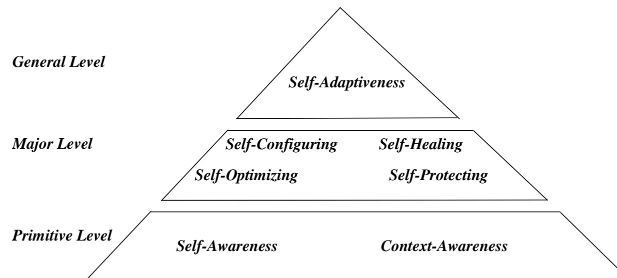

Figure 2.1: Hierarchy of the self-* properties

- **General Level**: This level contains global properties of self-adaptive software. A subset of these properties, which falls under the umbrella of self-adaptiveness [149], consists of *self-managing*, *self-governing*, *self-maintenance* [97], *self-control* [101], and *self-evaluating* [109]. Another subset at this level is *self-organizing* [87, 190], which emphasizes decentralization and emergent functionality(ies). A system with such a property typically consists of many interacting elements that are either absolutely unaware of or have only partial knowledge about the global system. The self-organizing property is bottom-up, in contrast to self-adaptiveness, which is typically top-down. Although most of the concepts in this thesis are applicable to the self-organizing property, this property is not the primary concern of this research. Noting the amount of research dealing with self-organizing systems, a separate section would be needed to adequately cover this emerging area.

- **Major Level**: The IBM autonomic computing initiative defines a set of four properties at this level [73]. This classification serves as the de facto standard in this domain. These properties have been defined in accordance to biological self-adaptation mechanisms [97]. For instance, the human body has similar properties in order to adapt itself to changes in its context (e.g., changing temperature in the environment) or self (e.g., an injury or failure in one of the internal organs). The following list further elaborates on the details.

  - *Self-configuring* is the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing or decomposing software entities.
  - *Self-healing*, which is linked to *self-diagnosing* [171] or *self-repairing* [48], is the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and take proper actions accordingly to prevent a failure. Self-diagnosing refers to diagnosing errors, faults and failures, while self-repairing focuses on recovering from them.
  - *Self-optimizing*, which is also called *self-tuning* or *Self-adjusting* [200], is the capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to-end response time, throughput, utilization, and workload are examples of important concerns related to this property.
  - *Self-protecting* is the capability of detecting security breaches and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating problems and taking actions to avoid them or to mitigate their effects.

- **Primitive Level**: Self-awareness, self-monitoring, self-situated, and context-awareness are the underlying primitive properties [73, 180]. Some other properties were also

mentioned in this level, such as openness [79] and anticipatory [152], which are optional. The following list further elaborates on the details.

- *Self-Awareness* [72] means that the system is aware of its *self* states and behaviors. This property is based on self-monitoring which reflects what is monitored.
- *Context-Awareness* [152] means that the system is aware of its *context*, which is its operational environment.

Relationships with Quality Factors

There is a belief that self-* properties are related to software quality factors. Salehie and Tahvildari discuss the potential links between these properties and quality factors [180]. The links can help us define and understand self-* properties better, and to utilize the existing body of knowledge on quality factors, metrics, and requirements, in developing and operating self-adaptive software. To demonstrate such relationships, it is better to analyze how a well-known set of quality factors defined in the ISO 9126-1 quality model [84] are linked to major and primitive self-* properties.

Self-configuring potentially impacts several quality factors, such as *maintainability*, *functionality*, *portability*, and *usability*. One may argue that self-configuring may be linked to reliability as well. This depends on the definition of reconfiguring. Assuming the definition given in the previous section (which is adopted by many researchers), we cannot associate all of the changes made to the system to for keep it reliable (e.g., fault recovery) with self-configuring. For self-healing, the main objective is to maximize the *availability*, *survivability*, *maintainability*, and *reliability* of the system [56].

Self-optimizing has a strong relationship with *efficiency*. Since minimizing response time is often one of the primary system requirements, it also impacts *functionality*. On the other hand, self-protecting has a strong relationship with *reliability*, and it can also be linked to *functionality*. Primitive properties may also impact quality factors, such as *maintainability*, *functionality*, and *portability*. Sterritt *et al.* also emphasize this view by discussing the relationship between dependability aspects (e.g., availability and reliability) and the self-* properties [199].

2.1.3 Adaptation Requirements Elicitation

One plausible way to capture the requirements of self-adaptive software is getting help from the *six honest serving men*[1]. These six questions are very important in eliciting adaptation

[1]Six questions *What*, *Where*, *Who*, *When*, *Why* and *How*, called 5W1H, from “Six Honest Men” poem of R. Kipling, Just so stories. Penguin Books, London, 1902.

requirements. Laddaga uses a similar idea to partially address these requirements [108]. The following set is a modified and completed version of Laddaga's questions to elicit the essential requirements of self-adaptive software.

- **Where**: This set of questions are concerned with where the need for change is. Which artifacts at which layer (e.g., middleware) and level of granularity need to be changed? For this purpose, it is required to collect information about the attributes of an adaptable software, the dependency between its components and layers, and probably information about its operational profile. Therefore, "where" questions set out to locate the problem that needs to be solved by adaptation.

- **When**: Temporal aspects of change are addressed by this set of questions. When does a change need to be applied, and when is it feasible to do so? Can it be applied at anytime the system requires, or are there constraints that limit some types of change? How often does the system need to be changed? Are the changes happening continuously, or do they occur only as needed? Is it enough to perform adaptation actions reactively, or do we need to predict some changes and act proactively?

- **What**: This set of questions identifies what attributes or artifacts of the system can be changed through adaptation actions, and what needs to be changed in each situation. These can vary from parameters and methods to components, architecture style, and system resources. It is also important to identify the alternatives available for the actions and the range of change for attributes (e.g., parameters). Moreover, it is essential to determine what events and attributes have to be monitored to follow-up on the changes, and what resources are essential for adaptation actions?

- **Why**: This set of questions deals with the motivations for building a self-adaptive software application. As discussed before, these questions are concerned with the objectives addressed by the system (e.g., robustness). If a goal-oriented requirements engineering approach is adopted to elicit the requirements, this set of questions identifies the goals of the self-adaptive software system.

- **Who**: This set of questions addresses the level of automation and human involvement in self-adaptive software. With respect to automation, it is expected that there will be minimum human intervention, whereas an effective interaction with system owners and managers is required to build trust and transfer policies (e.g. business policies). This issue will be discussed further in the taxonomy of adaptation (cf. Section 2.2).

- **How**: One of the important requirements for adaptation is to determine how the adaptable artifacts can be changed and which adaptation action(s) can be appropriate to be applied in a given condition? This includes how the order of changes, their costs and aftereffects are taken into account for deciding the next action/plan.

In order to realize self-adaptive software, the above questions need to be answered in two phases: a) the *developing phase*, which deals with developing and building self-adaptive software either from scratch or by re-engineering a legacy system, and b) the *operating phase*, which manages the operational concerns to properly respond to changes in the *self/context* of a software application. In the developing phase, designers elicit the requirements based on the above questions in order to build adaptable software as well as to set up mechanisms and alternatives to be used in the operating phase. In the operating phase, the system requires to adapt itself based on the above questions. In fact, the questions in this phase are in general ask about "where" the source of need for a change is, what needs be changed, and when and how it is better to be changed. the answer to these questions in the operating phase depends on the approach and type of adaptation chosen in the developing phase. Some of these questions may be answered by administrators and managers through policies, and the rest should be determined by the system itself.

The distinction between the what and where questions is notable. "Where" addresses which part of the system caused the problem (e.g., deviation from quality goals), while "what" refers to the attributes and artifacts that need to be changed to solve the problem. For example, in a multi-tier enterprise application, it is essential to know which part caused performance degradation (e.g., the database tier due to lack of resources) and after that, what needs to be changed (e.g., changing the service level in the web tier). Sometimes, the entity that is the source of change is also the entity that needs to be changed (e.g., component swapping). Therefore, although these questions are related, they address different aspects of adaptation.

2.1.4 Adaptation Loop

As explained earlier, self-adaptive software embodies a closed-loop mechanism. This loop, called the *adaptation loop*, consists of several processes, as well as sensors and effectors, as depicted in Figure 2.2. This loop is called the MAPE-K loop in the context of autonomic computing, and includes the Monitoring, Analyzing, Planning and Executing functions, with the addition of a shared Knowledge-base [97]. Dobson *et al.* also represent the similar loop as autonomic control loop in the context of autonomic communication, including collect, analyze, decide and act [50]. Oreizy *et al.* refer to this loop as adaptation management, which is composed of several processes for enacting changes and collecting observations, evaluating and monitoring observations, planning changes, and deploying change descriptions [149]. More details on the processes, sensors, and effectors of Figure 2.2 are provided in the following sections.

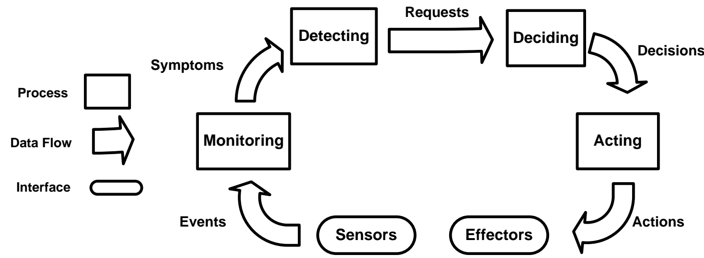

Figure 2.2: Four adaptation processes in self-adaptive software

Adaptation Processes

The adaptation processes, which exist in the operating phase can be summarized as follows:

- The **monitoring process** is responsible for collecting and correlating data from sensors and converting them to behavioral patterns and symptoms. This process partly addresses the *where*, *when*, and *what* questions in the operating phase. The process can be realized through event correlation, or simply threshold checking, as well as other methods.

- The **detecting process** is responsible for analyzing the symptoms provided by the monitoring process and the history of the system, in order to detect *when* a change (response) is required. It also helps to identify *where* the source of a transition to a new state (deviation from desired states or goals) is.

- The **deciding process** determines *what* needs to be changed, and *how* to change it to achieve the best outcome. This relies on certain criteria to compare different ways of applying the change, for instance by different courses of action.

- The **acting process** is responsible for applying the actions determined by the deciding process. This includes managing non-primitive actions through predefined workflows, or mapping actions to what is provided by effectors and their underlying dynamic adaptation techniques. This process relates to the questions of *how*, *what*, and *when* to change.

Sensors and Effectors

Sensors monitor software entities to generate a collection of data reflecting the state of the system, while effectors rely on *in vivo* mechanisms to apply changes. In fact, effectors

realize adaptation actions. Sensors and effectors are essential parts of a self-adaptive software system. Indeed, the first step in realizing self-adaptive software is instrumenting sensors and effectors to build the *adaptable software*. Building adaptable software can be accomplished in an engineering or re-engineering manner [143]. For example, Parekh *et al.* discuss adding sensors (probes) into legacy systems in order to retrofit the self- * properties [153].

Table 2.1 lists the most common set of sensors and effectors in self-adaptive software. Logging is likely to be the simplest technique for capturing information from software. The logs need to be filtered, processed, and analyzed to mine significant information. The IBM Generic Log Adapter (GLA) and the Log Trace Analyzer (LTA) [78] are examples of tools for this purpose.

Table 2.1: Different techniques for realizing sensors and effectors

Entity	**Technique**	**Example**
Sensors	Logging	GLA (Generic Log Adapter), LTA (Log Trace Analyzer) [78]
	Monitoring & events information models	CIM (Common Information Model) [44], CBE (Common Base Events) [78]
	Management protocols and standards	Simple Network Management Protocol[196] , Web-Based Enterprise Management[215] , Application Response Measurement[10], Siena [37]
	Profiling	JVMTI (JVM Tool Interface) [90]
	Management frameworks	JMX (Java Management eXtension) [89]
	Aspect-oriented programming	BtM (Build to Manage) [80], JRat (Java Run-time Analysis Toolkit) [193]
	Signal monitoring	Heartbeat and pulse monitoring [72]
Effectors	Design patterns	Wrapper (Adapter), Proxy, Strategy Pattern [55]
	Architectural patterns	Microkernel pattern, Reflection pattern, Interception pattern [32, 5]
	Autonomic patterns	Goal-driven self-assembly, self-healing clusters and utility-function driven resource allocation [99]
	Middleware-based effectors	Integrated middleware, Middleware interception [165], Virtual component pattern [188]
	Metaobject protocol	TRAP/J [176]
	Dynamic aspect weaving	JAC [159], TRAP/J [176]
	Function pointers	Callback in CASA [142]

Sensing and monitoring techniques from other areas can also be used. For instance, some of the protocols, standards, and formats that have been utilized are: CBE (Common Base Events) [78], WBEM (Web-Based Enterprise Management) [215] (containing

CIM - Common Information Model [44]), and SIENA (Scalable Internet Event Notification Architectures) [37]. Another noteworthy standard for sensing is ARM (Application Response Measurement) [10], which enables developers to create a comprehensive end-to-end management system with the capability of measuring the application's availability, performance, usage, and end-to-end response time. The ideas behind SNMP (Simple Network Management Protocol) [196] for network and distributed systems are also applicable to self-adaptive software.

Profiling tools and techniques can also help in defining desirable sensors. The Java environment provides JVMTI (Java Virtual Machine Tool Interface) for this purpose [90]. Software management frameworks, such as JMX (Java Management eXtensions) [89] provide powerful facilities for both sensing and effecting. Reflection can be also used for monitoring [47]. Another notable idea along this line is pulse monitoring (reflex signal) [72] adopted from Grid Computing, which is an extension of the heartbeat monitoring process. This technique encodes the status of the monitored entity.

Some of the effectors are based on a set of design patterns that allow the software system to change some artifacts during run-time. For instance, wrapper (adapter), proxy, and strategy are well-known design patterns [55] for this purpose. Landauer *et al.* utilize the wrapping idea at the architecture level of adaptive systems [112]. Moreover, microkernel, reflection, and interception are architectural patterns suitable for enabling adaptability in a software system [32, 5]. Furthermore, Kephart mentions several design patterns, namely goal-driven self-assembly, self-healing clusters, and utility-function-driven resource allocation for self-configuring, self-healing, and self-optimizing [99], respectively. Babaoglu *et al.* also discuss design patterns from biology, such as plain diffusion and replication, which are applicable to distributed adaptable systems [15].

An important class of techniques for effectors is based on middleware. In these solutions, system developers realize effectors at the middleware layer by intercepting the software flow [165], or by using design patterns [188]. Other solutions have been proposed for implementing effectors using dynamic aspect weaving (e.g., in JAC [159]), metaobject protocol (e.g., in TRAP/J [176]), and the function pointers technique (e.g., in CASA [142] for realizing callback).

2.2 A Taxonomy of Self-Adaptation

Several works have already discussed different aspects of self-adaptation. Oreizy *et al.* discuss the spectrum of self-adaptivity, which generally starts from static approaches and then moves on to dynamic ones [149]. On the other hand, McKinley *et al.* focus more on the techniques and technologies in this domain [136]. This section unifies these classifications into a taxonomy, and also introduces new facets to fill in the gaps.

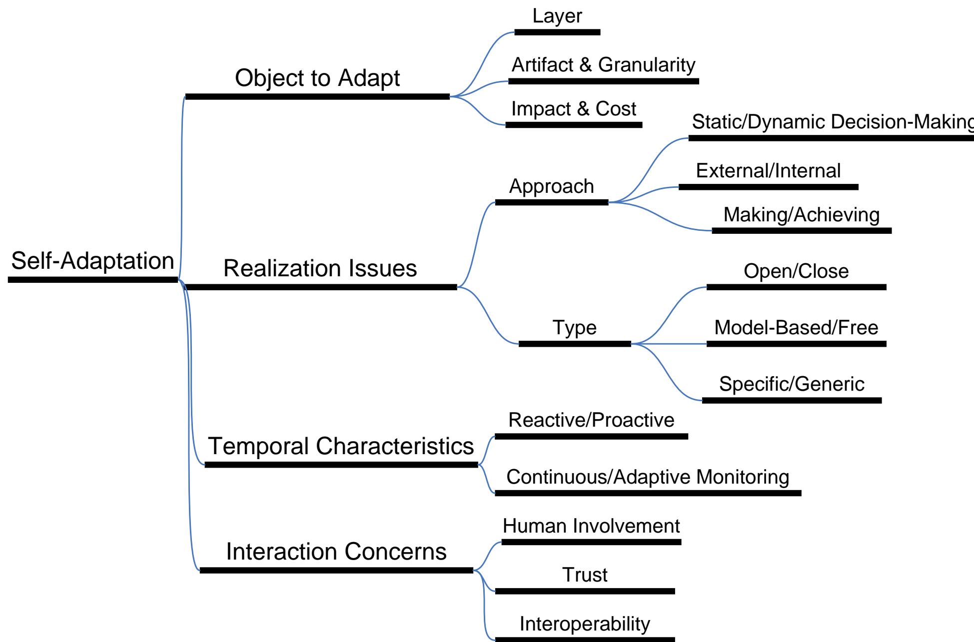

Figure 2.3: Taxonomy of self-adaptation

Figure 2.3 illustrates the hierarchy of the introduced taxonomy. The first level includes: "object to adapt", "realization issues", "temporal characteristics" and "interaction concerns". In organizing these facets, the requirements questions, introduced in Section 2.1.3, are considered. Although these facets cannot be mapped to the questions on a one-to-one basis, one or two questions in each facet are emphasized: "object to adapt" mainly addresses the "what" and "where" questions, while "realization issues" deal more with the "how" concerns, and "temporal characteristics" deal with issues related to the "when" aspect. Interaction concerns are more or less related to all four of where-when-what-how questions as well as to the "who" question.

2.2.1 Object to Adapt

This facet of the proposed taxonomy deals with *where* and *what* aspects of the change. In fact, both sets of what and where questions are covered in the developing and operating phases.

- *Layer*: Which layer of the software system (i.e., *where*) does not function as expected based on the requirements? Which layer of the system can be changed and needs to be changed? Adaptation actions can be applied to different layers. McKinley *et*

al. [136] define two levels of application and middleware for this purpose, in which middleware is decomposed into four sub-layers, as described in [187]. Application-layer adaptation differs from middleware-layer adaptation in a number of ways. For example, in application-layer adaptation, changes usually have direct impact on the user, and consequently, they may require the user's explicit approval and trust [113].

- *Artifact and Granularity*: Which artifact(s) and at which level of granularity can/needs to be changed? *What* artifact, attribute, or resource can/needs to be changed for this purpose? Adaptation can change the modules or the architecture and the way they are composed. An application can be decomposed into services, methods, objects, components, aspects, and subsystems depending on the architecture and technology used in its implementation. Each of these entities, as well as their attributes and compositions, can be subject to change, and therefore, adaptation can be applied at fine or coarse levels of granularity.

- *Impact & Cost*: The impact describes the scope of aftereffects, while cost refers to the execution time, required resources, and complexity of adaptation actions. This facet is related to *what* the adaptation action will be applied to, and partly to *how* it will be applied. Based on both impact and cost factors, adaptation actions can be categorized into the *weak* and *strong* classes. Table 2.2 lists a typical set of adaptation actions in both of the weak and strong categories.

  Weak adaptation involves modifying parameters (parameter adaptation) or performing low-cost/limited-impact actions, whereas strong adaptation deals with high-cost/extensive-impact actions, such as replacing components with those that improve system quality [136]. Generally, weak adaptation includes changing parameters (e.g., bandwidth limit), using pre-defined static mechanisms (e.g., load-balancing), or other actions with local impact and low cost (e.g., compressing data). Strong adaptation may change, add, remove, or substitute system artifacts. Cost in this classification refers to how much time and resources an action would need. It also highly depends on whether the action's requirements (e.g., alternatives for switching) are ready, or will become ready at run time.

  Note that this classification is not the same as "artifact & granularity", even though one may argue that in general, higher levels (e.g., architecture) have higher cost and impact. Although in some cases there may be some correlation between granularity and cost/impact, this is not always the case. An example is the case of having a load-balancing action that routes requests through duplicate components or servers. Another noteworthy point is that strong actions are mostly composite, and may contain several weak/strong actions. For example, changing the architecture may require redeployment of some components and changing a few parameters.

Table 2.2: Some weak and strong adaptation actions in self-adaptive software

Type	Action	Description
Weak	Caching [149, 52]	Caching data, states, connections, objects or components in order to lower the response time, load of servers, or help decentralized management
	Changing data quality [142]	Changing data quality (i.e., lower resolution) to save bandwidth and increase speed
	Changing type of data [41, 142]	For instance, switching from video to image and even to text to save bandwidth and increase speed
	Compressing data [110]	Saving bandwidth by transceiving compressed data
	Tuning [93]	Adjusting parameters to meet some adaptation goals (i.e., buffer size and delay time)
	Load balancing [219, 36]	Fair division of load between system elements to achieve maximum utilization, throughput or minimum response time
	Changing aspects [162, 203]	Changing aspect of a component or object with another one with different quality
	Changing algorithm or method [149, 172]	Changing the algorithm/ method to meet self-* properties and run-time constraints
Strong	Replacement, addition & removal [136]	Replacing an entity (e.g., a component) by another one with the same interface but different quality (non-functional)
	Reorganizing or changing architecture [127, 150]	Changing organization/ architecture of the system (it may change the architectural style or design patterns of the system)
	Resource provisioning [8]	Provisioning additional resources at different levels (this action can be extended to adding/removing any resources, such as servers)
	Restarting / redeployment [35]	Restarting/ rebooting (macro- or micro-) or redeployment of system entities at different levels mainly due to faults/failures

2.2.2 Realization Issues

This facet of the proposed taxonomy deals with *how* the adaptation can/needs to be applied. These issue are categorized into *approach* and *type* classes, and are discussed further in the following.

Adaptation Approach

One significant facet of the taxonomy is the approach of incorporating adaptivity into the system. The following sub-facets can be identified:

- *Static/Dynamic Decision-Making*: This sub-facet specifically deals with how the deciding process can be constructed and modified. In the static option, the deciding process is hard-coded (e.g., as a decision tree) and its modification requires recompiling and redeploying the system or some of its components. In dynamic decision-making, policies [98], rules [121] or QoS definitions [123] are externally defined and managed, so that they can be changed during run-time to create a new behavior for both functional and non-functional software requirements.

- *External/Internal Adaptation:* From a different perspective, the adaptation can be divided into two categories with respect to the separation of the adaptation mechanism and application logic. These two categories, as depicted in Fig. 2.4, are as follows:

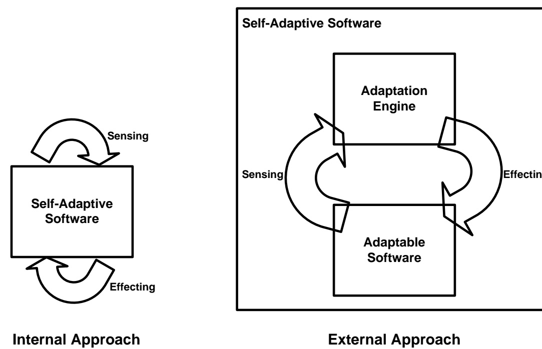

Figure 2.4: Internal and external approaches for building self-adaptive Software

  - *Internal* approaches intertwine application and the adaptation logic. This approach is based on programming language features, such as conditional expressions, parametrization, and exceptions [149, 54]. In this approach the whole

set of sensors, effectors, and adaptation processes are mixed with the application code, which often leads to poor scalability and maintainability. This approach can be useful for handling local adaptations (e.g., for exception handling). However, adaptation often needs global information about the system and correlating events happening in its self/context. Generally, this approach may be realized by extending existing programming languages or defining new adaptation languages.

- *External* approaches use an external *adaptation engine* (or adaptation manager) containing adaptation processes. As depicted in Fig. 2.4, using this approach, the self-adaptive software system consists of an adaptation engine and an *adaptable software*. The external engine implements the adaptation logic, mostly with the aid of middleware [102, 54], a policy engine [22], or other application-independent mechanisms. In a complex and distributed system, it is quite common to have multiple self-adaptive *element*s, each containing these two parts. In this case, the composition of elements in an appropriate architecture and an infrastructure for interoperability are essential.

The internal approach has some notable drawbacks. For instance, in this case the system will be costly to test and maintain/evolve, and it is often not scalable. On the other hand, a significant advantage of the external approach is the reusability of the adaptation engine, or some realized processes for various applications. This means that an adaptation engine can be customized and configured for different systems.

- *Making/Achieving Adaptation*: Generally speaking, self-adaptation can be introduced into software systems using two strategies [73]. The first strategy is to engineer self-adaptivity into the system in the developing phase. The second strategy is to achieve self-adaptivity through adaptive learning. Sterritt [198] calls these two approaches *making* and *achieving*. *Making* has an implied system and/or software engineering view to engineer adaptivity into the individual systems. *Achieving* has an implied artificial intelligence and adaptive learning view to achieve adaptive behavior. These two approaches do not necessarily contradict each other in the sense that their combination can be utilized as well.

Adaptation Type

Another important facet is the type of adaptation. It specifies whether the adaptation is open or closed to new alternatives, whether it is domain specific or generic, and whether it is model-based or model-free.

- *Close/Open Adaptation*: A close-adaptive system has only a fixed number of adaptive actions, and no new behaviors and alternatives can be introduced during run-time. On the other hand, in open adaptation, self-adaptive software can be extended, and consequently, new alternatives can be added, and even new adaptable entities can be introduced to the adaptation mechanism (e.g., through new joint-points for weaving aspects [159]).

- *Model-Based/Free Adaptation*: In model-free adaptation, the mechanism does not have a predefined model for the environment and the system itself. In fact, by knowing the requirements, goals, and alternatives, the adaptation mechanism adjusts the system. For example, Dowling uses model-free Reinforcement Learning (RL) in adaptation [51]. On the other hand, in model-based adaptation, the mechanism utilizes a model of the system and its context. This can be realized using different modeling approaches, such as a queueing model for self-optimizing [120], architectural models for self-healing [58], or domain-specific models in [94].

- *Specific/Generic Adaptation*: Some of the existing solutions address only specific domains/applications, such as a database (e.g., IBM SMART project [81]). However, generic solutions are also available, which can be configured by setting policies, alternatives, and adaptation processes for different domains (e.g., Accord [121]). This type addresses *where* and *what* concerns in addition to *how*, because the specific type only focuses on adaptation of artifacts or attributes of a particular part of the software system.

2.2.3 Temporal Characteristics

This facet deals with issues regarding *when* artifacts can/need to be changed. The following sub-facets can be identified:

- *Reactive/Proactive Adaptation:* This sub-facet captures the self-adaptive software anticipatory property [152]. In the reactive mode, the system responds when a change has already happened, while in the proactive mode, the system predicts when the change is going to occur. This issue impacts the detecting and the deciding processes.

- *Continuous/Adaptive Monitoring:* This sub-facet captures whether the monitoring process (and consequently sensing) is continually collecting and processing data versus being adaptive in the sense that it monitors a few selected features, and in the case of finding an anomaly, aims at collecting more data. This decision affects the cost of the monitoring and detection time.

2.2.4 Interaction Concerns

One cannot discuss the taxonomy without addressing the issue of interaction with other self-adaptive software systems/elements through interfaces. This facet consists of interacting with humans and/or other elements/systems. The facet is related to all four of the where-when-what-how questions as well as the "who" question. The following sub-facets can be identified:

- *Human Involvement*: As noted before, this facet is related to the question of "who" the agent of change is. In self-adaptive software, human involvement can be discussed from two perspectives. First, the extent to which the mechanism is automated, and second, how well it interacts with its users and administrators. For the former, we can use the maturity model proposed in autonomic computing [144]. The levels in this model include basic, managed, predictive, adaptive and autonomic. According to this view, human involvement is not desirable, therefore more automation is demanding. However, the second view addresses the quality of human interaction to either express their expectations and policies, or to observe what is happening in the system. According to this view, human involvement is essential and quite valuable for improving the manageability and trustworthiness of self-adaptive software. These issues have been addressed in some earlier research, such as [149] and [99]. For the rest of this thesis, the second meaning is used.

- *Trust*: Trust is a relationship of reliance, based on past experience or transparency of behavior. One view of trust is security, as highlighted by Dobson *et al.* in autonomic systems [50]. Another view, not orthogonal to the first one, is related to how much human or other systems can rely on self-adaptive software systems to accomplish their tasks. This view is linked first to assurance and dependability. Georgas *et al.* relate this issue to dependability as the "extent to which a system can be trusted to appropriately and correctly adapt" [60]. However, as McCann *et al.* point out, trust is not necessarily based on self-adaptive services or its quality [135, 76]. They discuss how trust can be built via revealing significant information about the system status and the visibility of adaptation processes. This explains why predictability can be considered as a major factor for placing trust upon self-adaptive software.

- *Interoperability Support*: Self-adaptive software often consists of elements, modules, and subsystems. Interoperability is always a concern in distributed complex systems for maintaining data and behavior integrity across all constituent elements and subsystems. In self-adaptive software, the elements need to be coordinated with each other to have the desired self-* properties and to fulfill the expected requirements. Global adaptation requirements will be met if elements and designated mechanisms

in different layers and platforms of a system (e.g., middleware and application) are interoperable .

2.3 Supporting Disciplines

This section briefly reviews how different disciplines are able to support and contribute to developing and operating self-adaptive software systems. The noteworthy point is that self-adaptive software is inherently interdisciplinary, and the combination of disciplines highly depends on the design metaphors adopted for building a specific self-adaptive software system. Laddaga enumerates three such design metaphors used by early researchers: coding an application as a dynamic planning system, coding an application as a control system [107], and coding a self-aware system [108]. These metaphors utilize ideas from artificial intelligence, and control theory. The following sections discuss several such disciplines - namely *software engineering*, *artificial intelligence*, *control theory/engineering*, and *network and distributed computing*. Although several other disciplines, such as *optimization theory*, can also be added to this list, due to space limitations, they are only partially discussed in connection with the other disciplines.

2.3.1 Supporting Software Engineering Concepts

Numerous research areas in software engineering are related to self-adaptive software. As discussed in Section 2.1.2, self-* properties can be related to quality factors. Consequently, the ideas developed in the context of *software quality* for realizing and measuring quality (including all -ilities) are potentially applicable to self-adaptive software.

Quality requirements are quite important in this thesis and their distinction from functional requirements needs to be clarified. Functional requirements specify what a system does, while quality requirements describe how well those functions are accomplished [24]. It is noteworthy that several terms are used for these quality requirements. “Non-Functional Requirements” (NFR) is one of them, which has ambiguity: “nonfunctional is a word that according to Merriam-Websters means something does not work. Some people have adopted the term ’ilities’ in recognition that many of these qualities end in the suffix ’ility’.” [24]

Quality requirements can be much more challenging to implement than functional ones for numerous reasons [24]: i) the stakeholders might believe that quality requirements are implicitly involved and they should be provided, ii) quality requirements often state trade-offs and conflicts that should be taken account by system designers, and iii) quality requirements are hard to measure and trace, and more importantly they may be met by different levels, not like functional requirements.

A few research efforts have aimed to address the link between quality requirements and run-time software adaptation, e.g., [185]. The important point is that self-* properties are mostly related to quality requirements, such as security and performance. In fact, fulfilling these requirements is the major trigger for change. However, functional requirements are also related to self-adaptation. An example is changing a component's behavior to an acceptable lower level of functionality, in order to improve its performance. These issues also bring *requirements engineering* into the picture. Several researchers have used NFR and quality requirements models, particularly goal models, in self-adaptive software; see for example [113] and [202].

From another point of view, we can look at requirements as user and system requirements. User requirements express the expectations of stakeholders from the system (i.e. system-to-be), while system requirements (or system specifications) express the desirable system property which hopefully meets the user requirements [128]. The notable point is that user requirements capture the stakeholders expectations in a set, whereas system requirements may specify alternative approaches to meet the user requirements. This is essentially emphasized in major RE books, such as [85]. What links this concept to engineering self-adaptive software is that those system requirements usually include different quality requirements. Therefore, by having alternative specs the system can have alternative artefacts to satisfy user requirements. This is similar to the idea of Software Product Line (SPL), but in contrast, it is to be used at run-time. Actually, this idea has recently appeared in the software engineering community as Dynamic SPL (DSPL) [70]. The idea is to have a variability model similar to SPL, but managing the selection and swapping between variants at run-time.

Another significant concept is using models in self-adaptive software, especially at run-time. Coupling software with its specification and formal model can allow monitoring correctness and many other metrics with respect to the specified requirements and self-* properties [158]. Model-driven development and formal methods provide various ways to model software systems and to utilize such models to reason about their behaviors. Accordingly, it is possible to rely on these approaches to model adaptable software in the adaptation processes. Moreover, formal methods can be used for validation and verification of self-adaptive software to ensure its correct functionality and to understand its behavior [111]. Due to various differences between traditional and self-adaptive software the existing models and methods developed for non-adaptive software systems are not directly applicable. This means that new approaches based on formal models, such as Model-Integrated Computing (MIC) [94], are required for this purpose. MIC has been applied successfully to some domain-specific embedded systems for managing run-time re-configuration [189].

Software Architecture models and languages, such as Architectural Description Languages (ADL), can certainly be helpful in software modeling and management, particularly

at run-time. Bradbury *et al.* survey several ADLs based on graphs, process algebras, and other formalisms for dynamic software architectures [27]. Garlan *et al.* use Acme ADL to describe the architecture of adaptable software and to detect violations from defined constraints [57]. Oreizy *et al.* point out that software architecture can also help in change management [149]. Another idea that can be useful is Attribute-Based Architecture Styles (ABAS) [100] as an extension of architectural styles. In fact, ABAS includes a quality-attribute specific model (e.g., performance), in order to provide a method of reasoning about an architecture design and the behavior of its interacting component types.

Component-Based Software Engineering (CBSE) can help the development of self-adaptive software in two ways. First, it is easier to design and implement an adaptable software relying on component models. Second, an adaptation engine needs to be modular and reusable, and CBSE can also be used in its development. Moreover, as pointed out in ACT [177], component models can be used in adaptive systems as a means of incorporating the underlying services for dynamic adaptation and adaptation management. Another related area, *Aspect-Oriented Programming (AOP)* and more specifically dynamic AOP, can also be used in realizing self-adaptive software. This facilitates encapsulating adaptation concerns in the form of aspects through dynamic run-time adaptation. It also helps in implementing fine-grained adaptation actions at a level lower than components [66]. For example, JAC (which is a dynamic AOP framework [159]) uses a wrapping chain that can dynamically change existing or new joint points. AOP can also be used for instrumenting sensors as in the IBM BtM (Build to Manage) tool [80].

Service Computing and *Service-Oriented Architecture* (SOA) can also support realizing self-adaptive software by facilitating the composition of loosely coupled services. Web service technology is often an appropriate option for implementing dynamic adaptable business processes and service-oriented software systems due to their flexibility for composition, orchestration, and choreography [160]. Birman *et al.* propose extensions to the web services architecture to support mission-critical applications [23]. Examples include standard services that track the health of components, mechanisms for integrating self-diagnosis into applications, and automated configuration tools. Another notable work is Autonomic Web Processes (AWP) [212], which provides web service-based processes for supporting the self-* properties.

2.3.2 Supporting Artificial Intelligence Concepts

As noted by Laddaga [107], in general, it is surprising that not much has been done to apply Artificial Intelligence (AI) techniques, such as planning and probabilistic reasoning, to develop and manage software systems. In particular, self-adaptive software has a remarkable common ground with AI and adaptive systems (see for example [200]). For the detecting process, AI can assist in log/trace analysis and pattern/symptom matching to

identify abnormal conditions or the violation of constraints. AI is also rich in planning, reasoning, and learning, which could be useful in the deciding process. One obstacle is quality assurance for AI-based systems, which becomes necessary because of the utilized intelligent, heuristic, and search-based techniques. This issue was pointed out by Parnas about two decades ago a specific class of software systems, but it is generalizable to the other systems as well [154].

One interesting approach to realize self-adaptive software is based on *AI planning*. In this approach, a software system plans and may replan its actions instead of simply executing specific algorithms. This is in particular related to the deciding process for selecting the appropriate course of action. The *planning-based adaptation* should be active all the times, through searching among existing plans or by composing adaptation actions. In fact, the adaptation engine needs *continuous planning* via *contingency planning* or *replanning* [174]. The former provides a conditional plan with alternative paths based on the sensed information, while the latter generates an alternative plan in the case that the original plan fails. A notable point is that planning, at least in its basic form, cannot be used for all of the self-* properties. According to [197], among all of the self-* properties, planning has the highest potential for being used in conjunction with self-healing. One example of using AI-planning for self-healing is the work by Arshad *et al.* [11].

An important concept that can be used in self-adaptive software is the way *software agents* model their domains, goals, and decision-making attributes. For example, an interesting goal-based model for action selection (composed of actions and goals with activation levels) has been proposed by Maes [125]. Goal-oriented requirements modeling is a well established area of research in agent-based systems and there are many research efforts that involve these models and methods in self-adaptive software, including [113] and [141].

The other important issues, especially in *Multi-Agent Systems* (MAS), are coordination models and distributed optimization techniques, which can be useful in multi-element self-adaptive software; for example, see [129] and [21]. In such systems, local and global goals, which are mostly derived from self-* properties, need to be coordinated. Tesauro *et al.* realize a sample MAS, called Unity, as a decentralized autonomic architecture based on multiple interacting agents [205]. Weyns *et al.* have also investigated employing multi-agent architectures for self-adaptive software. For example, they utilized a situated multi-agent architecture for a self-adaptive automated guided vehicle transportation system [217].

Machine Learning and *Soft Computing* are other areas with the potential to play important roles in self-adaptive software, especially through the "achieving" approach. Achieving needs analyzing the stream of sensed data and learning the best way to act. Oreizy *et al.* name evolutionary programming and AI-based learning in the category of approaches dealing with unprecedented changes with a clearer separation of concerns [149]. These algorithms generally use properties of the environment and knowledge gained from previous attempts to generate new algorithms. Genetic algorithms and different on-line learning

algorithms, such as Reinforcement Learning (RL) can also be used for this purpose. RL is a promising option for dynamic action selection [7] and decentralized collaborative self-adaptive software [51]. Tesauro discusses that RL has the potential to achieve better performance as compared to traditional methods, while requiring less domain knowledge [206]. He also adds that as RL relies on exploration for training, it is not always feasible to learn policies in a live system. Instead, offline training or a hybrid method to use the existing policies should be utilized. Fuzzy logic is also applicable to address the fuzziness of quality goals and policies [181].

Another notable field, related to this context under the umbrella of artificial intelligence is *decision theory*. This theory, in both classical and qualitative forms, can contribute to realizing the deciding process. The classical form is suitable for cases in which decision-making relies on maximizing a certain utility function in a deterministic manner, while the qualitative form is appropriate for problems including uncertainty.

One of the areas applicable to self-adaptive software is *utility theory*. The term *utility* refers to "the quality of being useful" for an action, choice, or alternative [174], and can be identified either with certainty or with uncertainty (in classical or qualitative form). Therefore, utility theory deals with methods to assign an appropriate utility value to each possible outcome and to choose the best course of action based on maximizing the utility value [96]. Walsh *et al.* demonstrate how utility functions can enable autonomic elements to continually optimize the use of computational resources in a dynamic and heterogeneous environment (a data center prototype) [214]. Poladian *et al.* employ a utility function for user needs and preferences in resource-aware services [164]. They use this function to formulate an optimization problem for dynamic configuration of these services. Nowicki *et al.* deal with decentralized optimization using utility functions [147]. Boutilier *et al.* use utility functions in self-optimizing and rely on incremental utility elicitation to perform the necessary computations using only a small set of sampled points from the underlying function [26, 156].

In practice, due to uncertainty, probabilistic reasoning and *decision-theoretic planning* are required in decision making. Markov Decision Process (MDP) and Bayesian network are two well-established techniques for this purpose. These techniques are also applicable to realizing self-* properties due to their uncertain attributes. For example, there are several research efforts utilizing these models for diagnosis and self-recovery; among other see [74], [171] and [172]. Porcarelli *et al.* also use a stochastic Petri net for decision-making in fault-tolerance [166].

2.3.3 Supporting Control Theory/Engineering Concepts

Control theory/engineering, similar to self-adaptive software, is concerned with systems that repeatedly interact with their environment through a sense-plan-act loop. The no-

tions of adaptation and feedback have been discussed for decades in control theory and engineering, and have been utilized in designing and developing systems in various domains. One of the interesting resources on utilizing control concepts and feedback loop in computing systems is the book written by Hellerstein et al. [71]. This section briefly reviews applicability of some ideas from feedback control in designing self-adaptive software systems. Interested readers can find more information in [186] and [40].

The control-based paradigm considers the software system (adaptable software) as a *controllable plant* with two types of inputs: control inputs, which control the plant's behavior, and disturbances, which change the plant's behavior in an unpredictable manner [101]. A *controller* (adaptation engine) changes the values of the plant's control inputs. The control-based paradigm is often based on a model of the software plant's behavior. For instance, Litoiu *et al.* use a hierarchical Layered Queue Model (LQM) of a software system for tuning parameters (weak adaptation) [120]. Abdelwahed *et al.* [1] also show that a model-based control architecture can realize the self-optimizing property, by tuning the plant parameters. Moreover, Bhat *et al.* discuss applying online control models to achieve self-managing goals by extending the ACCORD component framework [20]. Although closed-loop is the most widely used model for control-based self-adaptive software, adaptive and reconfigurable models are also recommended for several reasons, including large-range dynamic disturbances [101]. On the other hand, considering the discrete nature of software systems, one of the appropriate control-based approaches for self-adaptive software is *supervisory control* of *discrete event system* (DES) [168]; see for example [208] and [93].

Traditionally, control theory has been concerned with systems that are governed by the laws of physics. This allows them to make assertions about the presence or absence of certain properties, which is not necessarily the case with software systems. In practice, checking software controllability or building a controllable software is a challenging task, often involving non-intuitive analysis and system modifications [92]. Therefore, some researchers believe that applying this approach to software is often more complex than the case of traditional control systems [171].

2.3.4 Supporting Network and Distributed Systems Concepts

Techniques used in network and distributed computing can be extensively applied to self-adaptive software. This is due to the fact that the bulk of the existing software systems are distributed and network-centric. Although it may be difficult to directly apply some of these techniques to all layers of self-adaptive software (i.e., policy management at the application layer), their usage in addressing adaptation requirements and the engineering of such systems is promising. Another line of research in this area concerns Peer-to-Peer (P2P) applications and ad hoc networks, which deal with the dynamic change of environment, architecture, and quality requirements. Research in this area often uses self-organizing

elements in a bottom-up approach. However, as explained earlier, this thesis does not cover systems with the self-organizing property.

Policy-based management is one of the most successful approaches followed in network and distributed computing [195]. Policy-based management specifies how to deal with situations that are likely to occur (e.g., priorities and access control rules for system resources). Most of the definitions given for policy emphasize on providing guidance in determining decisions and actions. The policy management services normally consist of a policy repository, a set of Policy Decision Points (PDP) for interpreting the policies, and a set of Policy Enforcement Points (PEP) for applying the policies [216]. The most widely used policy type in networks is the *action policy* (in the form of event-condition-action rules), which is also applicable to self-adaptive software. In addition, other policy types like *goal policy* (specifying a desired state) and *utility policy* (expressing the value of each possible state) can also be exploited in self-adaptive software [98]. The adaptation policies may need to be changed based on new requirements or conditions. Some research efforts have addressed this issue. For example, Lutfyyia *et al.*, among several other efforts on policy-based management, have proposed a control-theoretic technique for dynamic change of policies in a data center [6]. Policy-based management has been adopted in a number of self-adaptive software research; see for example [95, 17, 178].

Quality of Service (QoS) management, another successful area in networking and distributed systems [77], is closely related to policy management [124]. In Information Technology (IT) and particularly service-oriented systems, QoS is one of the most important factors for the provided service and the business. "The term QoS is used in many meanings ranging from the users perception of the service to a set of connection parameters necessary to achieve particular service quality" [64]. The main artifact to define QoS conditions is Service Level Agreement (SLA) that includes guarantee clauses provided by service providers for their clients. These clauses should clearly state: i) what services are provided by the service provider, and what the guarantees are for the services, ii) under what conditions these are valid, and iii) what happens if any of these clauses are violated. Generally, the main focus of SLA is on performance and availability. Several definitions of SLA highlights the above points:

> *"A negotiated agreement between a customer and the service provider on levels of service characteristics and the associated set of metrics. The content of SLAs varies depending on the service offering and includes the attributes required for the negotiated agreement" [170].*
>
> *"SLA can be first defined internally by a business to ensure the satisfaction of its end-user experience, such as the speed of at which a Web search engine retrieves results or, second as a legally binding contract, such as a business-to-business e-commerce application" [69].*

An effective SLA exhibits three key properties: specificity containing clear conditions on metrics, flexibility in anticipation of unexpected conditions, and realism regarding system capacity and resources [69]. Normally, an SLA consists of objectives, monitoring conditions, and financial/legal conditions. Objectives are specified as SLOs (Service Level Objective). Each SLO often addresses one specific objective related to a single or a few number of system attributes. Monitoring conditions explain the parameters in SLA and how to measure them. Finally, the penalties of violating SLOs, SLA expiry date and other legal issues should be specified. There may be several user levels with different SLAs; for instance gold-level with a better service guarantee and higher cost in comparison with platinum-level.

QoS requirements are related to non-functional requirements of a system, and consequently, they can be linked to self-* properties in distributed software systems. In this context, QoS management methods rely on either modeling the application (e.g., queuing models), or using well-understood components (e.g., Prediction-Enabled Component Technology (PECT) [220]). Therefore, QoS management can assist in modeling the quality factors of a self-adaptive software system (and consequently self-* properties) and also in realizing adaptation processes.

Another powerful technology borrowed from distributed and network-based systems is *middleware* [136]. Middleware-based adaptation (in all four sub-layers discussed by Schmidt [187]) would also be applicable to adaptation processes. For instance, generic components of decision-making and change detection can be realized at the middleware level.

One of the well established areas in networks and distributed systems is *resource management*. Specifically, *virtualization* techniques can have a significant impact on the quality of self-adaptive software. Virtualization reduces the domain of an adaptation engine to the contents of a virtual machine [138]. Consequently, dynamic resource management and resource provisioning are easier to handle. Virtualization also provides an effective way for legacy software systems to coexist with current operational environments [19]. This property can be utilized in building adaptable software from legacy systems.

Monitoring and sensing techniques have been widely used in networks and distributed systems. Basic techniques, like heartbeat monitoring, and more advanced techniques, like pulse monitoring, have been used in self-adaptive and self-managing software [72]. One important issue, that is quite significant in self-adaptive software is the cost of sensing and monitoring. This issue has been addressed extensively in networking (e.g., in [49]) and distributed systems (e.g., in [2] and [175]).

2.4 Research Projects

The projects in this section are selected from different academic and industrial sectors to capture main research trends in the broad area of self-adaptive software. The information is collected from many academic and industrial research projects (from companies such as IBM, HP, Microsoft, Sun). However, a few of them are selected to represent the major research ideas in this field. Space limitations, the diversity of ideas, and their impact on the field, are the concerns taken into account for the selection.

Among other goals, the discussions in this section aim to identify the existing research gaps in this area. For this purpose, the projects are analyzed from several points of view. Since some of these projects are not available for evaluation, the reported properties and features are based on the referenced material.

Table 2.3 lists the selected projects sorted based on the date of cited publication(s). In the case of several related publications from the same research group, the more important ones are cited. These projects are selected on the basis of their impact on the area and the novelty/significance of their approach. In the rest of this section, the selected projects are compared in relation to three different views, namely *self-* properties*, *adaptation processes*, and the proposed *taxonomy facets*.

The first view discusses the major self-* properties that are supported by each project, as shown in Table 2.4. We can see that the majority of these projects focus on one or two of the known self-* properties. This shows that the coordination and the orchestration among multiple self-* properties have not yet received the full attention they deserve. Another notable point is that a limited number of projects in the literature support self-protecting (only one project in the selected set). Generally speaking, this is due to constant changes in the network topology, the increasing variety of software components/services, and the increasing complexity, as well as variety of attacks and viruses [167]. Most of the research dealing with the self-protecting property focus on the network layer, and particularly, on detecting attacks. Such research efforts are outside of the main scope of this thesis, as already mentioned.

The second view is concerned with how the selected projects address adaptation processes. Table 2.5 compares and categorizes the selected projects according to four levels from “no support” to “high support”. The level for each process is determined based on efficiency, coverage of different aspects, and support for available standards. Each of the processes also has its own specific aspects. For example, to evaluate the *deciding process*, it has been investigated whether a project takes into account dynamicity and uncertainty. To analyze the table column-wise, a vector is included with four components reflecting the relative frequencies of the different levels in each column, ranging from “no support” ($-$) to “high support” (H). For example, the vector $(2, 5, 7, 2)/16$ under the *Monitoring*

Table 2.3: Selected projects in the area of self-adaptive software

Projects	Summary
Quo [123]	Quality Objects (QuO) provides Quality Description Languages (QDL) for specifying possible QoS states, the system resources and mechanisms for measuring and controlling QoS, and behavior for adapting to changing levels of available QoS at run-time.
IBM Oceano [8]	Developing a pilot prototype of a manageable infrastructure for a computing utility powerplant
Rainbow [57, 58]	Proposing an architecture-based adaptation framework consisting of an adaptation infrastructure and a system-specific adaptation knowledge
Tivoli Risk Manager [207]	Providing an integrated security management structure by filtering and correlating the data from different sources and applying dynamic policies
KX [91, 209]	A generic framework for collecting and interpreting application-specific behavioral data at run-time through sensors (probes) and gauges
Accord [122]	Providing a programming framework for defining application context, autonomic elements, rules for the dynamic composition of elements, and an agent infrastructure to support rule enforcement
ROC [33, 35]	Building Recursively Recoverable (RR) systems, based on micro-reboot, online verification of recovery mechanisms, isolation and redundancy, and system-wide support for undo
TRAP [176, 177]	A tool for using aspects and reflective technique for dynamic adaptation in Java, TRAP/J, and .Net framework, TRAP/.Net
K-Component [52, 51]	A meta-model for realizing a dynamic software architecture based on *Adaptation Contract Description Language* (ACDL) for specifying reflective programs. ACDL separates the specification of a system's self-adaptive behavior from the system components' behavior
Self-Adaptive [171]	Establishing a model-based diagnosis and automatic recovery approach using of method deprecation and regeneration with the aid of a decision-theoretic method dispatch
CASA [142]	Contract-based Adaptive Software Architecture (CASA) supports both application-level and low-level (e.g., middleware) adaption actions through an external adaptation engine
J3 [218]	Providing a model-driven framework for application-level adaptation based on three modules J2EEML, JAdapt, and JFense respectively for modeling, interpreting and run-time management of self-adaptive J2EE applications
DEAS [113, 222, 146]	Proposing a framework for identifying the objectives, analyzing alternative ways of how these objectives can be met, and designing a system that supports all or some of these alternative behaviors using requirements goal models
MADAM [54]	Facilitating adaptive application development for mobile computing, by representing architecture models at run-time to allow generic middleware components to reason about adaptation
M-Ware [105]	Developing middleware to enable agility, resource-awareness, run-time management and openness in distributed applications, by especially addressing performance concerns and business policies
ML-IDS [4]	Detecting network attacks by inspecting and analyzing the traffic using several levels of granularity (Multi-Level Intrusion Detection System - ML-IDS), and consequently proactively protect the operating system by employing a fusion decision algorithm

Table 2.4: Comparing projects in terms of self-* properties "$\surd$": Supported, "-": Not supported.

Projects	Self-* Properties			
	Self-Configuring	*Self-Healing*	*Self-Optimizing*	*Self-Protecting*
Quo	$\surd$	-	$\surd$	-
IBM Oceano	$\surd$	-	$\surd$	-
Rainbow	$\surd$	$\surd$	$\surd$	-
Tivoli Risk Manager	-	-	-	$\surd$
KX	$\surd$	-	-	-
Accord	$\surd$	-	-	-
ROC	-	$\surd$	-	-
TRAP	$\surd$	-	-	-
K-Component	$\surd$	-	-	-
Self-Adaptive	$\surd$	$\surd$	-	-
CASA	$\surd$	-	$\surd$	-
J3	-	-	$\surd$	-
DEAS	$\surd$	$\surd$	-	-
MADAM	$\surd$	-	$\surd$	-
M-Ware	$\surd$	-	$\surd$	-
ML-IDS	-	-	-	$\surd$

column shows that out of the 16 selected projects, there are 2 with "no support", 5 with "low support", 7 with "medium support", and 2 with "high support". The column-wise assessment shows that monitoring, detecting, deciding, and acting each have only 2 or 3 projects with "high support" out of the selected 16 . These observations indicate that one needs to provide comprehensive solutions to realize all adaptation processes at a high level.

The taxonomy introduced in Section 2.2 provides a third view. This view is summarized in Table 2.6, which will also be analyzed column-wise. Before analyzing this table, it should be noted that the possible values for some of the facets are not mutually exclusive. For example, a project can rely on a hybrid approach including both *making* and *achieving*. The findings related to this view based on each taxonomy facet (corresponding to different columns in the table) are as follows:

- Layer (L): Most of the projects focus on the application layer, which is expected, since the focus of this thesis is on this layer.

- Artifact & Granularity ($A\&G$): Various artifacts at different granularity levels have been addressed, which is a positive point.

Table 2.5: Comparing projects in terms of adaptation processes- “H” (High Support): Provides explicit features to support the process extensively, “M” (Medium Support): Provides generic features to partially support the process, “L” (Low Support): Provides limited support, “–” (No Support)- For example, the vector $(2, 5, 7, 2)/16$ shows that out of the 16 selected projects, there are 2 with “no support”, 5 with “low support”, 4 with “medium support” and 2 with “high support”.

Projects	**Adaptation Processes**			
	Monitoring	Detecting	Deciding	Acting
Quo	L	L	L	M
IBM Oceano	M	M	M	M
Rainbow	H	M	M	M
Tivoli Risk Manager	–	H	–	–
KX	H	M	L	M
Accord	L	L	M	L
ROC	L	L	M	H
TRAP	L	L	–	M
K-Component	L	L	H	L
Self-Adaptive	M	H	H	M
CASA	M	M	L	H
J3	M	M	L	M
DEAS	M	M	M	M
MADAM	M	M	L	M
M-Ware	M	M	M	L
ML-IDS	M	H	M	M
Column-wise Assessment	(1,5,8,2)/16	(0,5,8,3)/16	(2,5,7,2)/16	(1,3,10,2)/16

- Impact & Cost (*I&C*): Most of the projects utilize both weak and strong adaptation actions. This is also a positive point because it is possible to use low/high cost and local/global actions depending on the circumstances.
- Making/Achieving (*M/A*): The achieving approach is rarely observed. This means learning and evolutionary algorithms have not yet been widely used in this area.
- External/Internal (*E/I*): All of the projects use the external approach, which means that they all support separation of the adaptation mechanism from the application logic.
- Static/Dynamic Decision-Making (*S/D DM*): The number of dynamic deciding processes is not too high, but is notable. This is partly due to the research activities in the area of policy-based management.

- Open/Close (O/C): Another remarkable observation is the high number of projects supporting close adaptation. This can be interpreted as the inability to attain openness due to stability and assurance concerns.
- Specific/Generic (S/G): A number of projects (7 out of 16, including J3) have been developed based on component-based systems. The justification is that such components are loosely coupled entities that can be changed dynamically at run-time, easier than the other entities.
- Model-Based/-Free (MB/F): Most of the projects are model-based, which is not surprising noting the wide-spread application of model-based and model-driven approaches in engineering disciplines.
- Reactive/Proactive (R/P): Most of the projects are reactive, which is not generally a disadvantage. However, for some domains, it is required to have proactiveness in order to decrease the aftereffects of changes, or to block change propagation (e.g., faults in safety-critical systems).
- Continuous/Adaptive Monitoring ($C/A\ M$): Most of the projects still use continuous monitoring, which is not preferable noting the cost and the load of this process.
- Human Involvement (HI): Most of the projects do not include an appropriate human interface. This matter impacts the usability and trustworthiness of these systems in practice.
- Interoperability (I): Only one of the projects proposes a mechanism for interoperability with other self-adaptive or autonomic elements or systems. This matter limits their applicability, especially in emerging service-oriented applications.

Table 2.6: Comparing projects in terms of the taxonomy facets- **"-"**: is not supported, **"?"**: Not clearly known. "L"" Layer, "A&G": Artifact & Granularity, "I&C": Impact & Cost, "M/A": Making/Achieving, "E/I": External/Internal,"S/D DM": Static/Dynamic Decision-Making, "O/C": Open/Close, "S/G": Specific/Generic, "MB/F": Model-Based/-Free, "R/P":Reactive/Proactive, "C/A M": Continuous/Adaptive Monitoring, "HI": Human Involvement, "I":Interoperability.

	Object to adapt			*Realization*						*Temporal*		*Interaction*	
				Approach			*Type*						
Taxonomy/ Projects	*L*	*A&G*	*I&C*	*M/A*	*E/I*	*S/D DM*	*O/C*	*S/G*	*MB/F*	*R/P*	*C/A M*	*HI*	*I*
Quo	application	aspect	w	m	e	s	c	comp.-based	mb	r	c	-	-
IBM Oceano	Infrastructure & network	data center	w/s	m	e	d	o	data center	mb	r	c	can be	-
Rainbow	application	architecture	w/s	m	e	semi-d	c	known arch. styles	mb	r	semi-a	can be	-
Tivoli Risk Manager	network & application	system	-	m	e	-	?	generic	mb	r	c	yes	yes
KX	application	application	w/s	m	e	d	o	generic	mb	r	semi-a	-	-
Accord	application	components	w/s	m	e	d	c	generic	mb	r	c	can be	-
ROC	application & middleware	components/ subsystems	w	m	e	s	c	comp.-based	mb	r	c	-	-
TRAP	application	aspect	w	m	e	-	c	comp.-based	-	r	c	yes	-
K-Component	application	component	w/s	m/a	e	d	c	comp.-based	mb	r/p	c	-	-
Self-Adaptive	application	method	w	m/a	e	semi-d	c	generic	mb	r	c	-	-
CASA	application	component	w/s	m	e	semi-d	c	comp.-based	mb	r	c	-	-
J3	application	aspect	w	m	e	s	c	J2EE app.	mb	r	c	-	-
DEAS	application	application	?	m	e	s	c	generic	-	-	-	-	-
MADAM	middleware	architecture	s	m	e	s	c?	comp.-based	mb	r	c	-	-
M-WARE	middleware	parameters	w	m	e	s	o?	generic	mb	r	c	-	-
ML-IDS	network	parameters	w	m	e	d	c?	generic	mb	r/p	c	-	-

2.5 Research Challenges

Self-adaptive software creates new opportunities, and at the same time, poses new challenges to the development and operation of software-intensive systems. This section aims to identify the challenges in realizing self-adaptive software, and underlying the parts covered by this thesis.

2.5.1 General Challenges

Kephart categorizes the challenges in the context of autonomic computing as [99]: i) *Element/Component Level Challenges* relate to building element interfaces and contracts to share information, designing/implementing proper adaptation processes, and designing an appropriate architecture for elements in order to execute and coordinate the adaptation processes, ii) *System Level Challenges* relate to coordinating self-* properties and adaptation processes between elements, specifying the evaluation criteria, and defining appropriate architectures to fulfill this level's requirements (e.g., inter-element communication), and iii) *Human-System Interaction Challenges* relate to building trust, providing an appropriate mechanism for collecting user policies, and establishing a proper mechanism to involve humans in the adaptation loop.

Although the above classification provides insight into the challenges associated with self-adaptive systems, it does not quite fit into the taxonomy introduced in 2.2. Moreover, for some of the identified challenges, depending on the underlying design decisions, they may be at the element level or at the system level (e.g., coordinating self-* properties). The underlying challenges are classified based on the points summarized in Figure 2.5.

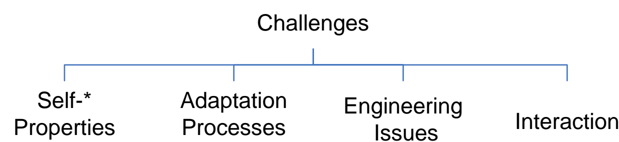

Figure 2.5: Classifying self-adaptive software challenges

The first category deals with the engineering challenges for requirements analysis, design, implementation, and evaluation of self-adaptive software. The second category covers the obstacles in front of realizing self-* properties, both individually and collectively. The third category discusses the challenges in designing and developing each adaptation process, and finally the last one addresses interaction challenges of self-adaptive software and human or other systems. This thesis does not expand the details of these challenges, and interested readers can find more information in [186].

2.5.2 Challenges Addressed by this Thesis

Among the mentioned challenges, some parts are addressed by this thesis. In each category, these are highlighted as the following:

- Self-* properties: In this category, the thesis mainly concerns self-configuring and self-organizing, whereas there is no assumption excluding other properties. The emphasis is on handling several self- * properties in an adaptation manager.
- Adaptation processes: Among four adaptation processes, this thesis addresses mainly the deciding process and partially the detecting process. It is notable that the other two processes have been implemented for case studies, whereas QFeam does not cover them.
- Engineering issues: This thesis focuses on engineering an adaptation manager. Specifically the emphasis is on building a run-time adaptation model, based on composition patterns, and adaptation mechanism design. Therefore, requirements analysis and design phases are addressed more than implementation and testing phases in the view of common software engineering development process.
- Interaction: This thesis does not deal with the interaction challenges, either with human or other systems. In the empirical studies, it is considered that an adaptation manager may not be able to handle a situation and the control should be handed over to human agents. Therefore, the assumption is that human agent(s) supervises self-adaptive software, even though not explicitly noted in QFeam.

The preceding challenges have been selected based on the defined problem and the limitations set for this thesis.

2.6 Summary

This chapter reviewed briefly the principles and terminologies in the domain of self-adaptive software. A taxonomy was provided to capture the fundamental aspects of adaptation. Then a landscape was illustrated from two points of view, the supporting disciplines and highlights of the research projects. A concise classification of challenges was also introduced.

This thesis focuses on some specific challenges regarding the problem defined in Chapter 1. In short, based on the categorized challenges first, self- * properties related to performance and availability quality factors are emphasized. Second, the deciding and

detecting processes are covered, and third, these processes are discussed in an engineering framework for building an adaptation manager. The last point is that this research does not take into account the interaction problems.

Chapter 3

QFeam: A Quality-Driven Framework for Engineering an Adaptation Manager

"*Without goals, and plans to reach them,*
you are like a ship that has set sail with no destination."
Fitzhugh Dodson

This thesis emphasizes that the challenging problem in the self-adaptive software domain is not currently finding a holistic solution for realizing self-* properties and corresponding adaptation processes (e.g., deciding process). Specifying problem space elements and moving toward an engineering process for building self-adaptive software are currently more important than solutions for specific self-* properties. This thesis introduces a framework that enables engineers to employ various patterns and mechanisms in engineering adaptation managers by focusing on quality requirements.

Figure 3.1 illustrates the general schema of engineering self-adaptive software. The initial step is to *elicit adaptation requirements* from the set of user requirements. As emphasized previously, quality aspects are the main focus of this process for self-adaptive software. The two major phases are *engineering adaptable software* and *engineering adaptation manager*. The former results in adaptability via sensors and effectors, while the latter realizes the manager in charge of navigating the adaptation processes at run-time. These two may interact with each other as shown by a bidirectional arrow in Figure 3.1.

The proposed *Quality-driven Framework for Engineering an Adaptation Manager (QFeam)* in this thesis targets engineering the adaptation manager of self-adaptive software. In QFeam, two key processes are emphasized for building a run-time adaptation model and for adaptation mechanism design. In the former, the main concern is modeling problem

space entities in a way that are usable at run-time by the adaptation manager. The latter utilizes this model (e.g., for reasoning or conflict resolution) to adapt the system by selecting an appropriate action/plan.

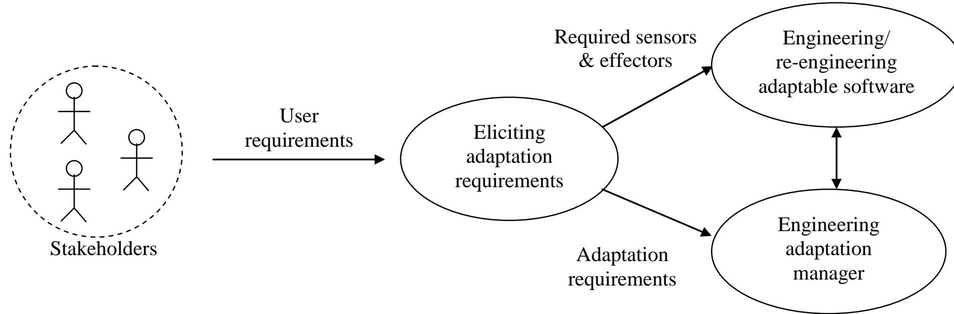

Figure 3.1: The big picture of engineering self-adaptive software

The following sections briefly decompose the above two phases of engineering adaptable software and the adaptation manager. However, because the main focus of this thesis is on the second phase, the following two chapters are dedicated to engineering the adaptation manager. Eliciting adaptation requirements is also addressed in this phase. In the investigated case studies the first phase will be discussed in more detail, specifically with respect to the employed technologies.

3.1 Main Role of Quality Requirements

A key idea in QFeam is that adaptation requirements need to be kept alive in the system as the representative of user requirements (i.e., stakeholders' expectations from system). Generally, adaptation can be described in terms of user and system requirements. In RE, the specification of system requirements alone scopes the system too narrowly, and may cause us to miss alternative solutions [128]. In adaptation, the objective is to keep alternative system requirements, that can satisfy user requirements in different situations. These different situations are domain statements (D) in Jackson's notation [85]. Thus, according to this notation, adaptation is to switch between alternative specification statements (S) based on domain statements (D), in order to satisfy requirements statements (R); or in other words, switching system requirements to satisfy user requirements.

Normally, the number of system requirements is more than user requirements, and they include variants and alternative specifications [128]. The variability is particularly

important regarding conflicts and trade-offs between quality requirements. In general, user requirements include both functional and quality requirements. However, as described in the previous chapter engineering self-adaptive software can benefit from software quality factors and quality requirements due to several reasons. For instance, self-* properties are related to quality factors, as discussed in Section 2.1.2. Therefore, these properties can be achieved by satisfying corresponding quality requirements. Thus models established for the quality requirements, and methods for quantifying them can be helpful in engineering self-adaptive software. Furthermore, quality goals are strongly related to adaptation requirements. The role of quality goals in the adaptation manager is as important as their role in software engineering, in general, as discussed by Yu and Mylopoulos [221]. This role is especially significant in addressing the "why" question for the adaptation requirements. Therefore, in this context, *adaptation requirements* only involves quality requirements. Chapter 4 elaborates how to elicit and analyze the adaptation requirements in more detail.

3.2 (Re-)Engineering Adaptable Software

The first phase of constructing self-adaptive software is building adaptable software. This can be an engineering or a re-engineering process. In both of the engineering and re-engineering approaches, specifying sensors and effectors from adaptation requirements is the first step. Specifying and even instrumenting sensors seem to be easier than for effectors. By looking at the list of sensors and effectors in Table 2.1, this statement makes sense. Moreover, several other reasons support this claim:

- Quality goals that are related to quality factors can be quantified with metrics and measures. For instance, performance has well-established quantifiable attributes such as response time. Measuring these attributes has been addressed extensively.
- Many enterprise distributed applications are based on middlewares and virtual machines. These technologies provide sensors or facilities to instrument sensors for such applications. Sensors, in this case, just read a parameter from application components or underlying servers. However, effectors are not widely supported by middlewares and servers, or they are not still reliable enough.
- Sensing application attributes causes performance downgrade and overload on resources, which is negligible or considerable depending on the number of sensors and the attributes. However, effectors potentially have more severe impacts, either positive or negative, not only on quality requirements, but also on functional requirements. This is because the effectors dynamically change system attributes or artifacts at run-time.

The next step is instrumenting sensors and effectors into the software application in order to augment adaptability. This step can be accomplished by either of, or a combination of the two following approaches:

- Manual retrofitting of sensors and effectors.
- Automatic or semi-automatic instrumentation of sensors and effectors, that could be possible through transformations due to applying appropriate design patterns. A set of suitable patterns and transformations is yet to be established and practiced.

Although building adaptable software is not the main focus of this thesis, a remarkable amount of time has been spent on this issue in the case studies. This is due to the fact that there is still no benchmark or shared system to serve as a basis for comparing adaptation mangers and different approaches in realizing adaptation processes. In the case studies, manual approach is selected because implementing the second approach takes much more time, and is out of the scope of this thesis.

3.3 Engineering an Adaptation Manager with QFeam

An adaptation manager, like any other software system, needs an engineering process including the general phases of requirements analysis and specification, design, implementation, and testing. Although establishing a complete and efficient process for this purpose is a long-term objective, QFeam captures the essential processes, as depicted in Figure 3.2. The main objective of this thesis is to put emphasis on two processes of engineering an adaptation manager: i) modeling the essential entities of problem space, and ii) designing a proper adaptation mechanisms based on the built model. These two processes can be accomplished in different ways, and QFeam covers several well-established approaches by highlighting commonalities.

The implementation phase is discussed only for Java-based systems, and some ideas for the evaluation are discussed mainly for goal-driven approaches. Note that the monitoring and acting processes are not covered by QFeam. However, these processes are implemented for the cases studies. Realizing the deciding process, and partially the detecting process are the chief objectives of QFeam.

In the modeling phase, the emphasis is on capturing adaptation requirements. The set of quality goals is the main tenet of these requirements, that navigates the process by addressing the essential "why" question. On the other hand, domain attributes including adaptable software *self* and *context* entities are important. Capturing these two sets of

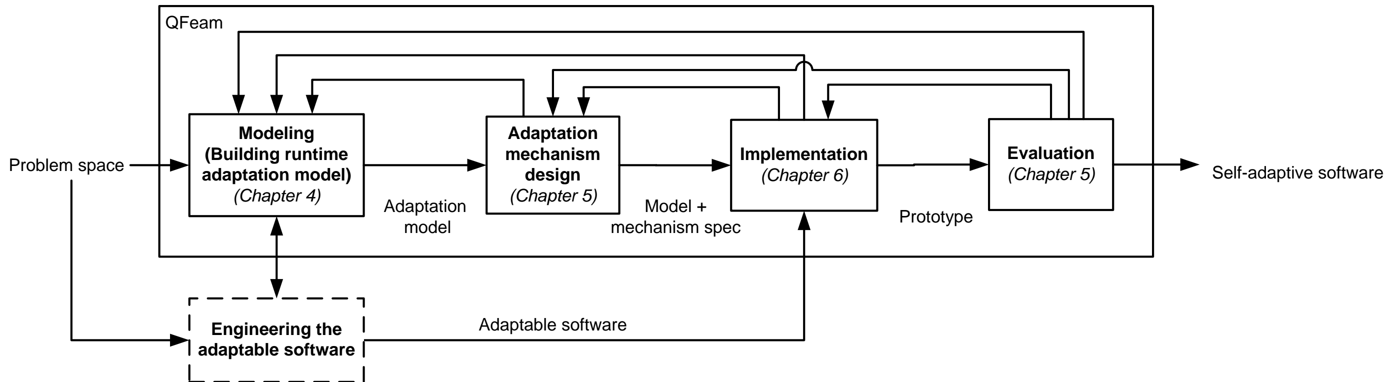

Figure 3.2: Quality-driven framework for engineering an adaptation manager (QFeam)

attributes are essential to provide two *self-awareness* and *context-awareness* properties, which are the foundations of higher level self-* properties (See Figure 2.1). The third set that also plays a key role is the set of adaptation actions. Actions are changes provided by effectors in self-adaptable software.

A simple example from biology makes the role of these three concepts- goals, attributes and actions- more sensible. Assume the human body as an adaptive system that has a goal of tuning its temperature. Self is the collection of body organs and context is the surrounding environment that interacts with the body. Changes in self, such as infection causes fever (temperature change), while moving to a cold place also triggers a temperature change. In both cases, the body reacts to changes by modifying the blood circulation speed. Thus, from a high-level conceptual view, goals, attributes of self and context, and actions help the body adapt itself properly to various situations.

An adaptation mechanism realizes the four essential processes discussed in Section 2.1.4. The mechanism can benefit from various techniques and theories, such as optimization and planning. Although these approaches have differences in their underlying models, the commonalities are not negligible. This is specifically true in problem space modeling. This thesis first assumes high-level conceptual entities commonly used in most approaches in self-adaptive software research. Next, several composition patterns to link these entities are discussed. Corresponding to these patterns, several adaptation mechanisms are investigated.

The proposed framework deals with the adaptation model and mechanism by focusing on flexibility and extensibility. Interestingly, these objectives have been briefly addressed for similar contexts in software engineering previously. For instance, Shaw and Garlan point out that an architectural style can be based on the process control model, by incorporating essential parts of data/process variables (i.e., model) and computational elements/control

algorithm (i.e., mechanism) [192].

3.3.1 Building Run-Time Adaptation Model

In the self-adaptive software research community, there are discussions on the suitability of numerous models for adaptation requirements and domain information. This includes architectural, configuration, performance and reliability models; to name a few. The important point is the common problem space concepts these models try to represent. Three concept spaces can be identified, as pointed out previously:

- *Goal space*: Quality goals are commonly used to represent adaptation requirements. As outlined before, these goals play a key role in navigating the adaptation mechanism by addressing the "why" question. In the rest of this thesis, *adaptation goals*, or simply *goals* are used interchangeably.
- *Attribute space*: By considering the key role of the closed-loop paradigm in adaptation, all of these models track the domain attributes via sensors. As emphasized before, these attributes belong to the self and context of adaptable software. *Domain attributes* or simply *attributes* are used to refer the entities in this space.
- *Action space*: Adaptation is based on changeability, and the adaptation manager needs effectors to apply changes to adaptable software. The ways an adaptation manager can apply changes are defined in the action space. *Adaptation actions* or simply *actions* are the terms used throughout this thesis for entities in this space.

These three concept spaces are the main tenets of the problem space. Figure 3.3 depicts the conceptual model combining these spaces. A fourth optional space can be also added to complement the model. This space may be required because some solutions need structural or behavioral information from the adaptable software system. The links between the main three spaces are shown by dotted lines, indicating that different patterns may bind the constituent entities of each space together, and all links do not exist in all solutions. These composition patterns are quite important, not only for building the model, but also for designing the adaptation mechanism. For instance, in a goal-driven mechanism the monitoring and detecting processes use the links between the attribute and goal spaces, while the deciding and acting processes function based on the links between the goal and action spaces.

The initial step in defining the adaptation model is capturing entities in each concept space. These entities can be linked in each space together to build models (i.e., intra-space links). The more important task is then linking entities or models in each space

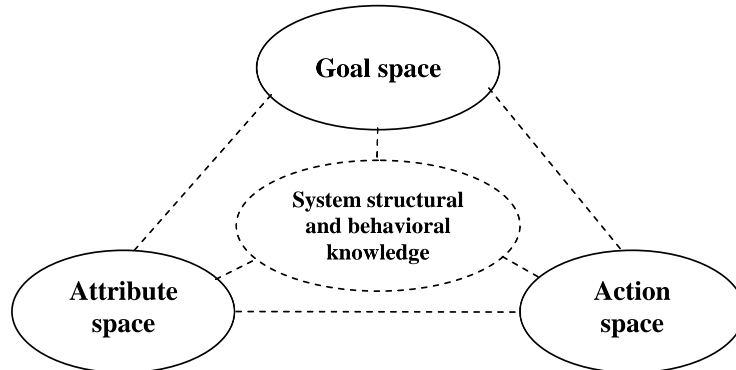

Figure 3.3: Adaptation conceptual model

to each other (i.e., inter-space links). Different composition patterns can be adopted for establishing inter-space links. Next chapter elaborates three of such patterns.

A valid question is that what differences exist between this adaptation model and the Belief-Desire-Intention (BDI) model for intelligent agents [169]. Beliefs represent the system knowledge about the situation (self and context). Desires refer to the objectives and can be instantiated by goals, and intentions are what an agent decided to do and may include plans. The BDI model has some limitations that are discussed in [163]. The three elements of BDI would be roughly mapped to the preceding three spaces. The point is that the introduced model overcomes some limitations of BDI; for instance in representing goals explicitly. Nevertheless, BDI and its extensions in multi-agent systems are well-established and built on a strong body of knowledge, that could be useful in designing self-adaptive software systems.

An important point is that a problem space model including the adaptation requirements is used at run-time. Figure 3.4 shows two ways of using models in engineering an adaptation manager. Most of the existing approaches, particularly for modeling requirements, adopt the first approach; for example see [114]. QFeam focuses on the second approach, which embeds the model in the adaptation manager. This run-time model can be a simplified version of the model used in the development phase.

3.3.2 Adaptation Mechanism Design

The main theme of adaptation is the closed loop that includes the four adaptation processes. While there is a consensus over the adaptation processes, although by different names, there are different approaches to design and realize the processes. One of the key factors

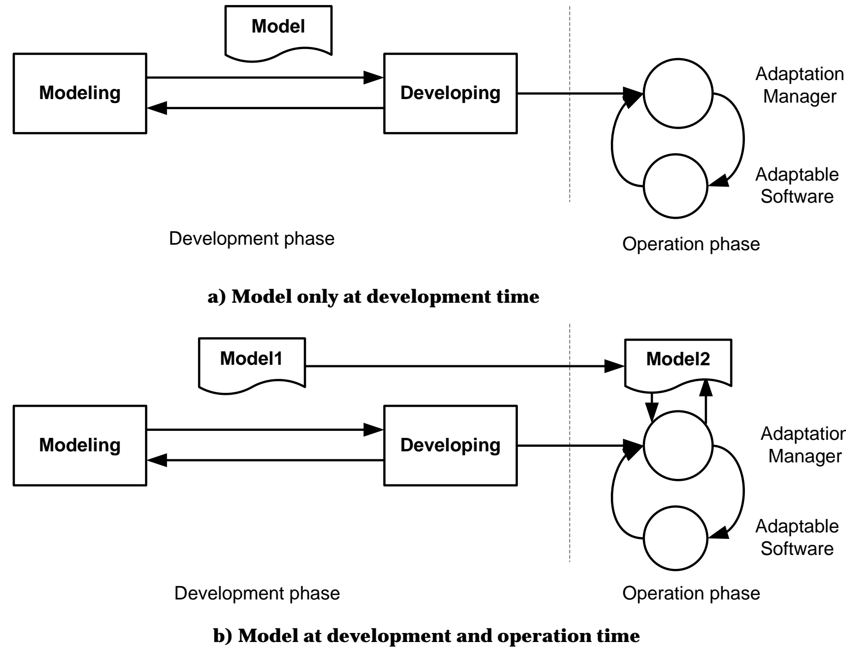

Figure 3.4: Different approaches of using models in engineering adaptation manager

in designing the adaptation mechanism is the structure of the adaptation model, and particularly the adopted composition pattern.

In fact, selecting a composition pattern and designing the adaptation mechanism are highly linked together. This thesis does not provide a complete catalog of composition patterns and mechanisms, instead three options are discussed. Note that for these patterns, the internal structure of each concept space can be changed. Therefore, various adaptation managers can be generated with those patterns. The main objective is to cover the commonly used mechanisms in this research area.

Many theories and algorithms have been employed for the design and implementation of adaptation mechanisms [186]. Lightstone discusses a few of the foundation techniques for solving self-management problems, such as expert system, tradeoff elimination, static and online optimization, control theory, and correlation modeling [119]. Each of these techniques may define entities in each of the three spaces differently and compose different structures. Choice of the adaptation mechanism also depends on the adaptation requirements and which self-* properties are desired to be attained. As pointed out before some mechanisms are suitable for specific self-* properties. Therefore, a framework that addresses multiple self-* properties and associated goals is more demanding. QFeam aims at providing this capability.

3.4 Design Considerations

Several principles are considered in engineering an adaptation manger. For the proposed framework, QFeam, the following principles are taken into account:

3.4.1 Separation of Concerns

Separating application logic from adaptation logic is an important design concern for self-adaptive software systems. This includes: i) adaptation knowledge represented by attribute, action, and goal spaces, ii) adaptation mechanism implementation and other helper modules such as a policy engine which interpret and execute adaptation logic.

Similar to separation of concerns as a general principle in software design, this issue has tremendous impacts on the maintainability and testability of an adaptation manager. From another point of view, due to the fact that self-adaptive software mainly emphasizes quality requirements, the specification and control of these requirements have been separated from the functional parts of the application.

3.4.2 Goal-driven Adaptation

Goal-oriented requirements engineering is one of the common ways to model requirements and to analyze the impact of variant design decisions on them. In engineering the adaptation manager, goals can be employed in two phases:

- In requirements engineering and designing an adaptation manager
- In run-time quality goals inside the adaptation manager to navigate the processes, particularly the deciding process

For the proposed quality-driven framework, the second item is especially emphasized because many of the existing research works in this area have not dealt with a systematic way to address this issue. The main belief is that explicit quality goals should be used in the adaptation manager. In particular, the benefits of using goals in modeling an adaptation manager are as follows: i) representing a high-level view of the requirements, which is better suited for decision-making, i) explicitly addressing the "why" question through tracing adaptation goals, ii) systematic addressing of multiple and probably conflicting self-* properties and adaptation requirements.

In order to gain these benefits, QFeam takes into account the following two concerns:

- *Addressing Multiple Goals*: As emphasized previously, one of the challenges for engineering an adaptation manager is addressing several self-* properties and consequently multiple quality goals. These goals can be local for separate components, or global system-wide goals. Furthermore, these goals can be satisfied at the same time, or may be conflicting due to limited resources and the other concerns (e.g., performance vs. security).

  The adaptation model needs to enable the adaptation mechanism to systematically address multiple goals. Therefore, resolving conflicts, coordination, or the orchestration of these goals should be taken into account in the framework.

- *Tracing Quality Requirements at run-time*: The adaptation manager needs a run-time tracing module in the adaptation mechanism to keep track of "alive" goals. This is essential for realizing the detecting process. In this process, entities in the attribute space continuously stimulate corresponding goals or actions depending on the adopted composition pattern. In other words, system requirements are linked to user requirements (of course only the quality requirements), and are traced to determine the satisfaction or denial of them.

3.4.3 Modularity and Reusability

In the design and development of an adaptation manager, modularity and reusability are important. Modularity eases the development of the adaptation processes, and also enables the adaptation manager to use external pre-built modules like policy engines for implementing processes. On the other hand, reusability enables us to use patterns for designing the adaptation model/mechanism and common modules for implementing them.

To address these two concerns, QFeam represents several composition patterns for designing the adaptation model and corresponding mechanisms. Moreover, in implementing the adaptation manager, component frameworks and middleware-based management frameworks can be utilized. In this thesis, Java enterprise beans and JMX are used for this purpose. A framework called StarMX has been developed in the University of Waterloo STAR lab as an infrastructure for building an adaptation manager and communicating with the underlying adaptable software [13].

3.5 Design Metaphors

In the proposed framework, some ideas from other related domains have been used. Two notable domains are robotics, particularly behavior-based robotics, and game theory.

3.5.1 Behavior-Based Robotics

Due to the similarity of the focused problem with the Action Selection Problem (ASP) in robotics and autonomous agents, several ideas from these disciplines can be adopted in the self-adaptive research area. In particular, the idea of behavior-based robotics [9] seems promising for this purpose. However, the problem in software is generally more complex than in robotics. Software systems, especially large-scale distributed applications, have much more attributes and effectors, and are often more complex than mobile robots. Some researchers believe that the complexity of software in comparison with most systems in mature engineering disciplines can be attributed to the nature of software systems: the notable difference is that their behaviors do not obey descriptive physical laws [34].

Many of existing architectures for the adaptation engine in self-adaptive software uses the Sensor-Plan-Act (SPA) model, used extensively in building traditional robotic systems. In these systems, events are collected, analyzed, and fused to update the domain model (i.e., world model). The system then plans its strategy in the new situation. However, the idea of behavior-based robotics is to use distributed specialized task-achieving modules, called behaviors, and to apply command fusion instead of sensor fusion [9]. In this way, there is no need to develop, maintain, and extend a coherent monolithic model of the adaptable software and its context. Gat argues that building such a model is one of the main problems of the Sense-Plan-Act schema used in most traditional robots [59]. The successful experiences in building behavior-based robots motivated us to apply this idea to self-adaptive software. This thesis borrows the idea of *action fusion* to employ a goal-driven approach in software systems. Behaviors in robotics are mapped and extended to goals in this domain.

A goal-driven action selection mechanism can be realized in two general competitive and cooperative forms. In the former, the goals compete with each other in order to select the next action, whereas in the latter, the preferences of goals are combined or fused to determine what to do next. Arkin discusses variant forms of cooperative and competitive mechanisms in behavior-based robotics [9]. In the cooperative category, Arkin names superpositioning (vector addition) as the most straightforward method, provided that it is feasible. In the competitive methods, arbitration is a way to select one goal (winner-takes-all), for example based on predefined priorities. The subsumption architecture basically employs this method [29]. A less autocratic method has been utilized by Maes in an activation network [125]. Maes argues that the lack of explicit goals and goal-handling capability in autonomous agents leads to significant limitations in their operation [125]. Notably, when an agent does not have goals, every situation has to carry complete information for deciding the next most appropriate action. Maes proposes an activation network for actions, in order to facilitate dynamic action selection based on stimulated goals or actions.

The Distributed Architecture for Mobile Navigation (DAMN) uses a more democratic way by adopting a voting mechanism [173]. This method, and generally the voting-based mechanisms, can be arguably placed into the competitive category by Arkin. The social choice methods and voting games are well-known in cooperative game theory, and are used for combining decisions made by agents [45]. DAMN, introduced by Rosenblatt, realizes the first level of behaviors in the subsumption architecture [173]. DAMN uses a voting mechanism for command fusion, regarding the safety behaviors for the turn and speed of the mobile robot. The beauty of DAMN's design is that the deliberative and reactive components of the architecture can operate at the same level, and it is scalable due to its lack of hierarchy.

3.5.2 Collective Decision-making and Mechanism Design

Game theory and mechanism design have been widely used in modeling and analyzing multi-agent systems. Mechanism design seems more helpful for engineering an adaptation manager, because it is, in a sense, the inverse of game theory. The mechanism design objective can be described as "Given desired norms of behavior by a set of agents, design a game in which the desired outcome is the only rational behavior by the agents." [151]

In the domain of self-adaptive software, some researchers have been tried to benefit from game theory in cases where several applications have common resources or interests (e.g., applications running on a data center). In this thesis, cooperative game theory and in particular voting games are employed in the proposed framework, as collective decision-making mechanisms. Due to differences in the nature of quality goals, they can be modeled as independent agents which cooperate in an adaptation manager to satisfy the adaptation requirements in the best possible way. The goal-ensemble mechanism, proposed in this thesis, is inspired by this idea and is applied to two case studies. Interestingly, after introducing this idea, it was found out that the possibility of using goals as decision-makers had been briefly noted before in a game theory book, without being explored in an application [201].

The successful application of voting in behavior-based robotics (e.g., DAMN system) suggests the usefulness of this idea in the context of self-adaptive software systems. Here the plan is the modeling of quality goals as voting agents, and the actions as candidates for designing the adaptation mechanism. This approach has the following benefits: i) applicability for local goals, global goals, and for several applications, ii) independence of the decision-making algorithm of each goal, and iii) applicability in systems with several deciding processes, not necessarily goal-based.

3.6 Summary

This chapter introduced the QFeam framework for engineering an adaptation manager. The framework aims at facilitating the design and development of an adaptation manager through two key phases of modeling the problem space and mechanism design. Quality requirements play a key role in defining this model, due to the fact that self- * properties and adaptation requirements are strongly related to these requirements. Among the adaptation processes the deciding and partially detecting processes are targeted, and among the software engineering processes testing is not extensively covered by QFeam.

A key consideration in QFeam is that the adaptation model is embedded into the adaptation manager, and is utilized at run-time. The models used in developing software systems are not entirely applicable as-is for this purpose. For example, the models in Requirements Engineering (RE) are helpful, but do not completely fit to this problem. For this reason, the QFeam framework gets help from metaphors in behavior-based robotics and game theory. The following two chapters, Chapter 4 and 5, elaborate the main processes of QFeam in detail, and the implementation will be discussed in Chapter 6.

Chapter 4

Building Run-Time Adaptation Model

"*Models are not right or wrong; they are more or less useful.*"
Martin Fowler, "Analysis patterns"

One of the major objectives of the proposed framework, QFeam, is to incorporate a quality-based model into the adaptation manager to be used by an associated mechanism at run-time. This chapter details the modeling phase of the proposed quality-driven framework for engineering the adaptation manager. The main focus of this chapter is on capturing the essential concepts of the problem space, and the process of building the adaptation model. The final concrete model is constructed by adopting a composition pattern and employing an adaptation mechanism accordingly. Several composition patterns are discussed in this chapter. In the next chapter, a novel concrete model is introduced based on one of these patterns.

The valuable experiences, well-established methodologies and models in Requirements Engineering (RE)– such as Tropos [28] and KAOS [46]– and robotics– such as activation network [125] and DAMN [173]– are helpful for our purpose. In this chapter, various ideas are borrowed from these efforts and mapped into the adaptation modeling process.

4.1 Notations

Before going through the modeling process and elaborating the adaptation model in QFeam, it is better to briefly review the key notations in this chapter. The following is a list of the main notations used:

- G denotes the set of adaptation goals.
- AT denotes the set of all attributes.
- AT^{self} and $AT^{context}$ denote self and context attributes, respectively.
- $MonVars$ and $ConVars$ denote monitored and controlled variables, respectively.
- $MonVars^{self}$, $ConVars^{self}$, and $MonVars^{context}$ denote self and context monitored and controlled variables, respectively.
- AC denotes the set of adaptation actions.
- VP denotes the set of variation points in adaptable software.
- V denotes the set of variants in adaptable software.
- $U(.)$ denotes the utility function, which for instance measures an action impact.

These elements are elaborated with some examples in the following sections, along with other associated auxiliary parameters and variables.

4.2 Modeling Process

Figure 4.1 illustrates the proposed process in QFeam for modeling the entities of the problem space, and specifically the three conceptual subspaces for goals, attributes and actions. The modeling process consists of the three following major phases:

- *Adaptation requirements analysis*, in which key entities of the adaptation requirements are elicited. These entities are extracted in each space from artifacts and stakeholders.
- *Modeling the adaptation problem space entities*, in which the identified entities in each conceptual space are specified and modeled. This phase focuses on intra-space modeling.
- *Composing the adaptation model*, in which a composition pattern is applied to specify the dependencies between entities in conceptual spaces. The dependencies can be preferences, in which the process will be a type of preference elicitation. This phase builds the inter-space links to compose the adaptation model based on the conceptual spaces.

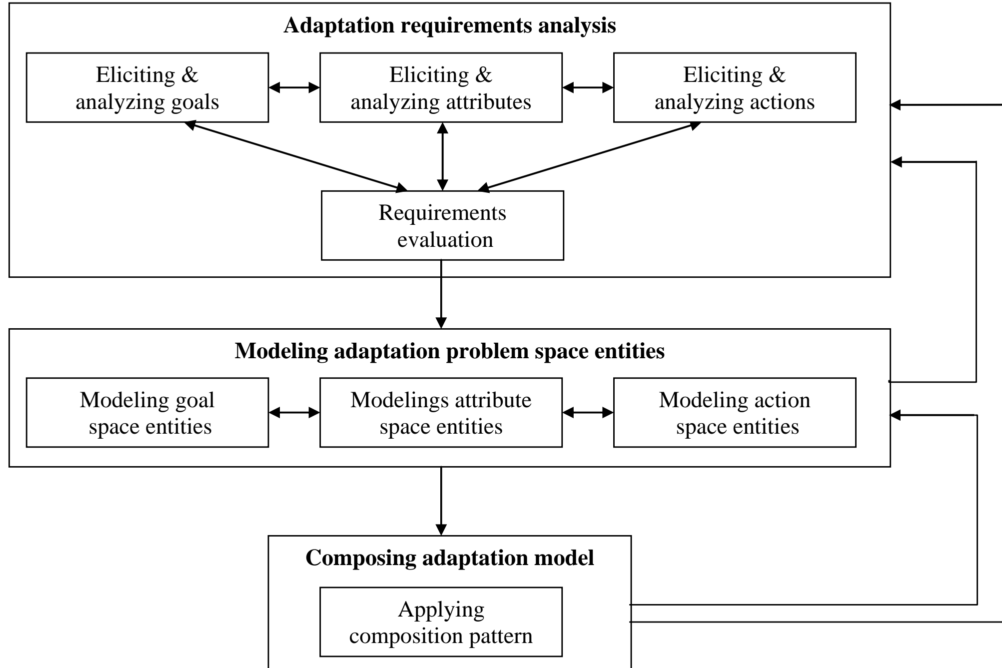

Figure 4.1: QFeam modeling process

The order of eliciting and modeling entities in each problem subspace is debatable. This is mainly because of the dependencies among these spaces, which are defined in the third phase. However, due to the similarities of this process with well-established goal-driven RE approaches, for instance KAOS presented by van Lamsweerde [211], the recommended starting point is the goal space. Next, it makes sense to select the attribute space, mainly due to its strong links with adaptable software artifacts, and its key role in elaborating requirements (i.e., ultimately satisfying goals). The distinction between the QFeam modeling process and goal-driven RE approaches is the focus on building a run-time model. Furthermore, the spaces in the adaptation model cannot be mapped completely to those in goal-driven approaches.

The bidirectional arrows between processes in each phase, and feedback from later phases indicate possible iterations. It should be noted that QFeam is a generic framework applied to mission-critical systems, and is more concerned with the deciding process (and partly the detecting process). Therefore, although the proposed modeling process has been practiced on several case studies, it still needs improvement.

4.3 Adaptation Requirements Analysis

Although some research efforts have tried to address functional requirements and emergent properties for adaptation, this area still needs much more work to be realizable in real-world applications. Therefore, as emphasized before, this thesis mainly focuses on quality requirements. Moreover, adaptation actions, which make changes at run-time, do not introduce new functional properties to adaptable software. This is an important assumption in QFeam and in the empirical studies of this thesis.

The main tasks in requirements analysis are domain understanding and requirements elicitation. These tasks in RE are broadly categorized into artifact-driven and stakeholder-driven approaches [211]. For the adaptation manager, these two approaches, which are not orthogonal, are also applicable and in practice both are utilized:

- *Artifact-driven techniques* rely on artifacts related to quality requirements and Quality of Service (QoS). In enterprise and service-oriented applications, SLA is a significant artifact for adaptation requirements elicitation. SLOs are normally an appropriate source for goal and attribute identification, which is elaborated further in the following sections.

  Because the adaptation manager is designed and developed for either an existing or a developing adaptable software, it is quite possible that adaptable software requirements artifacts are accessible. If goal models have already been developed with any modeling language (e.g., KAOS [46]), the adaptation goals, attributes, and actions can often be selected with less effort than by eliciting them directly from other sources. However, because the adaptation model is used at run-time, the development-time goal model needs to be simplified or modified.

- *Stakeholder-driven techniques* rely on interaction with stakeholders and eliciting what they expect from the system. These requirements may not be in SLA and other artifacts, but play a critical role in the degree of successfulness and user satisfaction. For instance, in the news application example, people always want to get contents (text, image and video) in high quality. Moreover, the opinion of domain experts is also extremely important in extracting adaptation requirements. The notion of policy is a key point in engineering the adaptation manager, which is not discussed in RE. Domain experts help us to extract and formulate policies. Later, we see the relation of policy and requirements statements.

  In this approach, the first step is stakeholder analysis. The stakeholder is defined as "a person or organization who influences a systems requirements or who is impacted by that system" [63]. While in conventional software engineering, all roles that impact the software development and operation are taken into account (e.g., tester,

and project manager), in QFeam only the operation-time stakeholders are considered. Thus, the typical roles are end-user, business owner, and administrator.

In the analysis phase, concepts in each conceptual space are elicited. These concepts are either descriptive or prescriptive. In the prescriptive part, goals play a key role as the main drivers of adaptation. The descriptive part is analyzed to determine the attributes and actions explained in the domain. Besides these conceptual spaces, other auxiliary concepts or models may be borrowed to complement these spaces. An example is variability information which is particularly helpful in analyzing the action space. Note that this auxiliary information may or may not exist in the final concrete adaptation model. In fact, it depends on how the adaptation mechanism is designed and implemented.

The previously discussed W5H1 questions can help us in analyzing adaptation requirements, and especially in eliciting key entities in each space. For example, the "why" question is strongly related to the goal space, due to the fact that goals are prescriptive statements that navigate the adaptation mechanism.

4.3.1 Eliciting and Analyzing Goals

Adaptation goals G in the goal space are prescriptive statements that should be satisfied by the adaptation manager. These goals capture different quality requirements for adaptable software. In the news application example, instances of such goals are "maximize system performance" and "minimize response time".

Since the expected behavior of self-adaptive software is normally specified by self- * properties, these properties also play an important role in determining the target goals. Self- * properties are related to software quality factors [180]. This relationship can benefit goal identification in two ways:

- Quality models (e.g., ISO/IEC 9126 [84]), the previously collected knowledge in RE [43], and other related engineering fields (such as performance engineering [137]), provide a taxonomy of quality factors and subfactors which helps us identify quality goals from both artifacts and stakeholders. This issue helps in reusing domain-independent knowledge, as in RE [211].
- Available adaptable software goal models can be used for extracting adaptation goals [114]. Furthermore, SLA, which is used for specifying QoS statements is useful for this purpose. Each SLO in a SLA can be roughly mapped to a prescriptive requirements statement. This statement is used for goal identification. Generally, SLO conditions are clearly specified, which leads to the satisfaction criterion for the corresponding goal. Therefore, we can say that in the case of having SLOs for a system, $SLO_i \mapsto g_j$.

Although adaptation goals belong to the non-functional category, they are not necessarily soft goals, as Van Lamsweerde argues [211]. He argues that 'ility' goals, which originate from quality goals, are mostly soft goals. Statements such as "improve", "increase", "maximize", "minimize", and "reduce" all represent soft goals. Soft goals cannot be clearly evaluated as being satisfied or denied. This is why the term satisficied (originally from Herbert Simon [194]) is used for them [43]. Soft adaptation goals need to be decomposed into other types. Another goal type discussed in requirements engineering is the probabilistic goal [117]. The satisfaction of this goal type, unlike soft goals, can be determined by a partial degree (e.g., 80% of the time).

Goal elicitation is initiated by the "why" question to determine the main motivation for adaptation. These high-level goals are often soft goals. However, as will be discussed in the modeling section later, goal decomposition proceeds with the "how" question. It refers to how a goal can be satisfied by the subgoals. These subgoals will end up at the bottom level as probabilistic or behavioral goals. These bottom-level goals are called *leaf goals*.

For example, in the news application one of the responses to the "why" question is "maximize performance", which is a soft goal. At a lower level, for this goal we ask the "how" question for this subgoal,which results in "minimize response time" as a subgoal. This may result in prescribing a certain threshold for the response time of a specific method (e.g., search method) in the application.

As noted in Chapter 2, policy-based management is one of the common approaches in realizing adaptation processes. Policy is basically a descriptive statement, mostly captured by domain experts, system administrators, and operators. Among three main types of policies- goal, utility, and action [98] - goal policies, as the name suggests, can be mapped to goal space entities. Of course, goals in RE are broader than goal policies, because behavioral goals do not merely specify the target condition. But since we mostly deal with quality goals in adaptation, goal policies can be matched with adaptation goals. Utility policies, by definition, are more closely related to the attribute space and configuration states. However, utility values can also be assigned to goals, similar to conventional utility policies. Utility definition is discussed later in this chapter.

4.3.2 Eliciting and Analyzing Attributes

For the monitoring and detecting processes, and in order to essentially trace adaptation goals $g_i \in G$, domain attributes should be captured and tracked. Attributes AT represent measurable and quantifiable properties of adaptable software. In QFeam, the assumption is that $at_i \in AT$ are generated by the monitoring process from raw data collected from sensors.

Attributes are system variables which may be controllable or non-controllable. In the

four-variable model [155], these are called controlled and monitored variables, respectively. In this chapter, $MonVars$ and $ConVars$ denote monitored and controlled variables respectively. They can be also called control input and disturbance, as in control engineering [101, 71]. One may ask whether the adaptation manager only needs to know about monitored variables. In order to keep track of the adaptable software status, the adaptation manager needs to know the current configuration and the values of controlled variables. For instance, the number of maximum threads in a thread pool, or the size of a buffer, which can be modified dynamically, needs to be read in order to make a proper decision. These controlled variables may be stored in an auxiliary model, which can be called the configuration model.

From another point of view, attributes belong to either self or context. The set of attributes AT should cover both sets of self and context attributes:

$$AT = AT^{self} \bigcup AT^{context} \tag{4.1}$$

Self attributes in AT^{self} consist of all monitored and controlled variables in the adaptable software:

$$AT^{self} = MonVars^{self} \bigcup ConVars^{self} \tag{4.2}$$

Because the adaptation manager sets out to satisfy quality goals, self attributes are mostly related to goals like performance, security, and reliability. Furthermore, configuration properties, which may be controlled at run-time, are also important for the adaptation manager. Some common self attributes are as follows: i) system configuration attributes, such as the size of a buffer, or the number of threads in a thread pool, ii) errors, faults, failures, and availability of components and services, iii) performance attributes such as response time, throughput, and resource utilization, and iv) security attributes such as attacked services or components, and statistical data about malicious traffic on specific ports.

Some of these attributes such as availability belong to $MonVars$ and some like buffer size are in $ConVars$ (providing the buffer size is controllable at run-time). In the news application, the types of delivered content (text, image, video, and their combinations), the content quality (high/low quality), the size of the items (small/large image), as well as response time and availability properties of components are examples of self attributes.

Context is also called the environment or world in some studies [101]. Therefore, context attributes may belong to any element of the surrounding environment, human agents, and other interacting systems that impact self-adaptive software. Here, a notable assumption is that we cannot control context attributes. Therefore, context attributes do not cover controlled variables:

$$AT^{context} = MonVars^{context} \tag{4.3}$$

For example, the number of active users, the inter-arrival time of requests, and the service time of requests in the news application are examples of context attributes. Generally, the following context variables may be of interest to the adaptation manager:

- User-related attributes such as user behavior, the number of users, traffic probability distribution, usage time, and user location
- Domain-specific attributes, such as temperature
- Related systems' status such as the availability of an external web service

Identifying attributes, both self and context, is strongly based on the identified adaptation goals. For example, in the "minimize average end-to-end response time" soft goal, average end-to-end response time is an obvious attribute to be monitored by the adaptation manager. Of course, this is not the case for all goals. For example, "improve user satisfaction" does not lead to a directly measurable and quantifiable attribute. Goal decomposition and refinement can help effectively in this case. Note that attributes are bounded by sensors provided by the adaptable software. However, sometimes instrumenting additional sensors to monitor required attributes may be necessary. Again, quality models can help us in identifying attributes that are needed for evaluating the satisfaction criteria of goals.

Sometimes, static and/or dynamic analysis of adaptable software also aids us in identifying potential adaptation attributes. In fact, this task can locate all significant variables in the adaptable software that are required to be monitored. However, this task belongs more to the process of engineering or re-engineering the adaptable software rather than to the adaptation manager. Therefore, further details on this issue are out of the scope of this thesis.

4.3.3 Eliciting and Analyzing Actions

A central tenet of adaptation is the concept of change. Adaptation changes or actions basically aim to satisfy goals. As discussed briefly in Section 2.1.1, adaptation changes can be viewed in the context of software evolution. Change as an evolution vehicle of software often implies an off-line process in which the system evolves through a number of releases. However, in adaptation, dynamic change at run-time is emphasized and in this sense resembles *dynamic evolution* [103].

The adaptation change type is related to the nature of the change. The taxonomy of evolution takes into account two broad categories, namely structural and semantic changes [31]. Since only quality requirements are considered in this thesis, all changes are semantic-preserving. However, all adaptation changes cannot fit into the structural category. The categorization proposed by McKinley et al. [136] is more meaningful for run-time and dynamic adaptation changes. In this view, changes are either compositional or based on parameter adaptation. Compositional changes include any kind of addition, deletion, and swapping of software artifacts, while parameter adaptation involves changing system parameters, such as the buffer size. Changes can also be categorized from internal and external points of view [137], namely: i) user perceived changes such as the degrading or upgrading of service levels, and ii) internal changes, such as caching, which are not directly observable by users. The former is both stakeholder and artifact-driven, while the latter is mostly artifact-driven. In short, a list of important adaptation actions, in line with the effectors list in Table 2.1, is as follows:

- Resource management, which compensates resource shortage (e.g., CPU), or balances the cost and benefit of resources in satisfying adaptation goals
- Admission control which balances user traffic, the number of requests, or request inter-arrival time
- Swapping actions, which swap and switch components, methods, algorithms
- Architectural changes, which change connectors or add and remove components
- Tuning self parameters, which changes controlled self variables (e.g., increase a buffer size)
- Service level and feature management, which change services or functionalities for users, depending on goal conditions or attribute states (e.g., disabling some services under a high load or in non-secure situations)

As seen in the above list, these actions can directly change self attributes. However, it is possible to impact context variables indirectly by some actions as well. For instance, using admission control actions, it is possible to reduce the number of users or user requests, both of which are context monitored variables.

As discussed before, controlled variables $ConVars$ represent adaptable software properties that can be changed toward adaptation. In QFeam, specifying these variables and their alternatives in adaptable software is achieved by using a structure similar to the variability model in *Software Product Lines (SPL)* [210]. In SPL, the domain includes variation points that can have several alternative variants. Therefore,

$$ConVars^{self} \mapsto VP \tag{4.4}$$

where each vp^i is associated with two or more variants v^i_j. Each v^i_j can be assigned to a vp^i and the combination of all the tuples (vp^i, v^i_j) for the entire set of defines a configuration. This issue is elaborated further in the modeling section. In fact, adaptation actions manage variability in variation points. Therefore, the identified variation points, which are mappings of control variables, and their corresponding variants aid us in identifying actions. In Section 4.4.3, we see how this structure is applied to modeling actions.

Similar to attributes, adaptation goals can be the starting point for identifying actions. Van Lamsweerde argues that in goal-oriented RE, deriving operations (mostly equal to actions in this context) is based on goal fluents [211]. Indeed, this is often a straightforward task in the case of behavioral and functional goals . For example, in a goal like "LockDoorsIfMoving" in a car, we can easily identify lock and move actions. However, this is not easy for most quality and soft goals. For instance, "Improve availability" does not give a clue about how it can be satisfied. For these goals, the role of domain experts is significant in identifying actions.

Of course, this issue is also related to instrumenting effectors in adaptable software which is not covered in this thesis. However, similar to attributes, adaptation actions are bounded by effectors provided by the adaptable software in the case of engineering the adaptation manager after the adaptable software. Again, instrumenting new effectors to apply the required changes for satisfying goals may be needed. Another notable point about actions is that for each variation point, there are at least two actions for switching between variants. Sometimes they are called action and counter-action. For example, for the goal "LockDoorsIfMoving", in addition to the lock and move actions, the unlock and stop actions are also required as counter-actions.

4.3.4 Requirements Evaluation

In RE, requirements evaluation is initially performed after the primary requirements elicitation. Generally, this phase consists of analyzing alternative options, risk analysis, finding conflicting concerns, and requirements prioritization [211]. For the adaptation model, analyzing alternative options is certainly a part of the task of evaluating the action space, in order to find variants and to estimate their impacts on goals. This also implicitly includes impact analysis, which is an important part of identifying and instrumenting effectors. Risk analysis in a systematic way is not one of the main concerns of this thesis. However, diagrams like the obstacle diagram in some goal-driven methods can help in designing the adaptation mechanism, and particularly the deciding and acting processes.

Finding conflicts in the goal and action spaces is an important issue. Most self-adaptive systems have several adaptation goals, and most often these goals are in conflict. Therefore, taking these conflicts into account in the goal model, and later in adaptation mechanism design, is quite important. Moreover, actions may have conflicts over changing the same variation points, or over utilizing resources for accomplishing a task or workflow. An obvious example is an action and its counter-action, which may be decided to be taken at the same time or immediately after each other.

Another significant task in requirements evaluation is prioritization, which is particularly important for goals. Goal priorities are of special importance in conflict resolution and in the deciding process of the adaptation mechanism. Prioritization can start with high-level goals as they are elicited and analyzed, but it needs to be followed down to the lower levels. Common practices for prioritization in RE can also be applied for adaptation requirements. A suggested method is using the Analytic Hierarchy Process (AHP) by pairwise comparison of goals [24]. In this way, it is possible to check the consistency ratio of both the comparisons and priorities. Furthermore, in artifact-driven evaluation, the designated penalties in SLA can also lead to useful information in goal prioritization.

4.4 Modeling Adaptation Problem Space Entities

The three key conceptual spaces are the building blocks of the adaptation model. The identified entities in each space need to be specified and modeled. The requirements modeling practices are certainly helpful, keeping in mind that the adaptation model is embedded in the adaptation manager. The run-time use of this model hinders some aspects such as complexity.

The adaptation model can be seen from the two following points of view:

- The first view is the four variable model, in which system and software requirements are defined based on monitored, controlled, input and output variables [155]. In a self-adaptive software system, the adaptable software and its environment (context) are the environment from the adaptation manager viewpoint. Monitored and controlled variables are quantities that are measured and controlled by instrumented sensors and effectors, while input and output variables are data items which are passed between the manager and the sensors/effectors. Parnas and Madey argue that software requirements can be defined as $SoftwareReq \subseteq I \times O$, where I and O are input and output variables [155]. Adaptation requirements also capture these variables through sensors and effectors in the attribute and action conceptual spaces.

  One may argue that controlled variables could be used instead of actions. In fact, the action space includes controlled variables, but it also represents how these variables

can be changed. The architectures proposed for adaptation commonly use the acting process to manipulate the controlled variables. Therefore, the notion of action is used at a higher level of abstraction than that of controlled variables. More specifically, an action switches between variants of a variation point.

- The second view of the adaptation model is based on two types of requirements statements, namely, descriptive and prescriptive. All the facts and information about the domain that states adaptation manager looks like and how it behaves are captured by descriptive statements. Attribute and action spaces, besides optional auxiliary structural and behavioral models, fall under the umbrella of descriptive concepts and models. On the other hand, the goal space captures what the adaptable software should be and how it should behave (here only for quality requirements), and is therefore prescriptive.

These two views do not contradict each other, and QFeam benefits from both of them. Note that there is not necessarily a model connecting all entities in each conceptual space. For example, the framework does not enforce having an action model linking all the actions.

4.4.1 Modeling Goal Space Entities

Goals elicited in the analysis phase need to be specified by their properties, be decomposed/refined, and be incorporated into a goal model. Note that entities in the goal space are not necessarily part of a single model. There may be several goal hierarchies for each high-level quality goal. However, depending on how the adaptation mechanism needs to be operated, a dummy goal may be added to create a single hierarchy of goals. Also, there can be several goal models, when several adaptation mangers are available in the system.

An important property of a goal is its *activation criteria*. This denotes how the adaptation manager can determine whether a goal has been satisfied. This property may be also called fit criterion [211] or satisfaction criterion. In QFeam a goal is activated when it is not satisfied. This contrast is due to the fact that in this framework, active goals trigger actions to be satisfied instead of goal models in conventional requirements engineering, in which actions are specified to satisfy the goals. In Section 4.5, when the composition patterns are introduced, we see the importance of this notion. The activation criteria are related to the goal type. For soft goals, instead of goal satisfaction, goal satisficing is evaluated. Other goal types in the adaptation have a logical or threshold satisfaction criteria (e.g., having a response time less than 500 milliseconds). A parameter in using goals at run-time is how often their satisfaction should be evaluated. This parameter may be different for each goal depending on its probability of change, its priority, and the attributes which activate the goal. This is specified in adaptation mechanism design, and may be changed at run-time.

The selection of goals, and the design of a structure relating them depends on the design of the detecting process, which determines each goal's activation status. A richer goal structure surely needs a more complex and probably more time-consuming detecting process. Add to this the complexity of the action selection mechanism, which is required to incorporate goals into the deciding process.

Different quality goals may exist in a goal space, which are decomposed to lower level goals. Stakeholders often start to articulate high-level quality goals by specifying the desired behaviors of the system (e.g., achieving an acceptable level of performance). At this level, the goal type is basically soft. These goals are decomposed into low-level goals, which are more likely to be related directly to measurable attributes; "minimize response time" and "having 80% resource utilization" are two examples of these goals. It is true to say that in a goal hierarchy, soft goals are finally decomposed into goals whose their activation criteria can be evaluated (i.e., quantified). In other words, the hierarchy moves down from qualitative to quantitative activation criteria. These lower level goals, especially *leaf goals* at the bottom of a goal hierarchy, are particularly important in composing the entire adaptation model.

Another criterion for decomposing goals is the level of system they belong to. System level goals can be decomposed into local level goals related to subsystems, components and methods. For instance, in the news application, the availability of the entire system can be decomposed into the availability of each server- web server, application server, and database server- and consequently into the availability of the components deployed on each server, as well as on the servers themselves. These levels may be all quantitative, but the detecting process may need more precise information to select an appropriate action.

Sub-goals derived from a specific goal can be related to each other by different logical operators such as AND, OR, and XOR, similar to goal models in RE. These relations may or may not be used in the concrete run-time adaptation model, or may not be used by the adaptation mechanism at run-time. For example, "response time less than 500 ms at 80% of times", which could be a leaf goal, is traced and activated by the adaptation mechanism, but bottom-up reasoning to evaluate the activation criteria of higher level goals may not be needed. However, sometimes this reasoning may be applied for other purposes, such as informing administrators those goals that have been denied or satisfied.

Figure 4.2 depicts a partial goal hierarchy in the news application. The soft goal "maximize performance" is decomposed into "minimize response time" and "maximize throughput". "minimize response time", in turn, can be refined into "minimize end-to-end response time" and other response times (e.g., for EJB components deployed on an application server). The former goal can again be refined into "average response time less than 500 ms in 80% of times" or simply "average response time less than 500 ms".

Figure 4.3 depicts the goal meta-model. This includes the following information about goals:

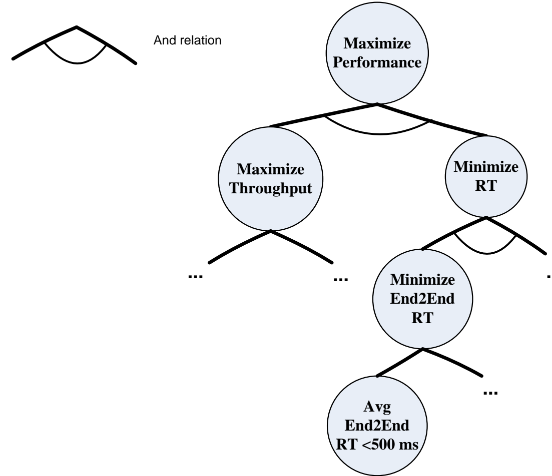

Figure 4.2: A partial goal hierarchy for the news application

- *Name*: This identifies each goal uniquely.
- *Priority*: This represents the importance of a goal in relation to other adaptation goals.
- *Tracing period* (optional): A property which specifies when a goal activation criteria should be evaluated. This period can be same as a global period of the entire adaptation process, called adaptation period T_{ad}. The goal may be activated by an event.
- *Level* (in a goal hierarchy): It may be required for relating goals to other entities in the attribute or action space. Particularly, leaf goals may be important in linking goal models to other spaces.

Activation criteria, which is an important piece of information about each goal, are represented by an association class between the goal and the attributes in the meta-model. This determines when a goal is satisfied or denied. Depending to the type of goal, this can be qualitatively or quantitatively determined. For a non-leaf goal g_i, the satisfaction statement is

$$\{SubGoal_1, ..., SubGoal_j\} \models g_i \tag{4.5}$$

while for leaf goals, we simply have

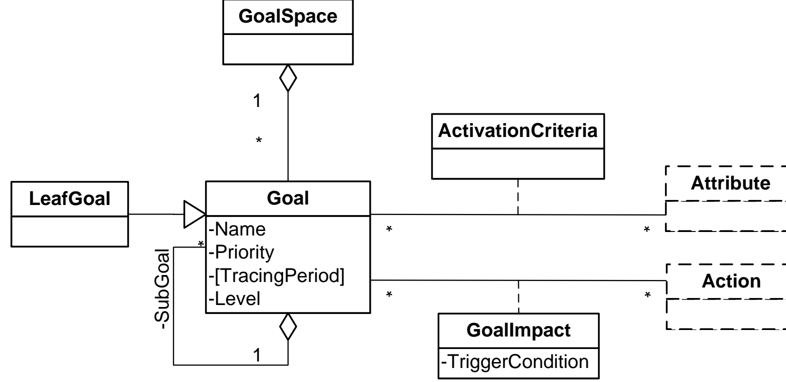

Figure 4.3: Goal meta-model in quality-driven adaptation framework

$$ActivationCriteria_i \models \neg g_i \tag{4.6}$$

The *GoalImpact* association class is discussed in Section 4.4.3.

4.4.2 Modeling Attribute Space Entities

An attribute model may consist of various models such as a configuration model, a state model (self states), and a context model. These models use different combinations of self and context attributes to represent the system status. If each state represents a set of tuples $s_i = \{(at_j, value_j)\}$, different spaces can be defined based on the category of attributes, such as a general state space:

$$StateSpace = S \times S = \{(s_i, s_j) | s_i = \{(at_k, value_k)\}, at_k \in AT\} \tag{4.7}$$

that can be based on only the the set of AT^{self} as well, or a context space,

$$ContextSpace = S \times S = \{(s_i, s_j) | s_i = \{(at_k, value_k)\}, at_k \in AT^{context}\} \tag{4.8}$$

Based on each of these spaces, we can define directed graphs that include the states, as $StateModel$, or contexts, as $ContextModel$, and transitions among them. The set of transitions involves events such as taken adaptation actions. A configuration model can be also defined based on controlled variables, but we will see later that a definition using variation points is more useful for modeling actions. The choice of a model depends on

the adopted composition pattern, which is discussed in the next section. For instance, the attribute model may play a role in defining actions, simply as transitions between states. Note that incorporating a concrete attribute model, which relates attributes together, into an adaptation model is not necessary. Attributes can be modeled as individual entities that are connected to goals and actions. An example of such a case will be discussed in the next chapter, in which attributes are used to activate goals or trigger actions, but they are not combined to represent a state.

Figure 4.4 illustrates the attribute meta-model in the proposed framework. The attribute model includes the following information about attributes:

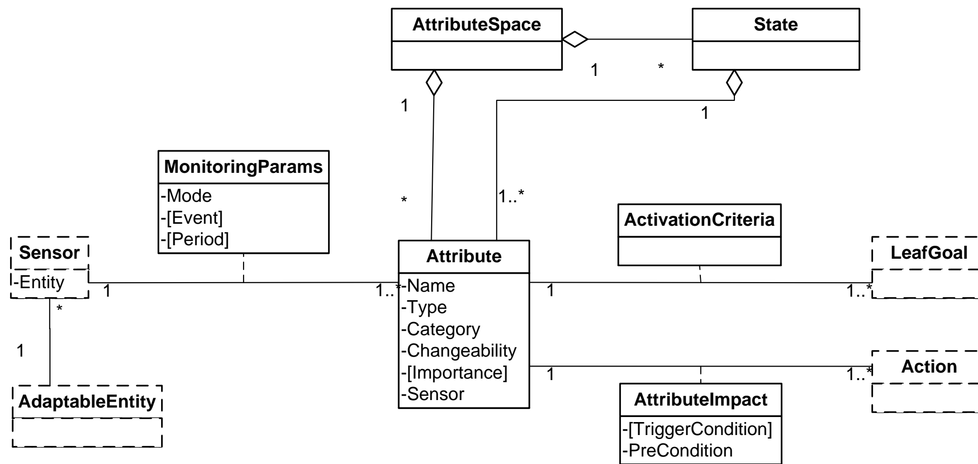

Figure 4.4: Attribute meta-model in quality-driven adaptation framework

- *Name*: This identifies an attribute uniquely.
- *Type*: This is an attribute type such as time.
- *Category*: This determines whether the attribute belongs to self or context.
- *Changeability*: This determines whether the attribute is monitored or controlled. Note that QFeam assumes a context variable cannot be controlled directly. The context variables may be indirectly affected; for example, by admission control, the adaptation manager may drop some service requests.
- *Importance* (optional): Based on the level of awareness, the adaptation manager may need to monitor different sets of attributes. It can take any of the values in $\{critical, regular, optional\}$.

- *Sensor*: This property identifies to which specific sensor this attribute belongs. Having the sensor, the entity or level can be specified: i.e.,. system level (global level), a specific component, method, or subsystem. One attribute can be instantiated for several levels. For example, average response time may be monitored for the entire system, a single component, or a method.

Moreover, *MonitoringParams* is defined as an association class for the link between an attribute and a sensor. The sensing mode (synchronous or asynchronous events), monitoring period, and other required specifications are properties of this class. The *AttributeImpact* association class is described in the following section.

4.4.3 Modeling Action Space Entities

The actions in *AC* can be modeled in various ways, depending on how the adaptation mechanism needs to use them at run-time. However, some fundamental aspects are common among these models, including the preconditions of an action, where it will be applied and what it changes, and its impact on goals or attributes. Figure 4.5 illustrates the action meta-model in the proposed quality-driven adaptation framework.

Based on the given meta-model, the following properties of actions are specified:

- *Name*: This identifies an action uniquely.
- *Type*: This determines whether the action is atomic or composite.
- *Params* (optional): Some parameters may be required in switching between variants. For example, we can parameterize the feasibility condition of the action.
- *Effector* : This determines by which specific effector this action can applied. The effector, in turn, specifies the variation point $vp^{AdaptPoint}$ that can be changed by the action. If the action is atomic, there is only a single variation point, whereas for a composite action, there are several.
- *TargetVariant* v_{target}: The target variant that will be assigned to the adaptation point after applying the action.
- *SourceVariant(s)* v_{source} (optional): A property which shows possible source variants.

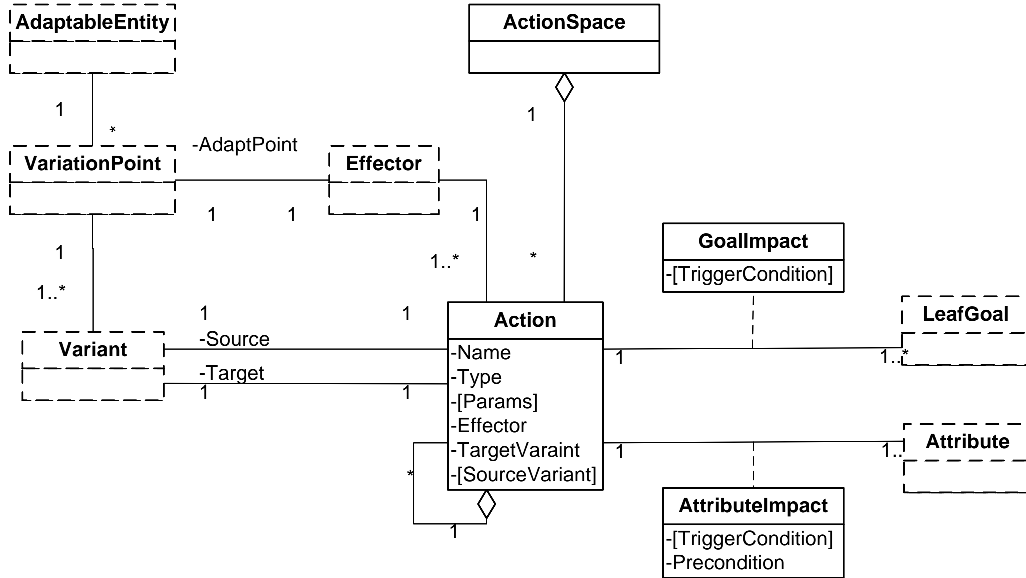

Figure 4.5: Action meta-model in quality-driven adaptation framework

Two association classes define the effect of other conceptual spaces on the action entities, namely *GoalImpact* and *AttributeImpact*, depending on the composition pattern. The impact need not be explicitly modeled, and it can be implicitly implied by other parameters such as goal preferences. An example of this case is discussed in the next chapter. Note that impact implies the postconditions of an action. Impact defines the following properties:

- PreCondition (only in the AttributeImpact): The necessary condition for triggering an action. It can be the source variant on which the action is applicable, or other domain properties. Satisfying this condition is not sufficient for executing an action. For example, video content should be provided for users before applying a disable-Video action. In a sense, PreCondition is the domain precondition regardless of the relation of the action with other spaces.

- TriggerCondition: The sufficient condition for triggering the action. It can be the activation (i.e., denial) of a goal or a criterion directly based on attributes. Note that if this condition is true, the action should be executed. For example, the condition can be "a very high user traffic" when "video content is provided" for the "disableVideo" action. We can say that $TriggerCondition \Longrightarrow PreCondition$. It means that if the TriggerCondition is true, PreCondition is also true. Depending on the design and adaptation mechanism, TriggerCondition can be solely used instead of defining a separate precondition.

Besides the above properties, additional data may be gathered and stored about actions. For instance, the history of action execution, the latest execution of an action, and the failure/success of previous executions.

An action can be modeled as *atomic* or *composite* changes to the adaptable software. An atomic action can be modeled as a transition between two variants, v_{source} and v_{target}, in a variation point $vp^{AdaptPoint}$:

$$ac_i : v_{source}^{AdaptPoint} \rightarrow v_{target}^{AdaptPoint} \tag{4.9}$$

where

$$\{(vp^{AdaptPoint}, v_{source}^{AdaptPoint})\} \subset Conf, \{(vp^{AdaptPoint}, v_{target}^{AdaptPoint})\} \subset Conf, vp^{AdaptPoint} \in VP$$

In this definition, action impact can be annotated on the action itself, or on the target variant. These impacts can be specified as labels in the SIG or i* model (e.g., labels like “hurts” or “++”), fuzzy labels (e.g., positive impact), or as cardinal values. The variability model is not a part of the action space, but it can be a base for defining the action model. Basically, it belongs to the optional auxiliary models presented in Chapter 3. If a variability model is kept as part of an adaptation model to be directly used at run-time, it can be seen as the utilization of a Dynamic Software Product Line (DSPL) [70]. The variability model can be formally described, for example using XVCL language [223].

Conf defines a valid configuration in the adaptable software of the form:

$$Conf = \{\bigcup_{i=1}^{|VP|} (vp^i, v_j^i) | vp^i \in VP, v_j^i \in V\} \tag{4.10}$$

such that each vp^i appears once. We can define the set of all valid configurations. However, we can define a configuration model to represent not only all the configurations but also the transitions among them as a $ConfModel$. Composite actions can be defined as transitions between configurations or system states, as suggested in other research efforts (e.g., [211]). A composite action is defined as:

$$ac_i : conf_{source} \rightarrow conf_{target} \tag{4.11}$$

where $conf_{source}$ and $conf_{target}$ are valid configurations in $ConfModel$. A composite action can be defined as a transition between states in a $StateModel$ or configurations in $ConfModel$. However, since the states include all of AT^{self} and $AT^{context}$, we may not determine the target state at run-time before the next round of monitoring. But, if an action is applied successfully, the target configuration is deterministic.

Accordingly, the impact of each action on goals is determined by the differences between the target and source variants. The impact can be formulated as a utility function

$$U(ac_l) = \sum_{i=1}^{|G|} u_i(v^j_{source}) - \sum_{i=1}^{|G|} u_i(v^j_{target}) \tag{4.12}$$

where $u_i(.)$ is the impact of each variant on each goal. Utility can be also calculated for composite actions as:

$$U(ac_l) = \sum_{i=1}^{|Conf|} u_i(conf_{source}) - \sum_{i=1}^{|Conf|} u_i(conf_{target}) \tag{4.13}$$

Formulating these impacts is not an easy and straightforward task. It can be done by static/dynamic analysis, and through some rigorous quality tests such as performance testing. This task is particularly significant in engineering adaptable software, which is out of scope of this thesis, even though it has been performed for the case studies in Chapter 6.

An important aspect of an action is its granularity. This is often discussed in terms of two general categories of coarse-grained and fine-grained changes. A remarkable number of well-known studies in the literature address coarse-grained changes such as architectural changes; to name a few [150, 57]. Although these changes can be highly effective, the problem is that they are hard to implement, test, and manage in complex large-scale distributed systems. This issue is much more difficult under unanticipated conditions. Fine-grained adaptation has been increasingly used, due to its lower cost, and also to the advent of supporting technologies (e.g., dynamic aspect composition). Parameter adaptation changes are fine-grained, but some resource management actions and the composition of dynamic aspects are also considered to fall into this category. For instance, in a recent effort, Grace *et al.* [157] use the fine-grained adaptation of aspect-oriented compositions. By fine-grained, they mean changing the cross-cutting concerns of a subset of entities [53, 66].

In another view, actions can be part of a conceptual hierarchy, as discussed in [178]. In this view, adaptation change is mapped to concepts in *activity theory*. Therefore, an adaptation change is an activity, broken down into a hierarchy of actions and operations. Example activities can be major self- * properties discussed in [97]. A sample set of actions and operations for performing these activities and their corresponding operations is shown in Table 4.1. For instance, in the news application, self-configuring can be realized using service degrading/upgrading for alternate quality levels of video and image [185]. Actions and operations can be chained together in order to form plans. For example, restarting a component may be planned as a sequence of storing the component state, undeploying the

component, redeploying it and restoring its state. The action hierarchy can be mapped to a goal hierarchy; see [178] for more details.

Activity	**Action**	**Operation**
Self-Configuring	Service level degrading/upgrading	Dynamic aspect weaving/unweaving Decomposing/Recomposing components Changing application configuration parameters
	Entity/Service switching	Dynamic aspect swapping Web service swapping Method swapping
	Entitiy/Service adding/removing	Decomposing/Recomposing components Changing user interface
	Changing architecture	Changing component composition Changing service composition
Self-Optimizing	Resource management	Changing deployment architecture Load balancing Resource provisioning and replicating Changing thread pool size Changing database connection pool size Changing KeepAliveTime
Self-Healing	Restarting component	Redeploying Rolling back
Self-Protecting	Isolating component	Decomposing component Component swapping

Table 4.1: A sample adaptation activity hierarchy

A metaphor for the activity hierarchy is the strategy-tactic relationship in military and politics. A strategy is a high-level plan to tackle a problem, while a tactic is a low-level action, mostly with local impact, to implement a part of a strategy. Strategies are often used to make a problem easier to analyze and solve. In this view, a strategy is mapped to an activity or action, while a tactic is related to an operation. Cheng *et al.* used this hierarchy in an adaptive application [41]. They utilized this view for adaptation in an example news management application with several objectives.

4.5 Composing the Adaptation Model

The last phase of building the adaptation model is composing the entities of the three conceptual spaces. Various composition patterns can be used for this purpose. The patterns specify which spaces need to be linked and exactly which entities of each space need to be related to each other. Three significant composition patterns are:

- *Goal-centric composition pattern*: In this pattern, as illustrated in Figure 4.6, the goal space plays the main role in navigating the adaptation processes. In particular, the detecting and deciding processes are heavily based on goals. Attributes are

linked to leaf goals based on the semantic dependencies. An obvious example is linking "Average end-to-end response time" to "Average end-to-end response time less than 500 ms". The links between goal and action entities are not similar to the ones connecting attributes to goals. These links are specified by different association classes in the discussed metamodels.

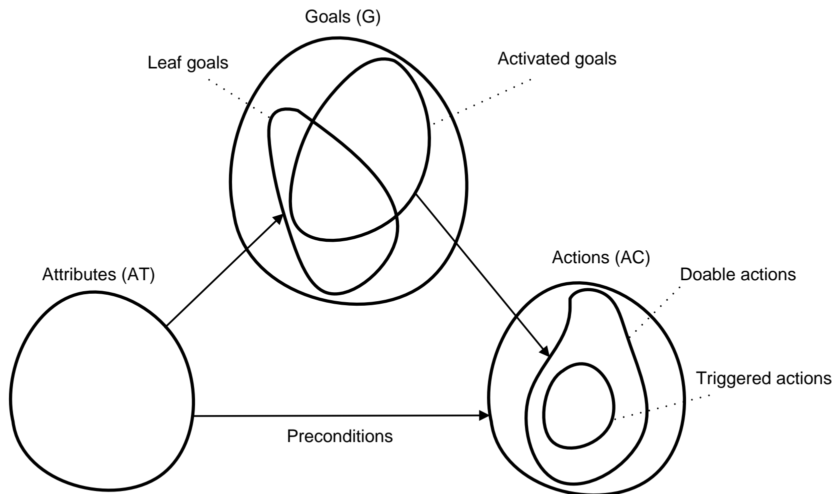

Figure 4.6: Goal-centric composition pattern

In short, goal-based requirements statements, and also goal policies, are the key tenets of the approach for building the adaptation model. However, the important point is that although goals are mediating between actions and attributes in this pattern, the model still needs links between attributes and actions. This is for the sake of defining preconditions.

- *Attribute-action-coupling composition pattern*: This pattern is not explicitly based on quality goals, as depicted in Figure 4.7. In fact, it mostly relies on action policies (e.g., in the general form of condition-action rules), which defines either low-level actions or a hierarchy of actions as in the strategy-tactic paradigm. Goals are either implicitly involved or appear in the form of behavioral goals. Behavioral goals are similar to action policies with clear-cut guidance and activation criteria.
- *Hybrid composition pattern*: This pattern combines the two previous patterns, as illustrated in Figure 4.8. Therefore, attributes can activate goals and trigger actions

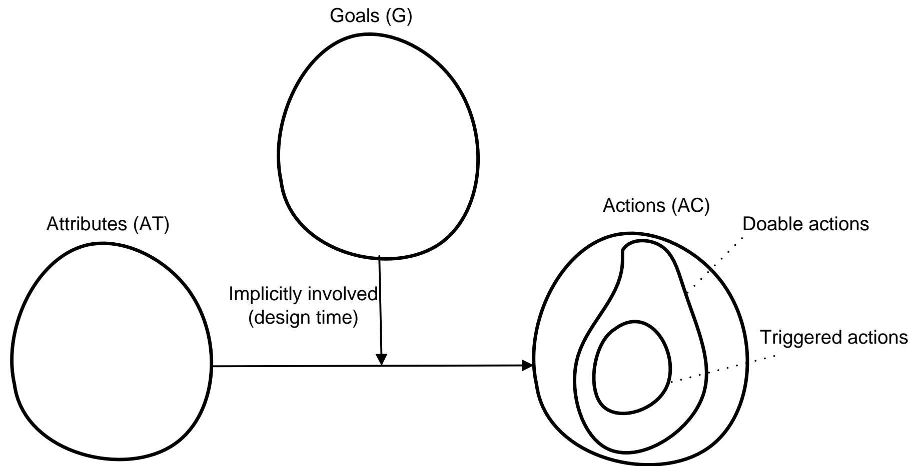

Figure 4.7: Attribute-action-coupling composition pattern

at the same time. In this case, a mechanism is required for fusing the effect of attributes and goals on actions. One option is that goals, due to their higher level view of the system, suppress the effect of attributes. But, another solution is to aggregate these effects. In the next chapter, these solutions will be elaborated further.

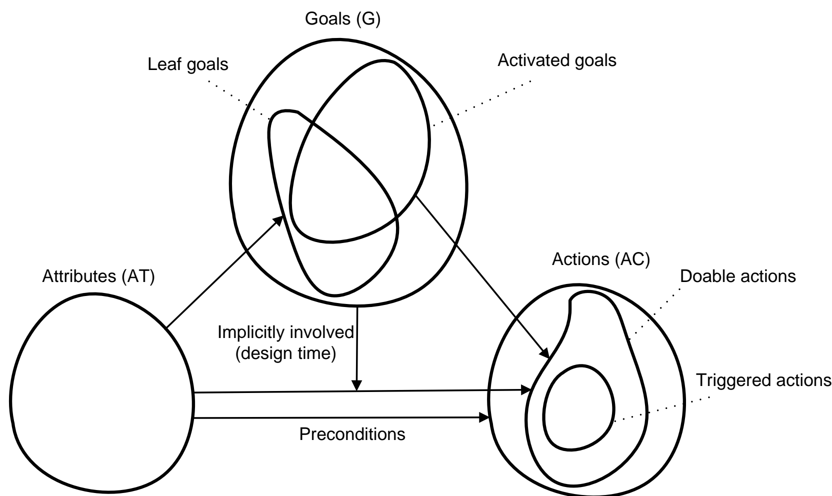

Figure 4.8: Hybrid composition pattern

Basically, these patterns define policy-based adaptation based on actions, goals (different forms), and even a utility-based form. As mentioned, goal policies are linked to the goal-centric pattern, while action policies are related to the action-attribute-coupling

pattern. Utility policies may be connected to any of these three approaches. Utility values may be assigned to either goals- their satisfaction and denial states- or attributes- each state or configuration. By assigning utility values to both goals and attributes, the hybrid pattern can also benefit from utility policies. Various mechanisms can be designed and developed based on the above composition patterns. Several mechanisms are discussed in the next chapter.

4.6 Summary

This chapter discussed modeling the key entities of an adaptation manager. The main idea was inspired by models developed in RE and robotics. Two significant points need to be addressed by an adaptation model. First, the model is required to be usable at run-time by the adaptation processes. This is particularly true for detecting and deciding processes that need to trace adaptation requirements and to reason during the system operation. Second, the dynamic variability of adaptable software should be taken into account, whereas in common models in RE the decisions are made off-line. In other words, variants are available at run-time as tradeoffs to satisfy adaptation requirements in different situations. Note that in this chapter, actions as transitions between these variants, are emphasized instead of the variants themselves.

The introduced quality-driven framework, QFeam, captures three key concept spaces in the adaptation manager. The goal space includes goals and their subgoals, which are generally represented in a goal graph. A metamodel has been discussed to capture the essential characteristics of goals. The attribute and action spaces have been also presented by metamodels. The important task in building an adaptation model is composing the entities in the three spaces. This task has been represented by three composition patterns to cover different approaches to building adaptation models.

Adopting a specific composition pattern is strongly correlated with the selected mechanism for run-time reasoning based on the built model. The next chapter focuses on adaptation mechanisms corresponding to these patterns.

Chapter 5

Adaptation Mechanism Design and Evaluation

> "*High degrees of specialization may be rendering us unable to see the connections between the things we design and their consequences as they ripple out into the biosphere.*" Terry Irwin

An adaptation mechanism includes four essential processes, namely monitoring, detecting, deciding, and acting, as discussed in Chapter 2. This thesis focuses on the deciding process, as a key process in run-time adaptation. Of course, other processes are also important in an adaptation mechanism, but this specific process has been addressed to a lesser extent relatively in the self-adaptive software research community. The following are some pertinent points with respect to the four adaptation processes in QFeam:

- The *monitoring process* basically provides situation-awareness, and as described in Chapter 2, self-awareness and context-awareness are among the primitive properties underlying all other high-level self- * properties. This is the reason why both self and context attributes are involved in the attribute set AT. The important point is that attributes are the outcome of the monitoring process rather than the sensors in the adaptable software.

  The sensor outputs may be filtered or pre-processed before generating the attributes in the adaptation model. For example, applying moving average to the response time of an entity in the adaptable software is required for computing the attribute "average response time".

- The *detecting process* aims at finding violations of adaptation requirements. As pointed out in the problem description, since the deciding process cannot be completely isolated from the other processes, the detecting process is partially covered

in this thesis. This is particularly essential in designing a mechanism for models built using the goal-centric composition pattern. The detecting process may behave differently depending to the adopted composition pattern. Generally, the detecting process analyzes the situation in each adaptation period to find violations from requirements. In case of using the goal-centric and hybrid cases, this mostly turns into *tracing goals* in order to find activated goals (i.e. denied goals). However, in the case of employing the attribute-action-coupling pattern, the detecting process relies more on abnormal or invalid patterns of attribute values or certain states that trigger actions.

- One of the main tasks of the *deciding process* is resolving conflicts between activated goals and suggested actions. This process completely depends on the adopted composition pattern, similar to the detecting process. In a goal-centric mechanism, the process should incorporate goals in decision-making through a competitive or cooperative manner. From another point of view, goals can be involved via qualitative or quantitative approaches, similar to reasoning in the requirements goal models [61].

  Since multiple goals need to be considered in decision-making, an aggregation module, such as an action fusion module, may also be required to select the final set of actions. Another point is that the goals can be in different levels of the goal hierarchy, but decision-making is much easier if all of the goals in G that are involved in the process are at the same level. Variant solutions for composition patterns are briefly discussed in this chapter.

- The *acting process* is responsible for executing the triggered actions. These actions, depending on the deciding process, may be atomic and low-level, like a tactic, or composite and high-level, like a strategy. This thesis does not address the details of the acting process, except in a limited way in the case studies. There may be some conditional decisions in composite actions, that this process manages to execute. The realization of this process is partly related to the effectors and how they are implemented.

In this chapter, two assumptions should be taken into account: First, while the mechanisms are generally applicable to the asynchronous mode as well, discussions in this chapter are limited to synchronous adaptation by default. Second, it is assumed that the adaptation mechanism is running every adaptation period T_{ad} by default. If any adaptation process or part of a process is running with a different period, it is explicitly mentioned.

5.1 Adaptation Mechanisms

This section discusses various adaptation mechanisms, which can be designed and developed based on the three composition patterns mentioned in Chapter 4. Two of them are employed and compared in the case studies presented in Chapter 6.

This section classifies adaptation mechanisms based on their corresponding composition patterns. Therefore, the three following sections address three classes of such mechanisms. However, regardless of the composition pattern each mechanism relies on, two general forms can be adopted in mechanism design: i) *Mono-deciding* including a single goal model and a single deciding process for all activated goals, and ii) *Multi-deciding* consisting of several goal models, several deciding processes, and an action fusion module. In a sense, these two options are patterns for adaptation mechanism design. Mono-deciding is basically a classic goal-driven decision-making process based on a single world model. As discussed in Chapter 3, it has been commonly used in classical robotic systems, but some problems limit its effectiveness and scalability.

5.1.1 Goal-centric Mechanisms

In the goal-centric composition pattern, goals play a key role in the adaptation mechanism, especially in the deciding process. Several mechanisms can be designed for working with a model built upon the goal-centric composition pattern. This can be a mono-deciding mechanism, which has been employed in some systems, such as mobile robots. Alternatively, it can be a multi-deciding mechanism, which can be realized in at least two ways. In both ways, the mechanism includes several deciding processes based on several goal models. However, in one mechanism all goal models consist of a similar subset of G, whereas in the other mechanism different partitions of G are involved in each goal model. The latter may even partition G into subsets including a single goal. In both of these cases, a fusion module is required to aggregate the decisions made by the processes.

A *Goal-ensemble* mechanism employs the second way of realizing multi-deciding. This section discusses the goal-ensemble in detail, and its concrete realization is discussed later in Section 5.2. Figure 5.1 illustrates the goal-ensemble mechanism. The monitoring and acting processes are shown as black boxes to indicate that they are out of the scope of the QFeam framework. Although each partition of goals in the goal-ensemble mechanism may have more than one goal, without lack of generality, the assumption in this section is that only a single goal is present in each partition. The following parameters are used in the goal-ensemble mechanism:

- $\overline{AC}$ denotes the set of doable actions.

- $\hat{AC_i}$ denotes a preference set representing the suggested actions by goal g_i.
- $\tilde{AC}$ denotes the set of triggered actions.
- $\hat{G}$ denotes the set of activated goals.
- $LeafGoals$ denotes the set of leaf goals.

In a goal-ensemble mechanism, the detecting process is realized by the *tracing* and *activating* modules, depicted in Figure 5.1. As discussed in the previous chapter, leaf goals have quantifiable activation criteria. Therefore, they are traced based on the latest values of attributes $(at_i, value_i)$ in each period T_{ad}. Next, the activated leaf goals can stimulate the goals $g_i \in G - LeafGoals$. The goals are related together by logical operators, and the active goals can propagate their new states to the other goals.

The activating module is implemented based on how the activation criteria have been specified for goals. It can be quantitative or qualitative, similar to reasoning in the requirements goal models [211]. The activation can also be accomplished as a fuzzy inference, which is basically qualitative. Activated goals $\hat{G}$ can be any of the goals in G, but it is possible to assume that they are only a subset of $LeafGoals$.

In each adaptation period T_{ad}, doable actions should be specified before selecting an action to be executed. The necessary preconditions for each action need to be checked before generating any plan. This is performed by the *checking preconditions* module in Figure 5.1. The output of this module is $\overline{AC}$.

The deciding process is a mapping $\overline{AC} \rightarrow \tilde{AC}$, and the *generating action list* is responsible for this part of the mapping:

$$\overline{AC} \rightarrow \hat{AC_i}, \hat{AC} = \bigcup_{i=1}^{|\hat{G}|} \hat{AC_i}$$

In a goal-ensemble mechanism, each goal behaves as a decision maker. Each activated goal $\hat{g}_i$ suggests actions through a preference set $\hat{AC_i}$. This set for each $\hat{g}_i \in \hat{G}$ consists of 3-tuple elements in the form of

$$\hat{AC_i} = \bigcup_{j \neq k} (ac_j, PrefRel, ac_k) \tag{5.1}$$

with preference relations

$$PerfRel \in \{\prec, \succ, \preceq, \succeq, \sim\} \tag{5.2}$$

such that

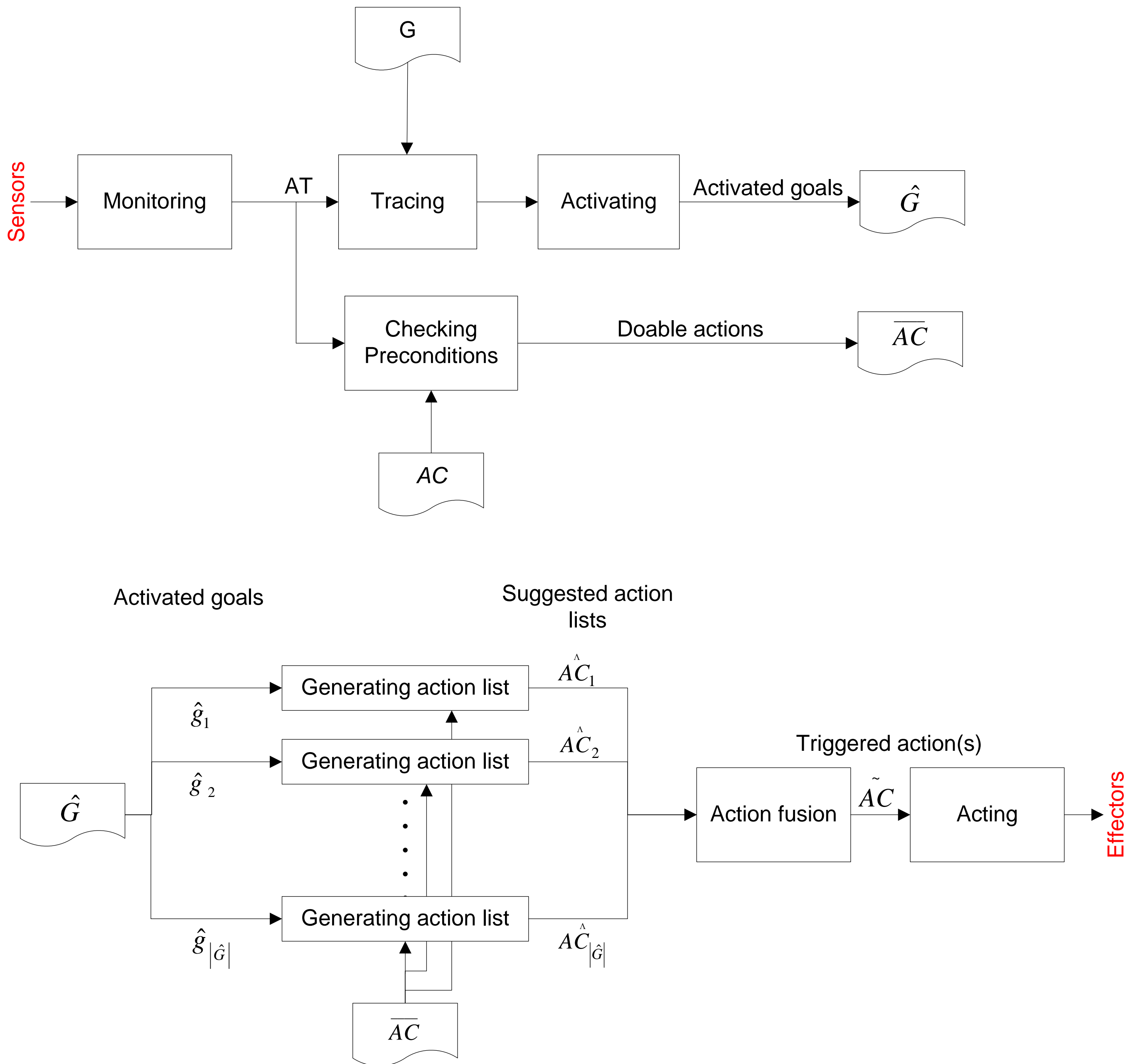

Figure 5.1: Goal-ensemble mechanism

- $ac_i \preceq ac_j$ iff $ac_j \succeq ac_i$
- $ac_i \prec ac_j$ iff $ac_j \succ ac_i$
- $ac_i \succ ac_j$ iff $ac_i \succeq ac_j$ and $\neg(ac_j \succeq ac_i)$
- $ac_i \sim ac_j$ iff $ac_i \preceq ac_j$ and $ac_i \succeq ac_j$

One may wonder why each goal does not simply suggest only a single action. The reason is that generating preference matrices provide more information to the deciding process for selecting the final action. In fact, this leads to having a richer action fusion method.

Each action in $\hat{AC}_i$ is not necessarily an atomic action. It can be a composite action that includes a workflow of atomic actions. In a more complex form, each element of $\hat{AC}_i$ may be a new plan generated by $\hat{g}_i$ using the actions in $\overline{AC}$. Technically, it is possible to design the mechanism with the aid of AI planning algorithms like STRIPS [133]. However, as pointed out by other researchers, planning is not an suitable option for all self- * properties- such as self-optimizing- and their associated quality goals [197]. Moreover, in run-time adaptation, because the adaptation manager is receiving feedback continuously from the external world, continuous planning is required. As discussed in Chapter 2, this is possible via contingency planning or replanning [174], based on execution monitoring from the acting process or attributes from the monitoring process. If this is feasible with moderate complexity and performance, it can be an option for generating action lists. However in practice, at least for large-scale software systems, it is still out of reach. On the other hand, the successful experiences with the Action Selection Problem (ASP) in behavior-based robotics, outlined in Chapter 3, indicate that selecting a single action in each T_{ad} can be quite effective. Continuous feedback from adaptable software (i.e., self) and environment (i.e., context) can improve and correct the decision through time. Therefore, in the goal-ensemble mechanism the assumption is that in each T_{ad} the deciding process is in charge of selecting one final action. Of course, this mechanism can provide an ordered set of actions.

The second phase of the deciding process is *action fusion*. In order to determine the final triggered action, the generated sets $\hat{AC}_i$ should be aggregated. Therefore, the action fusion is a mapping $\hat{AC} \rightarrow \tilde{AC}$. The fusion can be competition or cooperation among activated goals. The former could be a winner-takes-all game, in which the action fusion determines the winner goal, which in turn determines the triggered action among doable ones. The straightforward way to implement winner-takes-all is using goal priority. The cooperation approach can be realized by a weighted voting game, in which each goal nominates preferred actions, and the action fusion aggregates the preferences with the aid of a voting schema. Section 5.2 elaborates GAAM that realizes the goal-ensemble mechanism using a weighted voting game.

In the beginning of this section, it has been mentioned that the goal-centric pattern can be implemented in two ways using the multi-deciding approach. In the case of having similar subsets of goals in each partition, the adaptation mechanism also uses collective decision-making. In fact, this option realizes an ensemble of several deciding processes, each of which selects an action or generates a preference matrix. This idea is similar to the approaches employed successfully in other domains, like data mining and control. In data mining, consensus clustering and multi-clustering methods are applied to the problem domains in which a single method with specific parameters is not sufficient (e.g., see [14]). This idea is also used in control engineering where a single controller cannot be designed in such a way that it behaves as expected in all situations (for example see [88]).

5.1.2 Attribute-Action-Coupling Mechanisms

This set of mechanisms are commonly used in self-adaptive and autonomic software systems. These mechanisms connect attributes to actions without explicit goals. Attributes can be connected directly to actions as in simple condition-action rules, or they can define states to specify actions as state transitions. This set of mechanisms includes the following categories:

- Rule-based mechanisms using crisp or fuzzy rules to connect attributes or states to actions
- State-based mechanisms, which are often based on actions as state transitions, such as the Bayesian decision network or Reinforcement Learning (RL)

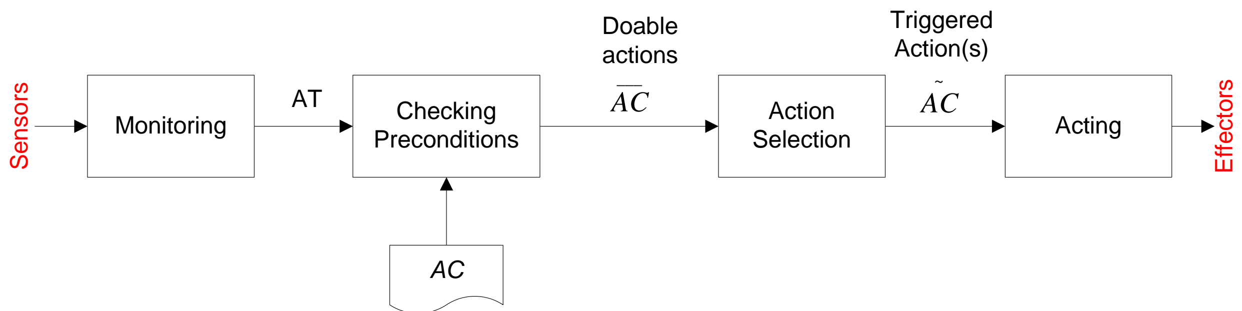

Figure 5.2: A general schema of attribute-action-coupling mechanism based on rules

The first category mostly conforms to action policies, while the latter is closer to utility policies, since different states can be labeled by utility values. Figure 5.2 illustrates

the general schema of an attribute-action-coupling mechanism realized by rules. Because checking preconditions and trigger conditions are performed in sequence, these processes can be merged. Action selection in rule-based mechanisms is often a conflict resolution process. A simple way to do this is to assign priority to rules that in large rule sets may not be an easy task. The detecting and deciding processes in this approach are similar to a typical rule or policy engine. Note that a multi-deciding design can also be adopted here to aggregate several sets of such mechanisms. In this case, action fusion is added after action selection, which again can be based on a voting game.

A crisp rule-based mechanism is used in the case studies in Chapter 6, and is compared with a goal-centric mechanism. Also, a fuzzy rule-based mechanism has been employed during this research, but the results are not presented in this thesis for the sake of brevity. Interested readers can find more information in [181]. A state-based mechanism using RL has also been implemented in a case study [7], and it is briefly covered in the next chapter.

5.1.3 Hybrid Mechanisms

A hybrid mechanism combines the goal-centric and attribute-action-coupling mechanisms for realizing the deciding process. A goal-centric mechanism has the power of explicit goals as the pivot of the decision-making. Generally, goals have a higher level of abstraction, and using a goal model enables the adaptation manager to trace goals in the hierarchy. However, goal-centric mechanisms may be slower than the attribute-action-coupling mechanisms, particularly in the case of using planning algorithms. Furthermore, in the case of having a priori domain knowledge for linking attributes or states to actions, less effort might be required to design and develop the adaptation manager. On the other hand, an attribute-action-coupling mechanism can be hard to design and tune in a large system with any of attributes and states. Add to this the tougher task of maintaining and evolving the links between attributes and actions (i.e., action policies). A hybrid mechanism aims at benefitting from the merits of both patterns while avoiding their disadvantages.

Several solutions can be employed for mixing goal-centric and attribute-action-coupling mechanisms. Again, these solutions can be assumed as a fusion of either competitive or cooperative manners. In the former case, an idea similar to the subsumption architecture can be used [29]. As explained in Section 3.5.1, the subsumption uses a set of behaviors as action triggers. The idea can be adopted so that the actions suggested by goals, suppress the actions suggested by attributes or state transitions. In fact, this idea was suggested later in the three-layer architecture by Gat [59]. In the three-layer architecture, the deliberator layer deals with goals and it is clearly separated from the controller layer dealing with low-level rules. Based on Gat's idea, Kramer *et al.* suggest an architecture-based adaptation approach, which utilizes a goal management layer [103]. This layer generates a new plan to satisfy the goals, in the case of facing a situation which has not been foreseen. This solution

uses the idea of a complementary goal-based deciding process, while a hybrid mechanism may use a collective approach instead.

Figure 5.3 depicts a general schema of a hybrid mechanism with an action fusion module that realizes the collective decision-making. Action fusion can be realized as a voting game for aggregating suggested actions. However, this can be in the form of run-off voting with more than a voting phase. For the complementary option, the figure is slightly different, in that a connection is added between the two decision-making layers. The action fusion module is not necessary in this case.

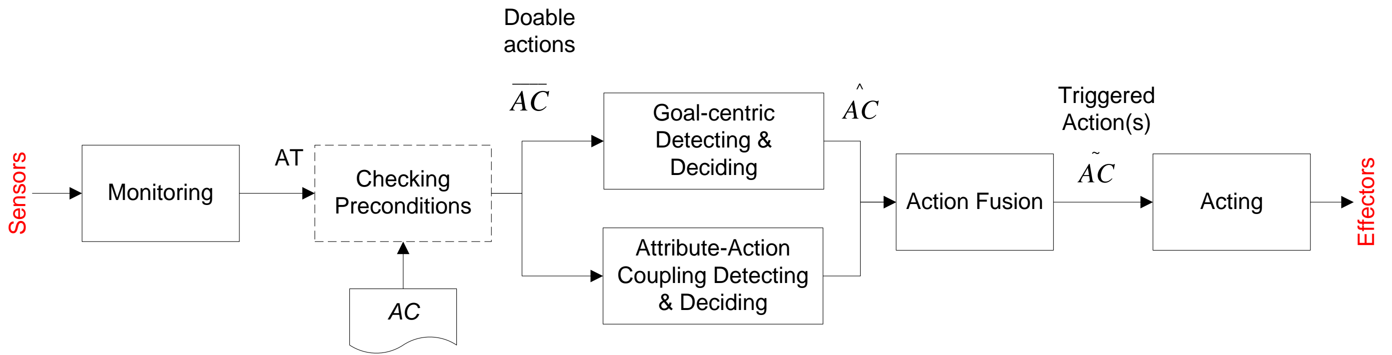

Figure 5.3: General schema of a hybrid mechanism using collective decision-making

There are other possible designs to realize a hybrid mechanism, such as the activation network from Pattie Maes [125]. In the activation network, which is quite complex and hard to tune, not only goals can trigger actions, but also actions can trigger each other based on their preconditions. In this case, preconditions are not merely related to attributes, and the postconditions (i.e., impacts) of other actions may be a part of preconditions.

In short, a hybrid mechanism can be designed in the following ways:

- Complementary, in which the goal-centric mechanism decides when the attribute-action-coupling mechanism cannot proceed.
- Collective, in which two separate goal-centric and attribute-action-coupling mechanisms suggest actions, and the action fusion module aggregates them to select the final adaptation action.
- Fully connected network, in which goals, actions, and attributes are linked together, and after stimulation by attributes, the network converges to an action.

5.2 Goal-Attribute-Action Model (GAAM): A Concrete Adaptation Model

This section elaborates the details of a concrete model, the Goal-Attribute-Action Model (GAAM), built based on the goal-centric composition pattern and the goal-ensemble mechanism. The model was first introduced in [185]. The notable assumption is that in GAAM each goal partition includes only a single leaf goal.

5.2.1 Adaptation Model in GAAM

GAAM specifies a graph $H = \{V, E\}$, where the vertices are $V = \{G \bigcup AT \bigcup AC\}$ and the edges are $E = \{AM \bigcup PM \bigcup ASC\}$, where AM is an activation matrix, PM is a preference matrix, and ASC is an aspiration criterion matrix. H is depicted in Figure 5.4. In this figure, it is assumed that

$$n = |AT|, m = |G|, p = |AC|$$

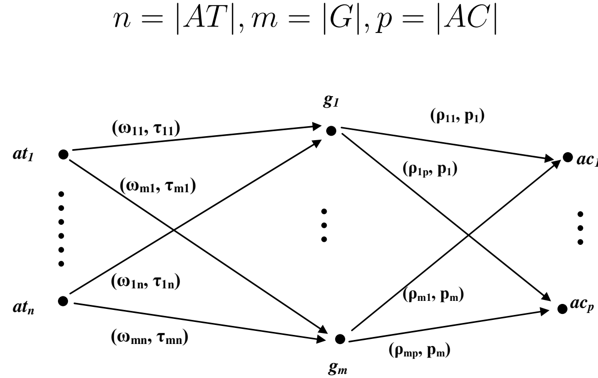

Figure 5.4: Representing GAAM as a graph

The following two sets of parameters are defined for composing GAAM:

- **Relating goals to attributes**
  - **Activation Matrix**: The matrix $AM = \{\omega_{ij}, i = 1..m, j = 1..n\}$ shows the relationships among m goals and n attributes. The values show how much each attribute affects any of the goals. In a sense, this is the weight of each attribute for each goal. It can be zero, which means "don't care", or other positive values. In an extended form it can be a fuzzy term.

- **Aspiration Criterion Matrix**: The matrix $ASC = \{\tau_{ij}, i = 1..m, j = 1..n\}$ specifies the aspiration criteria for each attribute at_i of each goal g_i. By default, this a threshold value that determines what the desired levels of attributes for each goal should be.

- **Relating goals to actions**
  - **Preference Matrix**: The matrix $PM = \{\rho_{ij}, i = 1..m, j = 1..p\}$ shows the action preferences of each goal g_i . As noted before, the postconditions of actions are defined as goal preferences. In fact, each goal g_i determines its preferred actions based on their impacts. The preferences are in the form of the three relationships $\succ$, $\prec$ and $\sim$. Element ρ_{ij} in PM can be defined using ordinal or cardinal utility functions. These two forms of defining preferences will be elaborated further in Section 5.2.3.

The goal space in GAAM represents only leaf goals, which means that at run-time only these goals are used by the adaptation mechanism. Goals in GAAM have two other prominent properties:

- **Activation criteria**: The ACtivation Criteria (ACC) vector is defined as $ACC = \{\theta_i, i = 1..m\}$, where can be threshold values or more complex conditions. In fact, each θ_i is defined as

  $$\theta_i = f(\sum_{j=1}^{|AT|} \omega_{ij}\tau_{ij})$$

  Another possible parameter for activating goals can be the rate by which goals participate in decision-making. This is suitable for synchronous events (e.g., the expected number of user requests in specific hours) or for goals that do not need to be satisfied frequently.

- **Priority**: The priority p_i determines the weight of the goal g_i in the GAAM model, and impacts its influence in the adaptation mechanism. In fact, the *Priority vector* $PV = \{p_i, i = 1..m\}$ quantifies the order of goals in $g_i \preceq g_j \preceq ...g_k$. It is assumed that $\sum_{i=1}^{m} p_i = C$, where C is a constant value (e.g., 1 or 100).

Attributes $AT = \{at_i, i = 1..n\}$ represent quantifiable properties of adaptable software, as defined in Section 4.4.2. The attributes in GAAM can either be kept separately or combined as states in order to reduce the dimension of the attribute space. Adaptation actions $AC = \{ac_i, i = 1..p\}$ are changes applicable to adaptable software entities that use the provided effectors. For an action to be eligible fir selection, its preconditions should

be satisfied. The precondition set is denoted as $PC = \{pc_i, i = 1..p\}$. In GAAM, each ac_i can be a single or composite action, and the precondition set refers to the first set of conditions in the chain of actions. In a composition action, the preconditions of each action are prepared by the previous action in the workflow. A trigger condition matrix $TCM = \{tc_{ij}, i = 1..m, j = 1..p\}$ in GAAM is defined as the winning condition in the action fusion, which is a voting game. For example, in the majority voting schema, each tc_{ij} has the maximum number of votes from all goals. In this way, the trigger condition fires the first action, and then the preconditions of other actions will be fired in a domino effect, unless some precondition is not satisfied.

5.2.2 Goal Preferences in GAAM

Determining goal preferences in GAAM is not a straightforward task. Generally, this task can be mapped to the preference elicitation problem in decision theory, utility theory, and AI. In the context of the preference elicitation problem, different solutions have been suggested [38], and some of these solutions have been employed in RE, as described in [25]. In GAAM, the preferences belong to adaptation goals, which represent the adaptation requirements. The goal preferences are actually based on the impact of actions on goals. As described in Chapter 4, different ways can be utilized in building the preference matrices of goals. In GAAM, goal preferences can be defined as ordinal or cardinal utilities for actions in each goal. The following sections elaborate further on these two approaches.

Preferences with Ordinal Utility

One way to define action preferences is using the *ordinal utility* form. In this way, each goal g_i presents its preference as a relationship among actions using the operators $\succ$, $\prec$, and $\sim$ defined in equation (5.2). For example, a goal g_i may define $ac_i \prec ac_j \sim ac_k$ to express that it prefers ac_j and ac_k over ac_i, if it is eligible to decide. The ordinal utility keeps the preference structure simple, although it does not clearly show how much an action is preferred to another.

The postconditions of each action play a key role on its impact on goals, and its rank in the preference lists. For some goals, it is impossible or difficult to quantify the impact of actions on goals, but it is feasible to judge (by stakeholders or administrators) their order. For example, assume we have maximum user satisfaction in the adaptation specification. Evaluating this goal depends on the end-users working in each period of time, and it is not easy to say exactly which actions improve or deteriorate the satisfaction level. However, it is possible to say that the users prefer actions providing more security to those guaranteeing a good response time.

For example, suppose that there are four goals g_1 to g_4, and four alternative actions $\{ac_1, ac_2, ac_3, ac_4\}$. Goals have priority values $PV = \{3, 2, 2, 1\}$ for g_1 to g_4 respectively. The following preference structure may be defined by these goals in PM:

g_1	g_2	g_3	g_4
ac_1	ac_4	ac_2	ac_3
ac_2	ac_1	ac_4	ac_4
ac_3	ac_2	ac_1	ac_1
ac_4	ac_3	ac_3	ac_2

For this form of preference structure, various voting mechanisms are applicable. The most common ones are the plurality and Borda count mechanisms [201]. In plurality voting, the winner is simply the candidate that the majority of voters preferred to the others. Borda count extends this method by involving the entire preference list by assigning an index number to each element in the list [201]. The winner is the candidate that has the highest sum for all voters. The advantage of this method is that it is possible to know the rank of each candidate and its difference with the others in the voting results.

The fairness of the voting schemes has been investigated by other researchers, and is out of the scope of this thesis. Here, two important characteristics, monotonicity and Pareto optimality, can be considered. The former means that if a goal raises the rank of a winning action, it remains the winner, and when a goal lowers the rank of a loser action, it remains the loser. The Pareto criterion means that when every goal prefers ac_i to ac_j, ac_j is not selected [67].

For the above example, assume that the outcome of this weighted voting system is a single selected action. The following table shows the social choice for this case using three different voting schemes. Pairwise comparison is another applicable scheme, which compares the candidates pairwise to find the winner. The order of comparison (called agenda) is important in this scheme, and for this reason it may have different outcomes.

Voting Schema	Winner(s)
Plurality	ac_1
Borda count	ac_1
Pairwise comparison	ac_1 and ac_2

The Plurality and Borda count voting methods, which have Pareto and monotonicity properties, choose ac_1 as the winner. Pairwise comparison also has these properties, but sometimes chooses ac_1 (depending on the agenda). Among these voting schemes, the Borda count method uses all of the elements in each preference list without using an agenda or any other extra parameter. The Borda count method also produces more reasonable results

for different weighting schemas. For instance, for equal weights, the Borda method still chooses ac_1, while Plurality voting has no choice. For the latter, conflicts exist among all four candidates. The Borda count schema is selected as the default voting mechanism for the GAAM model, although the voting schema can be easily changed.

Preferences with Cardinal Utility

If it is possible to quantify how much an action is preferred (e.g., an absolute value in the range of 0 to 100), then the preference structure can be defined by *cardinal utility.* In this case, a voting schema can be applied to add the cardinal vote of each action, and select the action with the maximum value. Sometimes, this is called intensity voting [201]. It is also possible to multiply the cardinal utilities of candidates (Nash voting) [67]. Since this voting method needs to normalize the multiplication values, it is not appropriate in some cases. By default, intensity voting is the default voting schema for cardinal preference lists.

5.2.3 Adaptation Mechanism in GAAM

The GAAM employs a goal-ensemble adaptation mechanism. The deciding process models a cooperative game based on the weighted voting of activated goals (i.e., denied goals). Here, it is assumed that the GAAM is continuously being traversed from attributes to actions for a specific adaptation period T_{ad}. This implicitly means that a polling method is used for monitoring; although an event-based method can be used as well. Using the event-based method does not drastically change the mechanism: attributes stimulate goals by sending event notifications, and activated goals select their preferred actions. As noted before the detecting process is also involved in the traverse. Actually, tracing goals is a part of the detecting process to analyze adaptable software status and to find any anomalies (i.e., activated goals).

Figure 5.5 illustrates the flow of preparing voters and candidates, and the action selection mechanism. Before making a decision, it is essential to determine which goals have been activated and which actions are feasible. The activated goals ($\hat{G}$) are voters, and feasible actions ($\overline{AC}$) are eligible candidates.

The following algorithm elaborates further the steps of the action selection mechanism. In Step 1, all of the attributes at_i in AT are sensed and updated. In Step 2, for each goal g_i, related attributes in AM are filtered using τ_{ij} values in ASC. The outcome set $\hat{AT}$ indicates which attributes for each goal are denied based on the aspiration levels. Step 3 is tracing goals to determine which goals are activated (denied and need to be satisfied) using $\hat{AT}$ and ACC. The trace method can be implemented using a qualitative or quantitative approach. A straight-forward method is activating a goal when at least one associated at_i

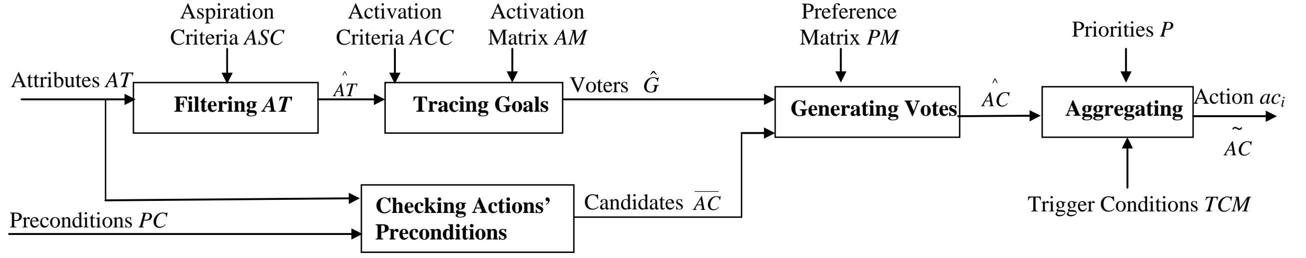

Figure 5.5: GAAM adaptation mechanism

in $\hat{AT}$ exists. In Step 4, feasible actions are identified by checking the set of preconditions PC. In Step 5, activated goals generate their preferred lists of actions (votes). This can be done differently for each goal. For example, the action list can be generated based on a fixed set of preferences or by dynamically generating preference lists using the history of previously performed actions. In fact, due to the different nature of adaptation goals and desired self-* properties, in many cases it is essential to have such a capability. The default preferences are specified in PM, but can be changed dynamically. The last step, Step 6, is aggregating the votes, which is the actual voting mechanism. Selecting the voting mechanism depends on the structure of the preference lists and how those preferences are used. Ordinal and cardinal preference lists can be used for this purpose.

Algorithm 1 Detecting and deciding processes in adaptation mechanism

Step 1: Sensing and updating all attributes
$sense(AT), \forall at_i; i = 1..m$

Step 2: Comparing attributes related to each goal with aspiration levels
$\hat{AT} = filter(G, AT, ASC, AM), \forall g_i, at_j, \tau_{ij}, \omega_{ij}; i = 1..n, j = 1..m$

Step 3: Tracing goals to activate those have the conditions
$\hat{G} = trace(G, \theta, \hat{AT}, ACC), \forall g_i, at_j \in \hat{AT}, \theta_i; i = 1..n$

Step 4: Checking actions preconditions to find eligible candidates
$\overline{AC} = checkPreconditions(AC), \forall ac_i; i = 1..p$

Step 5: Generate votes based on the predefined $\rho_i j$ or updated preferences
$\hat{AC} = vote(\hat{G}, \overline{AC}, PM), \forall g_i \in \hat{G}, ac_i \in \overline{AC}, \rho_{jk}; j = 1..m, k = 1..p$

Step 6: Aggregate votes and select an action or a set of actions
$aggregate(\hat{AC}, PV, TCM), \forall p_i \mid g_i \in \hat{G}; i = 1..n$

Two properties of interest for the introduced adaptation mechanism are soundness and completeness. In this context soundness means that "if an action ac_i is selected, it is feasible to apply, and it is towards satisfying adaptation goals." If we look at the mechanism, we see the transformation $AC \rightarrow \overline{AC} \rightarrow \hat{AC} \rightarrow ac_i$. Since $ac_i \in \overline{AC}$, its preconditions are satisfied and it is doable (i.e., feasible to apply). On the other hand, since ac_i is recommended by some goals to win the voting game, it is acting toward satisfying the goals. However, note that it may not be possible to find an action that is highly recommended by all goals. Therefore, the selected action can be satisfying for some goals and satisficing for others. There may be cases in which an action is at the bottom of the preference lists of some goals. Overall, a voting game, regardless of the applied schema may not result in an optimum selection for all goals. In fact, in some cases there is no such optimum action at all.

Here, completeness is defined as "if an action ac_i is sometimes doable, and can satisfy adaptation goals, it can be selected by GAAM." Assume $ac_i \in \overline{AC}$ is an action that is doable in some adaptation periods. The adaptation mechanism considers all doable actions in $\overline{AC}$ in each T_{ad} as candidates for the voting game. Therefore, ac_i would be one of the selected actions in the periods in which it is doable. If $ac_i \in \overline{AC}$ then $ac_i \in \hat{AC}$ providing ac_i is recommended by goals, and it would be selected if it is the winner of the voting game. The factor that is strongly impacts the chance of ac_i being selected is the set of preference lists. Therefore, there may exist an action ac_i that is sometime doable but is never selected by the adaptation mechanism. Being doable is a necessity condition but not a sufficient one. However, if action ac_i satisfies/satisfices enough goals participating in a game, it will be selected. In fact, one of the design guidelines is to test if the mechanism can select all the actions in the preference list. If an action is never selected, it can be deleted without any impact on the system.

5.3 Evaluating Adaptation Mechanisms

An adaptation manager is a software system, and as such, it can be evaluated with respect to i) what it is expected to do (i.e., functional requirements), and ii) how well it performs its functionalities (i.e., quality requirements). Of course, a subtle point is that the functionality of the adaptation manager is the quality of the adaptable software. Roughly speaking, the first aspect is the effectiveness, but the second one is too broad, and can be called "quality of adaptation", as suggested in [62]. An important point is that since the adaptation mechanism is the realization of the four adaptation processes, and the evaluation mainly targets this mechanism.

The first step in the evaluation process is to define criteria and metrics for evaluating the adaptation manager and in general, self-adaptive software. For the deciding process, Maes enumerates several conditions including finding good enough actions, minimizing back and

forth switching between actions that contribute to distinct goals, and never getting stuck in a loop or deadlock situation to satisfy an unattainable goal [126]. However, to the best of our knowledge, no specific set of criteria and metrics exist for verifying that a solution complies with these conditions. Recently in the SEAMS workshop, a metric (SAFU - Self-Adaptation Fitness Unit) has been proposed [42]. This metric tries to aggregate the quality dimension, i.e., resource overhead and engineering effort, as development and run-time costs and benefits of the system. However, some of these criteria have not been defined. For instance, effort is not defined precisely, because it depends on many other parameters like the complexity of the adaptation requirements, the number of sensors and effectors, and the complexity of the deciding process, to name a few. In short, after reviewing the few studies addressing evaluating self-adaptive software, it is not possible to come up with appropriate metrics and functions to evaluate such systems. This is one of the challenges that needs to be addressed [186].

This section aims at reviewing some helpful ideas in evaluating the effectiveness and quality of the adaptation manager. Some of these are employed in the next chapter in the case studies.

5.3.1 Effectiveness

Evaluating the effectiveness of an adaptation mechanism is discussed in two cases: i) in the presence of explicit goals, and ii) in the absence of explicit goals. In the *absence of explicit goals*, for attribute-action coupling mechanisms, utility values play the central role. These values are defined for states, for example, in the configuration model. As described in Chapter 4, $U(ac_i)$ is calculated based on these utilities. The utility function will be an aggregation of utility values in each T_{ad} during the evaluation period $EvalPeriod$.

Another way of formulating the utility function to evaluate the adaptation manager in this case is through SLA. This can be performed by calculating the cost-benefit value function of the provided services. This way is commonly used in evaluating Quality of Service (QoS) [140].

In the *presence of explicit goals*, for goal-centric and hybrid mechanisms, goals play the main role in the set of metrics and in the evaluation function. The general form of the evaluation function in this case can be stated as

$$E = f(\ |Denials(G)|, Deviation(G), Priority(G), EvalPeriod) \tag{5.3}$$

where function f takes into account the number of times the goals have been denied $|Denials(G)|$ (i.e., activated), how much goals were deviated from their activation criteria

$Deviation(G)$ (if measurable), the priority of goals $Priority(G)$, and the evaluation period $EvalPeriod = kT_{ad}, k \in \mathbb{N}$.

For goals with quantifiable activation criteria, the evaluation function can be formulated as a function that aggregates the deviations of all goals during a specific period. The deviation function, in the presence of explicit goals, is similar to the idea of the goal programming formulation [83]. The evaluation function for quantifiable activation criteria, for instance in leaf goals, in one round of adaptation is formulated as

$$E_i = (1/|LeafGoals|) \sum_{j=1}^{|LeafGoals|} Dev(g_j) \tag{5.4}$$

where $Dev(g_j)$ is the degree that a goal has deviated from the expected value of its activation criteria. If the activation criterion is expected to be in a specific range, the function can be evaluated as

$$Dev(g_j) = d_j^+ + d_j^- \tag{5.5}$$

where d_j^+ and d_j^- are the overshoot and undershoot corresponding to the range. Finally, the total deviation function, as an evaluation function, is defined as

$$E = (1/\ Rounds) \sum_{i=1}^{Rounds} E_i \tag{5.6}$$

where $Rounds = EvalPeriod/T_{ad}$ is the number of adaptation rounds. By this definition, the less E is, the more effective the adaptation mechanism will be.

Note that utility values can be employed in the presence of goals as well, but each $U(ac_i)$ is calculated based on a goal state instead of configuration states. In this case, the concept is similar to a deviation function, by considering the point that often, the higher the utility, the more effective the system is.

5.3.2 Quality of Adaptation

The "quality of adaptation" includes a broad class of factors such as performance and reliability as discussed by Gjorven *et al.* [62]. They state that some general software quality factors can be applied to adaptation as well. In another work McCann and Huebscher also list several items in which some are related to this category [134]. Other parameters may also be taken into account for evaluation, such as the level of automation and the

degree of human involvement. The latter is significantly important in defining, tuning, and maintaining adaptation requirements and policies, and more importantly building the stakeholders' trust to the system. However, these works did not extensively describe how quality factors defined in the quality models can be applied to the adaptation manager, and exactly what are the differences.

If we consider only efficiency in "quality of adaptation", two important factors are important:

- *Resource overhead* of the adaptation mechanism, especially if the adaptation manager uses shared resources with the adaptable software
- *Mean-Time-To-Adapt (MTTA)* as an indicator of mechanism performance. One assumption in the case of having a constant T_{ad}, can be $MTTA < T_{ad}$. However, depending on both how the acting process is implemented and the conflict between actions, a new action can be selected while the previously selected action is still executing.

Note that reactive/proactive and synchronous/asynchronous mechanisms may be evaluated differently regarding how soon they detect and act toward adaptation.

5.4 Summary

This chapter began by discussing three categories of mechanisms according to the previously discussed composition patterns. In the first two categories, two instances, goal-ensemble and rule-based mechanisms, were investigated more than the other solutions.

The goal-centric mechanisms enjoy the benefits of goal-driven decision-making. Major benefits include representing goals explicitly in addition to enabling the adaptation manager to know "why" adaptation is required. Moreover, enabling the manager to have a long-term or proactive plan is notable. The question that may come to the reader's mind is that which mechanism can outperform the other. The next chapter tries to answer to this question by a series of empirical studies.

This chapter also dealt partly with a less investigated area in self-adaptive software research, evaluation. This means how we can evaluate the adaptation manager with respect to the two aspects of effectiveness and quality. The work done here is not enough for this purpose, although it argues that in addition to formulating the common QoS functions, goal-based evaluation can also be helpful. This is true even in the case of having implicit goals in attribute-action-coupling mechanisms. As noted, the other software quality factors

can be also investigated in the evaluation phase. This line of research is the basis of testing and quality assurance for such systems.

An important aspect which was not addressed in this chapter is the human involvement in the adaptation mechanism. As discussed in Chapter 2, this is a prominent factor in developing and operating self-adaptive software systems, even in fully-automated systems. Human involvement can appear in different ways: as a supervisor, as an alternative adaptation manager if no action is effective, or merely as an end-user. Selecting any of these ways can affect the selection of the adaptation mechanism. For instance in behavior-based robotics several design patterns are discussed for the interaction between human and these systems [65].

Chapter 6

Implementation and Empirical Studies

"In the computer field, the moment of truth is a running program; all else is prophecy." Herbert Simon

This chapter includes two parts. The first part briefly reviews a developing framework that implemented at STAR lab in a collaborative research, called StarMX (STAR Management eXtension). StarMX has mainly developed for realizing the action-attribute-coupling pattern [13]. This framework has been presented as a Master's thesis by Asadollahi [12]. StarMx is now available as an open source framework[1]. GAAM has been added as an extension to StarMX, in order to facilitate implementing the goal-centric pattern.

In the second part, the empirical studies are discussed based on three applications. A separate section is dedicated to each case study, including an introduction, how to build adaptable software, how to build the adaptation manager, design of experiment, the obtained results, and the lessons learned. Results are analyzed using the SAS statistical analysis software. The first case study is a bookstore web application [178, 179]. The second system is a news application, introduced in Chapter 1, and simulated for the experiments [185, 7]. The third system is a VoIP call controller application that is built based on a service-oriented architecture.

Another system that was also studied during this research was a simulated data center. Fuzzy rules in an attribute-action-coupling mechanism were utilized for adaptation and the results were published in [181]. This system is not presented here for the sake of brevity.

Case studies are selected from two categories: enterprise and service-oriented systems. The rationale behind this selection is that these two classes of software systems have not

[1] http://sourceforge.net/projects/starmx/

been extensively addressed in the self-adaptive software community yet. Moreover, these two classes have different properties (e.g. see [148]), which causes different conditions for our experiments. Although these systems can be enriched by a variety of other sensors and effectors, the current setting seems sufficient for the experiments.

6.1 StarMX Framework for Java-based Systems

StarMX is a generic configurable framework that enables the creation of self-adaptive software applications. It is based on the separation-of-concerns principle, and clearly separates adaptation logic from application logic. StarMX has no dependency on application characteristics (e.g. architecture or environment) or any particular self-* property. Designed for Java-based systems, it incorporates Java Management Extensions (JMX) technology [89] and is capable of integrating with various policy/rule engines. The framework has been originally designed for Java EE systems, but because JMX has been incorporated in Java SE as well, after J2SE 5.0, the framework can be used for all Java-based systems.

6.1.1 StarMX Architecture

The StarMX framework does not enforce any specific approach or algorithm, and it aims to provide enough flexibility for the self-adaptive system developer to apply different composition patterns and adaptation mechanisms. Figure 6.1 shows the high-level view of the StarMX architecture. Generally, it consists of two main elements: the *execution engine* and a set of *services*.

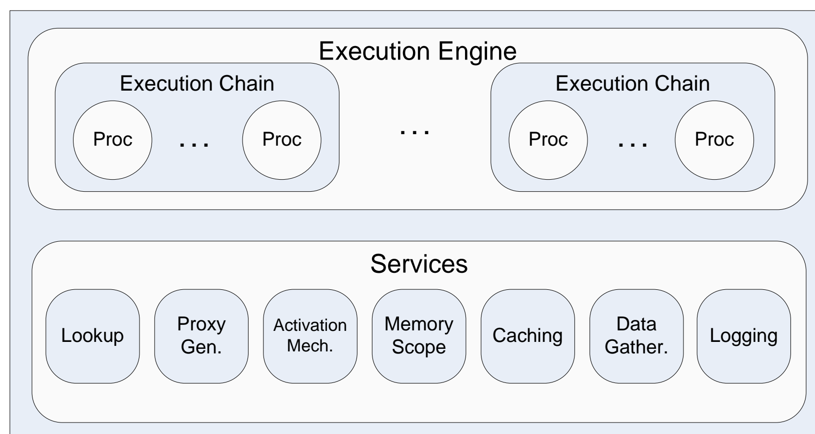

Figure 6.1: StarMX high level architecture [13]

The *Execution Engine* module realizes the adaptation processes. It executes the adaptation logic defined by the application developer to adapt the system to its current situation using *services* provided in the service layer. In other words, it enables the adaptation managers to perform their jobs.

The two key components of the execution engine are *Process* and *Execution Chain*. Figure 6.2 shows the architecture of this part of the framework and its interaction with the adaptable software. Processes are considered the building blocks of an adaptation manager. Each process may represent a single function or a group of consecutive modules of adaptation mechanism. The processes are chained together to form execution chains. The connection of processes in an execution chain is based on the Chain-of-Responsibility design pattern [55]. The execution chains act as adaptation managers, and each one is associated with an *activation mechanism*. When activated, the processes in the chain are executed sequentially.

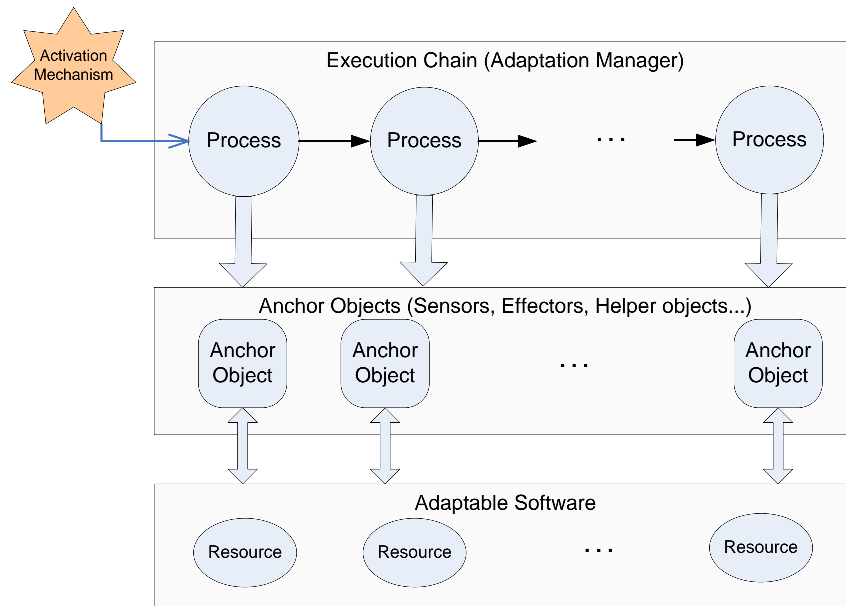

Figure 6.2: StarMX execution chain architecture [13]

As depicted in Figure 6.2, each process needs a collection of objects, called *Anchor Objects*, to perform its task. These can either be sensors/effectors of the underlying adaptable resource, or helper objects that provide some services. The required set of anchor objects for each process and their lookup information are defined in the framework configuration file starmx.xml. At execution time, the anchor objects are created and injected into the process by the execution engine. To support standard forms of access to the anchor objects, StarMX offers the following techniques:

- *MBean* or *MXBean*: These are application instrumentation interfaces in the JMX architecture. The application developer may implement the application-specific sensors and effectors through a set of MBeans, or use MBeans/MXBeans provided by other frameworks or the Java EE application server.

- *JavaBean*: An anchor object can be a simple Java object instantiated by either the execution engine or a factory class using the Factory-Method design pattern [55].

There are two approaches for defining a process: a) using a policy language, or b) using Java code by implementing the *org.starmx.core.Process* interface. In the first case, the user defines the self-* requirements in the form of action policies in the attribute-action coupling pattern. StarMX employs the Adapter design pattern [55] to interact with external policy engines. StarMX also allows for the implementation of a process using the Java language to benefit from all features of the programming language in the specification of the adaptation requirements.

The composition of processes in execution chains to build adaptation managers can be either *static* or *dynamic*. In static mode, the chain of processes is pre-configured, while in dynamic mode, several execution chains may form an adaptation manager on-the-fly. In this case, the execution of a process may result in the activation of another execution chain (via sending an event). The execution chain architecture provides a high degree of flexibility in the design of adaptation manager. It allows to define all adaptation mechanism modules in a single process, or design an arbitrary number of chained processes for this purpose. For example, detecting and deciding processes can be merged into one StarMX process, or deciding can be split into several processes in certain complex cases.

Several services are also provided by the framework to enable the behavior of the execution engine, such as i) lookup to access the anchor objects, ii) proxy generation to create a proxy object [55] dynamically when the anchor object points to an MBean, and iii) activation mechanism to trigger execution chains by timer-based and event-based methods.

6.1.2 Run-Time Behavior

The run-time behavior of the framework is divided into three phases: *startup*, *operation*, and *shut down*. At startup, StarMX prepares the environment for the optimized operation of the execution chains. All services are initialized, and processes and execution chains are deployed based on the specified properties in the configuration file. As a part of the execution chain deployment, the activation mechanism is enabled to trigger the execution chain later (e.g., by scheduling a timer or subscribing for an event).

Once the framework has successfully started, it is ready to operate. In this phase, the execution chains are activated by their own activation mechanisms. Upon activation, the

Table 6.1: Evaluating StarMX overload using CC2 system [12]

	Light workload	Heavy workload
Total adaptation time (sec)	2.5189	4.8152
Total policy execution time (sec)	2.1485	3.8645
Total sensing/effecting time (sec)	1.2758	1.3617
# executed policies	2063	1472
Avg policy execution time (ms)	1.221	3.2712
Avg framework cost per policy (ns)	**179554**	**645856**
Adaptation proportion to total time (%)	0.21	0.57
Framework cost proportion (%)	14.7	19.7

processes in the chain are executed in order, providing each process with the required set of anchor objects. The process invokes the anchor objects' methods to obtain data from or to send commands to the application. If a policy language is used, the related policy engine is invoked through its adapter to execute the policy with the provided anchor objects. The process can also use the memory scopes to keep or share data with other processes. Note that it is not necessary for a process to use anchor objects. It may use only the output of the previous processes to perform its task. The StarMX framework is designed to serve the best performance at run-time by reducing the amount of time spent in the framework during execution. Table 6.1 shows the performance data measured for the CC2 case study, discussed in Section 6.4.7, under two workloads. This data indicates that the average framework cost per policy is less than 1%. This overhead is a part of the efficiency in quality of adaptation discussed in Section 5.3.2.

However, the total execution time can be affected by other external factors, such as the time consumed in the anchor objects, or in the external policy engine. Finally, at the shut down phase, the framework undeploys the execution chains and processes, and stops working.

6.1.3 GAAM Implementation

GAAM is implemented using the generic facilities provided by StarMX. In fact, it is an extension that uses the StarMX processes and its provided services to communicate with adaptable software and policy engines. Figure 6.3 shows the high-level view of GAAM class diagram, based on the StarMX framework. In this structure, goals and actions are StarMX processes, but attributes are directly linked to goals and Mbeans. In fact, attributes are related to processes via Mbeans. The "Voter" class is designed as a process to fuse the candidate actions. The voter implements the voting mechanism for this purpose.

Monitoring and acting adaptation processes are realized using two StarMX processes.

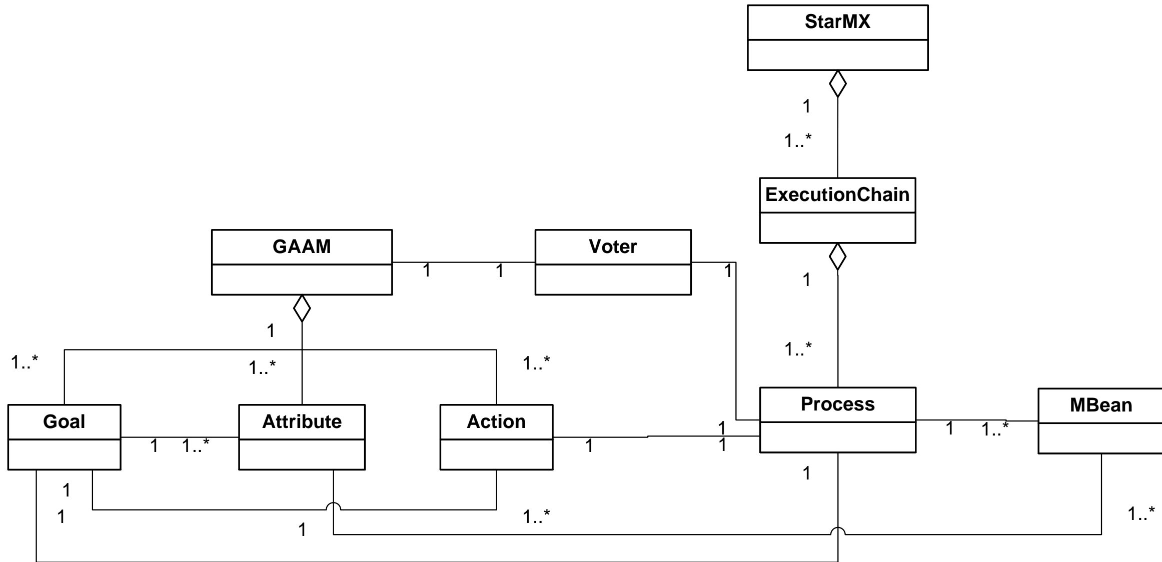

Figure 6.3: Implementing GAAM with StarMX processes

These two processes are implemented by two policy files, by default. However, depending on the developer's decision, these processes can be implemented by a chain of processes. For example, a composite action may be developed as several chained processes.

Goals and actions are defined as classes and GAAM instantiates the specified entities in *gaam.xml* from these classes. This XML file specifies the properties of the GAAM entities and their relationships. One of the major properties in the "gaam.xml" is the preference list for each goal. Preferences are defined by ranks and for some ranks there may be several actions. This means these actions are $ac_{first} \sim ... \sim ac_{last}$. XML code 1 shows a sample starmx.xml to define mbean server, mbeans, beans, and starmx processes. For instance, g_1 class and factory method are defined, then linked to a policy file, p1.arl, and an assigned object. Attributes are properties of Mbeans, which are linked to goals. Activation criteria can be checked inside a goal object or with the associated policy.

In each adaptation period T_{ad}, the GAAM is traversed using the flow described in Figure 5.5. By default, all $g_i \in G$ are traced in each T_{ad}, but the goals can be set to be traced based on different periods kT_{ad}. XML code 2 shows a sample gaam.xml. Actions and goals are defined by specifying their properties. Goal preferences are significant for linking goals to actions and facilitating decision-making at run-time.

XML code 1 starmx.xml sample

```
<starmx>
...
<mbeanserver lookup-type="jndi" id="ms">
    <jndi-param jndi-name="jmx/invoker/RMIAdaptor">
        <property name="...">org.jnp.interfaces.NamingContextFactory</property>
        <property name="...">jnp://129.97.92.121:1099</property>
        <property name="...">org.jboss.naming:...</property>
    </jndi-param>
</mbeanserver>
...
<mbean id="forwardingControl" object-name="..." mbeanserver="ms" interface="..." >
</mbean>
...
<bean id="g1" class="org.gaam.GoalModel" factory-method="..."/>
<bean id="g2" class="org.gaam.GoalModel" factory-method="..."/>
...
<bean id="voter" class="org.gaam.Voter" factory-method="..." />
...
<execute/>
    ...
    <process id="policy1" policy-type="arl" policy-file="p1.arl">
        <object name="g1" ref="g1" />
        <object name="forwardingControl" ref="forwardingControl" />
    </process>
    ...
</execute>
</starmx>
```

XML code 2 gaam.xml sample

```
<gaam>
...
<action id="ac1" ... />
<action id="ac2" ... />
<action id="ac3" ... />
...
<goal id="g1" priority="5" ... >
    <preferedAction id="pac1" actions="ac2" />
    <preferedAction id="pac2" actions="ac3" />
    <preferedAction id="pac3" actions="ac1" />
    ...
</goal>
...
</gaam>
```

6.2 Case Study 1 (CS1): TPC-W Bookstore Application

The experiments conducted in the first case study, focused on the attribute-action-coupling pattern and corresponding mechanisms have two objectives. First, how goal and action decomposition can help us in engineering the adaptation manager based on the attribute-action-coupling pattern. Second, the impact of fine-grained actions are of interest, particulary parameter adaptation and those actions implemented by dynamic aspect weaving. In other words, the second objective is to investigate whether low-cost actions can cause a positive impact on self-adaptive system behavior in comparison with the no-adapt case.

The TPC-W application[2] has been selected as the case study regarding the mentioned objectives. TPC-W is a typical mission-critical e-commerce system, implementing a bookstore application in J2EE. It is often used as a benchmark for performance evaluation of application servers. Except for adding a payment method from a bank service, none of the functionalities in TPC-W has been changed. In the experiments, TPC-W is deployed on a JBoss application server v4.0.5 and it uses a MySQL v5.0 database server.

The details of the findings from this study have been published in [178] and [179]. The action and goal decomposition has been performed based on activity theory [3, 145]. The conceptual objective and activity hierarchy has been adopted for this purpose. However, for the sake of brevity, this section skips this theory and deals only with the experiences gained during engineering adaptation manager.

6.2.1 Building Adaptable Software

The SLA in this case study addresses three high-level goals, as depicted in Figure 6.4. The figure illustrate the refinement of these goals regarding SLA and SLOs. The leaf goals are defined based on the three following objectives: i) a Response Time (RT) less than or equal 220 ms, ii) a full service level including a full search result page with images, and iii) an availability (service uptime) equal to or greater than 99.9%. The RT threshold is low because the experiments have been performed in a local dedicated network that omitted the network delay effect. Because none of the selected actions monitor the network delay and bandwidth, this condition does not hurt the objectives. Of course, incorporating the network parameter in a shared network is more realistic.

[2]http://www.tpc.org/tpcw/

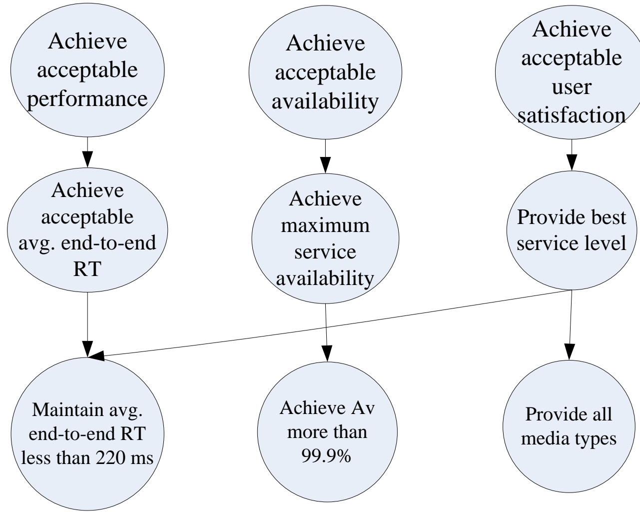

Figure 6.4: Goal hierarchy in TPC-W

As illustrated in Figure 6.5, two high-level actions are used for TPC-W. One action is service-level degrading and/upgrading and the other one is resource management. The decomposition of these two actions to atomic action (operations) $op_{ij}^{+/-}$ is as follows:

- Service-level degrading/upgrading (ac_1)
  - Quick (op_{11}^{+}) and slow (op_{11}^{-}) search method
  - Image disable (op_{12}^{+}) and enable (op_{12}^{-})
  - Switch to quick encryption (DES) (op_{13}^{+}) and slow encryption (RSA) (op_{13}^{-})
- Resource management (ac_2)
  - Increase (op_{21}^{+}) and decrease (op_{21}^{-}) KeepAliveTime

Here $op^{+/-}$ denotes changing from/to the default value. Indices i and j refer to action and operation respectively. Service-level degrading/upgrading (ac_1) adjusts the level of service provided by TPC-W. The first two operations (op_{11}^{+} and op_{11}^{-}) change the search result page to provide slow or quick search methods. op_{12}^{+} and op_{12}^{-} modify the web pages to exclude or include book images. Two other operations, op_{13}^{+} and op_{13}^{-}, alter the encryption/decryption methods used by the payment method to communicate with the bank service, to DES (key size 56 bits) or RSA (key size 1024 bits). These operations are implemented by dynamic aspects, and the adaptation manager uses JBoss AOP v1.5.5 to weave/unweave the corresponding aspects. The resource management action (ac_2) in this case study only

consists of two operations for adjusting the KeepAliveTime parameter. This parameter indicates how long JBoss keeps a thread for a connection when the connection is inactive (in milli-seconds - default value: 60k).

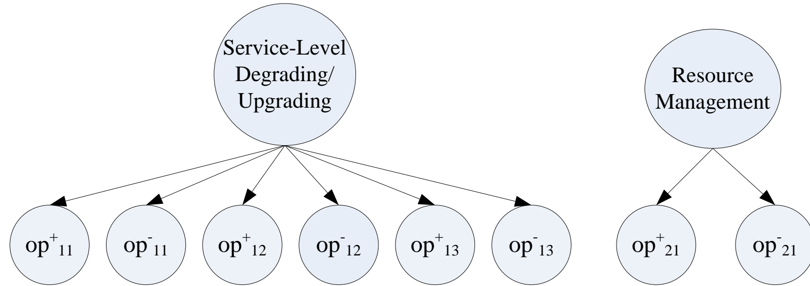

Figure 6.5: Adaptation action hierarchy in TPC-W

Preconditions of the above operations are not complex, because they do not need a considerable amount of resources, and they are only local and limited changes in the system. The only precondition is whether their counter-action (op^+ or op^-) has been implemented or not. For example, encryption can be switched to DES if it has been already set to RSA.

Several load and stress tests were run on TPC-W to determine the change impact of each operation. Based on the findings, the utility of the operations have been set as a fixed priority. This priority in ac_1 is $op^+_{11} \succ op^+_{12} \succ op^+_{13}$ and $op^-_{11} \prec op^-_{12} \prec op^-_{13}$. Since in some cases two activities are applicable (when their corresponding objectives were denied), $ac_2 \succ ac_1$ is set to apply the operations under ac_2, before the operations under ac_1.

6.2.2 Building the Adaptation Manager

The adaptation manager realizes the monitoring, analyzing, planning, and executing processes in order to manage adaptation changes at run-time. A common approach for realizing these processes and implementing the adaptation logic is using a set of rules (also called action policies or policies).

A set of action policies is developed and tuned to implement the adaptation logic. This set considers all of the activities and objectives available in the requirements specification (i.e. adaptation specification) for decision-making. Utilities, which are ordinal in this case, are the key information for prioritizing the adaptation changes in activity hierarchies. Overall, selecting a change (e.g. in the level of operation) depends on its utility,

failure/success of previously performed changes, preconditions, and the level of satisfaction/denial for corresponding entities in the objective hierarchy. In the presence of several competing actions or operations (like in this case study), a graph is required to present and relate all of this information. For TPC-W, an operation graph is used , which includes the system state in terms of previously taken operations. In fact, this graph is a conditional plan (or workflow) for executing an action.

Major parts of the adaptation manager for this case study were implemented using IBM ABLE v2.3.0 and the ABLE Rule Language (ARL) [22]. ARL rules are used to implement the analyzing and deciding processes. The monitoring and executing parts have been implemented in Java. For several cases, sensors and monitors were implemented by statically woven aspects. Sensors and effectors are introduced to the rule set using Java reflection.

The adaptation manager periodically checks the status of TPC-W through sensors and determines violated entities in the objective hierarchies (objective, goal or condition). For the experiments, the period of $t_m = 3sec$ is used for monitoring. When a goal or condition is violated, a policy will be fired and appropriate actions and operations will be executed. Of course, preconditions should be checked before actions are eligible to run.

After applying each operation in a specific time window, the planning process does not make any new decision in order to give the operation enough time to impact the system. This delay time is set to $t_d = 15sec$ in the experiments. A policy, at the operation level, checks whether it can apply an op^+_{ij}, or can roll back and execute op^-_{ij}. The corresponding conditions for an operation and its counter-operation are slightly different because otherwise, the system would have a bouncing effect of performing op^+_{ij} and then op^-_{ij} immediately. For instance, the condition for response time has a threshold of 220 mili-seconds in op^+_{ij}, whereas for op^-_{ij}, it will be about 200 mili-seconds.

6.2.3 Design of Experiments

The experiment is planned as a one-factor design with blocking, where treatments are different ways of adaptation and blocks are different workloads. In each block, three replicated experiments are conducted for each treatment under randomly generated workloads, in order to minimize experimental errors due to sporadic events.

Four different treatments have been used in the experiments, namely:

- No adaptation (**T1**)
- Resource management by adjusting KeepAliveTime (**T2**)
- Service-level degrading/upgrading using dynamic aspects (**T3**)

- A combination of the second and third treatments (**T4**)

Because the workload has a tremendous effect on adaptation effectiveness, the experiment is conducted with two different workloads. This can reduce the experimental error due to considering two regiments of workload. The first workload is a severe load (in comparison with the capacity), in which 400 users are generated in 100 seconds. In the second workload (medium workload), 800 users are generated in 400 seconds. The high and medium labels are based on the capacity of TPC-W in the described setup. Because these workloads determine two blocks in the experiments, high and medium loads are called **B1** and **B2** respectively.

Apache JMeter 2.3.1 is used to generate the workload. The workload is generated based on randomness in the behavior of users that go through different paths of the web application. A CBMG (Customer Behavior Model Graph) is used for modeling user behavior [137]. The CBMG is a stochastic graph where nodes are pages and arcs are transitions annotated by a probability value. The number of users and the arrival rate are set differently for each workload.

For the resource management action, only the KeepAliveTime parameter is used in the experiments. Two other important parameters were also tried during experiments, ThreadPoolSize and ConnectionPoolSize, but no significant impact was observed. JBoss seems to control the former by itself and the latter seems not to be adjustable effectively at run-time. In fact, JBoss lets us modify these parameters, but they seem to be adjusted again internally.

Under the high-load, when the system crashes, JBoss performs a recovery action and after a while the service is available again (the application is crash-safe). This means that the self-healing activity at the middleware-level has been involved in the experiments even in T1 (no adaptation).

In the conducted experiments two specific *research questions* are considered, as follows:

- **CS1-RQ1**: Are adaptation treatments (T2-T4) effective in comparison with T1?
- **CS1-RQ2**: Among adaptation treatments (T2-T4), which one has more positive impact on the system regarding the evaluation function and costs? In other words, how parameter adaptation, dynamic aspect weaving and hybrid actions impact the system behavior.

As mentioned before, a third research question can be also considered to investigate how goal and action decomposition can help engineering the adaptation manager using attribute-action-coupling. This question cannot be investigated quantitatively in a way

that is similar to the above questions, but the experience is valuable to compare with the previous work on attribute-action-coupling pattern in [181].

Goal deviations are used to evaluate how effective the whole adaptive software is. For this purpose, a function $D(.)$ is formulated to calculate deviations from conditions articulated in all objective hierarchies. Function $D(.)$ is defined as:

$$D_j(.) = \frac{1}{n}\sum_{i=1}^{n} w_j(d_i^+ - d_i^-) \tag{6.1}$$

where n is the number of samples from the system, j is the index of a condition, w_j is the weight of that condition, and d_i^- and d_i^+ are the negative and positive deviations respectively. In order to evaluate the adaptation, a value function is defined based on the deviation function as $E(.) = \sum_{j=1}^{m} D_j(.) + C$, where C is a constant and m is the number of conditions. C has been added because, in the best case, deviation is zero and in all other cases it is negative. So, $V(.)$ shifts the values to a positive region by a constant. C is set to $C = \max_n(D(.)) + b$ for n samples, and it is not exactly the maximum, in order to avoid a zero value in the data set ($b = 1$ by default). A notable point is that because $D_j(.)$ may have different units, a projection needs to be applied in order to put all the data in a single scale.

6.2.4 Obtained Results

System attributes are measured using both in vitro JMeter and JBoss sensors, and in vivo aspects woven to sense the data. For instance, the availability measure is calculated by analyzing JBoss logs, which show crashes and recoveries of the application.

Table 6.2: Evaluation function $E(.)$ in conducted experiments

	Response Time (msec)		**Availability**		**Evaluation Function**	
Treatment/Block	**High Load**	**Med. Load**	**High Load**	**Med. Load**	**High Load**	**Med. Load**
No adaptation	6537.39	256.71	85.31%	100%	1535.68	7963.29
	5225.25	268.56	97.21%	100%	2967.16	7951.44
	7300.92	299.43	90.1%	100%	820.03	7920.57
Resource management	1107.49	291.13	100%	100%	7112.51	7928.87
	5438.59	245.37	92.3%	100%	2703.87	7974.63
	1185.49	284.70	100%	100%	7034.51	7935.30
Service level upgrade/degrade	465.70	205.44	100%	100%	7691.81	7992.33
	381.83	198.18	100%	100%	7776.38	8010.88
	523.96	212.40	100%	100%	7633.14	7984.96
Hybrid fine-grained	705.04	234.13	100%	100%	7456.03	7944.58
	540.68	202.54	100%	100%	7621.55	8003.81
	521.01	205.82	100%	100%	7642.33	7991.04

Based on the availability, response time, service-level, and other system parameters, the evaluation function values are calculated by setting $C = 8000$ and $W_j = 1$ (See Table 6.2). For running ANOVA tests, within-block and the whole data sample set are taken into account. The former considers each of the medium- and high-load conditions separately, whereas the latter analyzes the whole data set. In order to project service level values, it is assumed each service level degradation equals to a 50msec degrade in RT. Each percentage of availability degradation is also mapped to a 10msec degradation in RT.

The first property that needs to be checked in the data set is the validity of parametric one-way ANOVA, including normality of residuals and homogeneity of variances in the treatments. Normality test is performed using the visual check of Quantile-Quantile plot (QQ-plot), and the homogeneity of variances in treatment groups is tested by the visual check of Spread-Location plot (S-L plot) for the second root of residuals and the mean of responses for each treatment group. In order to double-check the variance homogeneity, the Levene test is used as well [104].

For the purpose of investigating the details of differences among treatments, MCB (Multiple Comparison with the Best) and MCW (Multiple Comparison with the Worst) tests [75] are conducted to find the best and worst treatments using the Dunnette test.

Intra-block Analysis

In both cases of high and medium loads, QQ-plots show that the data are approximately normal, which is satisfactory for the test. The spread-location plots indicate that the variances are not significantly heterogenous in different treatments. To ensure the homogeneity, the Levene test was run for both cases, which endorses validity of the assumption by a 95% confidence level.

The summary of ANOVA tables for high and medium loads is shown in Table 6.3. In the high-load case, the treatment means are different (due to a rejected F-test), but this is not the case for the medium load.

Table 6.3: Summary of ANOVA for high and medium loads separately in TPC-W

Block/Stat	**F-test value**	$Pr > F$
High load (B1)	12.1	0.0024
Medium load (B2)	3.41	0.0735

In the high-load block (B1), the MCB test indicates that each of the adaptation treatments can be the best by 95% confidence level. On the other hand, the MCW test shows that only “no adaptation” treatment (control treatment) is the worst. For the medium load,

treatments are closer, and both MCB and MCW candidate all of the treatments for the best and the worst treatments. Pairwise comparisons of the treatments (by Dunnette test) under high load also verify that all adaptation treatments are similar by 95%. However, scrutinizing the simultaneous confidence intervals (SCI) reveals that T3 (service upgrading/degrading) is sightly more effective than the others. For the medium load condition (block 2), pairwise comparison again shows no significant differences (by 95% confidence), but T2 and T3 are slightly better regarding the simultaneous confidence intervals.

Inter-block Analysis

The analysis is also performed on a block design to compare four treatments simultaneously under high and medium loads. In this way, the most effective treatment can be determined.

The normality test in the blocking design passed because the QQ-plot shows a roughly straight line for the residuals. However, the S-L plot shows suspicious variations in variances. The Levene test indicates that by considering two blocks in the design, variances are not homogeneous by a 95% confidence level. A remedy for this situation is transforming the data samples. Box-Cox transformation is applied to find an optimal power p in the power transformation $x = (y^p - 1)/p$, where x and y are the transformed and original data respectively [104]. Then, the transformed data is used to test the H_0 assumption of mean equality, and contrasts between treatments.

Table 6.4 shows the summary of the ANOVA table for two-block design. The F-test indicates that the four treatments in these experiments are different from each other by rejecting the $H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4$ by a 95% confidence level.

Table 6.4: Summary of ANOVA (4 treatments, 2 blocks, and 3 replications in TPC-W)

Source	**F-test Value**	$Pr > F$
Error	9.63	0.0002
Treatment	5.01	0.01
Block	23.48	0.0001

The MCB test verifies that all of the adaptation treatments can be the best. However, MCW interestingly nominates T_2 (resource management), in addition to T_1, as the worst treatments. Pairwise contrasts show that T_2 is different from control treatment (no adaptation) by a 95% confidence level (in the Dunnett test). Scrutinizing confidence intervals in the above tests give us some clues that T_3 is more effective than the other adaptation treatments. In order to investigate this finding more accurately, a Dunnett test (by confidence level of 99%) is run to compare all treatments with T_3 (service upgrading/degrading). Results of this test, shown in Table 6.5, indicate that T_3 is significantly different from "no

adaptation", while we cannot say it is statistically better than T_2 and T_4. However, the confidence intervals in the other Dunnett tests show that its effectiveness is slightly better than the others.

Table 6.5: Contrasts between T_3 and other treatments (SCI: Simultaneous confidence Interval)- *** means significant difference

Contrast	**Mean Dif**	**SCI Low**	**SCI Up**	**Significance** ($\alpha = 5\%$)
T2-T3	-126.4	-2078.3	1825.6	
T4-T3	-1030.8	-2982.8	921.1	
T1-T3	-2012.1	-3964	-60.1	***

The contrast between the two blocks are also tested, which indicates these blocks are clearly different. This makes sense, because the behavior of TPC-W was not the same in medium and high loads.

6.2.5 Lessons Learned

The first research question, CS1-RQ1, contrasts adaptation and no-adaptation treatments. The outcome of intra- and inter-block ANOVA indicate a significant difference between treatments. MCB and MCW specifically show that T1 is the worst in terms of evaluation function. Therefore, the answer to this question is that T1 is less effective than adaptation treatments.

Regarding CS1-RQ2, the obtained results do not indicate that an adaptation treatment significantly outperforms the others. MCB shows that any of T2 to T4 are candidates for the best treatment. Although MCW suggests that T3 and T4 are better than T2, the Dunnette test does not endorse this by 95% confidence level. Hence, all the studied fine-grained actions in this case study seem equally effective.

The previous experience in developing the adaptation manager using attribute-action-coupling for a data center [181] showed that formulating action policies and tuning them can be time-consuming and difficult. In this case study, the adaptation manager is built with the aid of goals. As discussed before, in the attribute-action-coupling pattern, goals are implicitly involved in the relation between attributes and actions. In fact, the approach taken in this case study uses the goal-driven requirements engineering. The process starts from goals and then, by operationalizing goals, moves toward designing the adaptation mechanism. Comparing the engineering processes of this case study with the previous one on the data center case study shows that this approach needs less effort and is more

systematic. This approach especially aids us by highlighting why the system needs adaptation instead of directly dealing with how to adapt. In systems with lots of attributes and actions, using goals at the development time can speed up building the adaptation mechanism by the attribute-action-coupling pattern.

6.3 Case Study 2 (CS2): News Web Application

The second case study is a news web application that aims at evaluating the goal-centric pattern and especially the novel goal-ensemble-based GAAM in engineering an adaptation manager. This application has been introduced in Section 1.1. As described before the selected scenario is motivated by the performance and availability problems occurred in several news web sites after 9/11, especially in web sites across the US. For this case study, a multi-tier news application is simulated.

The self-optimizing and self-healing properties are considered as main adaptation goals. Two quality goals, performance and availability, are derived from these properties. As explained before, for this case study, application-level adaptation is taken particularly into account. The interesting question is how adaptation actions at the application level would impact the system behavior. Although other actions are applicable in this case, the focus is only on the application-level adaptation actions.

6.3.1 Building Adaptable Software

An experimental model is simulated as an adaptable multi-tier news web application. The model is based on a generic multi-tier enterprise application, which can easily be used for other domains such as e-commerce. Figure 6.6 illustrates the schematic view of the experimental model and the data flow. The model is a network of queues (or queueing network [86]) that is built as an open network with infinite population. The component model is a queue-server model that is implemented in the Simevents toolbox [131] of Matlab/Simulink. The queueing model is one of the common models in software performance engineering [18] and can be applied more easily to component-based applications in particular [39]. In this model, the stakeholders are two groups: i) business owners and administrators, and ii) end-users. The stakeholders define their expectations as goals for the system.

Each component has a priority queue based on the service time for requests. The inputs and outputs are as follows:

- Inputs: Service time, failure duration, failure probability, restart signal, and input traffic

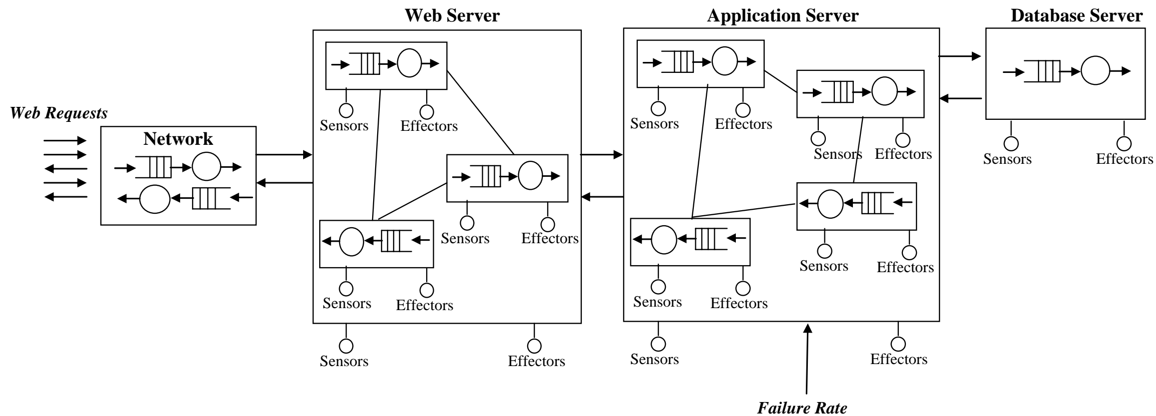

Figure 6.6: Experimental model of a multi-tier news application

- Outputs: Average end-to-end response time, number of requests served per second, component state, average number of requests in queue, and output traffic

For traffic generation, two parameters are taken into account: *inter-arrival time* (IATime) and *service time* (SVTime). For the conducted experiments, exponential and Weibull distribution functions were used for IATime. For SVTime, an exponential distribution was used with a changing mean to generate burst traffic, as was the case in 9/11. The three main data formats in news are video, image, and text, which are delivered to end-users with high/low quality or normal/small size. The quality and size factors have some impact on the SVTime. As will be described later, five combinations of these formats with different levels were used in the experiments, which resulted in various SVTime values.

For this model, the assumption is that only the business tier components, that are deployed on the application server may fail under the high traffic load. In practice, most of the load is on the application server and database server, depending on the nature of the deployed application. Since most database servers are empowered by recovery mechanisms, we did not consider this case in the experiments. The failure rate is generated by a probability distribution function for each component. For the experiments, the probability function was chosen as an exponential with a constant duration time. The restart time was also considered to be constant. However, because the model is required to be evaluated for a no-adaptation case as well, restart should also be available in that case (e.g., by manual administration). Therefore, a restart action has a longer T_{ac} (e.g., three times longer than the automated restart). The restart mechanism was designed as a micro-reboot mechanism, as discussed by Candea *et al.* [33]. The micro-reboot was modelled as a queue-server model that is enabled by the restart action signal. The failure, restart, and warmup states of the

component were modeled by a state-flow diagram in the Matlab StateFlow toolbox [132]. Interested readers can refer to [183] for more details on this state-flow diagram. In a set of experiments, a timeout mechanism is used to drop requests after a certain time. The timeout value is set at the traffic generator and each component has an output that reports the dropped requests.

6.3.2 Building the Adaptation Manager

Two adaptation mechanisms are developed for the experiments. The first one is based on the attribute-action-coupling pattern which, in this case, is implemented by rules (i.e., action policies). The rules are implemented using the Stateflow toolbox in Simulink [132]. The second adaptation manager is developed using GAAM. This approach is realized by the aid of the general modules of Simulink. The following sections elaborate further the details of this approach.

GAAM Specification

Table 6.6 lists the attribute set AT used in the experiments for the news application model. Throughput is calculated based on the number of requests served per second for the whole system. While the simulation gives us the response time and throughput for each component separately, only the end-to-end value is used for these attributes (respectively for at_2 and at_5). For the user load (at_3), due to a lower service time in the web tier components, the number of requests in the components' queue at the business logic tier is used. The component state at_4 is the state of a specific component that fetches requested news items from the back-end database, and is either active or failed. In practice, for example in the case of Java EE, this component can be implemented by an EJB that uses entity objects and Java Persistence API (JPA) to work with a database.

Table 6.6: Attributes in news application

Attribute	**Values**
News quality (part of at_1)	Video resolution: {High, Low}, Image size {Normal, Small}
News data type (part of at_1)	{Video, Image, Text}
End-to-End Response time (at_2)	Zero or positive real values
User load (traffic) (at_3)	Zero or positive integer values
Component state (at_4)	{Active, Failed}
End-to-End Throughput (at_5)	Zero or positive real values

For the sake of design simplicity, the data type and data quality attributes were combined into one attribute, at_1. Table 6.7 lists different values that at_1 can get during the

simulation. Switching to lower values increases the response time and throughput, while it decreases the level of satisfaction for the best data type and quality goals.

Table 6.7: Data type/quality values for at_1 in news application

Data Type/Quality	**Definition**
TNH	Text + Normal size Image + High resolution Video
TNL	Text + Normal size Image + Low resolution Video
TN	Text + Normal size Image
TS	Text + Small size Image
T	Text

Figure 6.7 depicts the goal hierarchy in the conducted experiments. Low-level goals are the ones related to measurable attributes. Maximum availability is translated to min Mean Time To Repair (MTTR), as stated by Candea *et al.* [33]. As the authors discussed in [33], reducing MTTR can be as effective as increasing Mean-Time-To-Failure (MTTF) in maximizing availability. As seen in this figure, higher level goals may share sub-goals. For example, end users want the service to be available, quick, and of high quality.

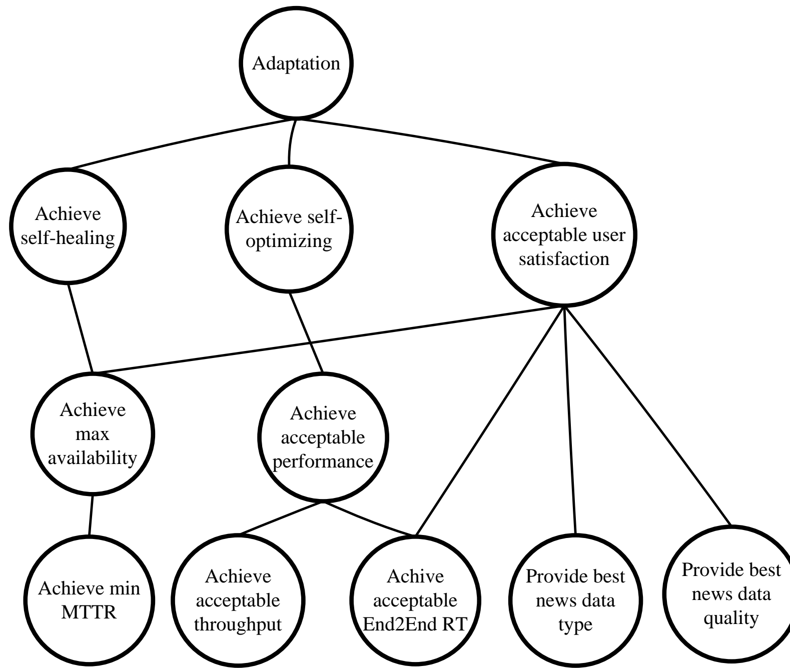

Figure 6.7: Goal hierarchy of the experimental model in news application

Table 6.8 lists low-level goals in the goal hierarchy as goals in GAAM, along with their parameters such as θ_i, p_i and τ_i. The value of τ_i has the measurement unit of its corresponding at_i. This is the reason for the different value types in Table 6.8 for τ_i. The

summation of p_i is set to 100. The maximum throughput goal will not be activated unless both throughput and load are high enough to exceed their τ_i values. For this purpose, the θ_i value for this goal is higher than the other goals.

Table 6.8: Goals in GAAM - News application

Goals	**Major Properties**
Min (MTTR) (g_1)	p_1=30, θ_1=0, τ_{14}='active'
Best news data type (g_2)	p_2=10, θ_2=0, τ_{11}= not ('TNH' or 'TNL')
Best news quality (g_3)	p_3=15, θ_3=0, τ_{31}= not ('TNH' or 'TN')
Min end-to-end response time (g_4)	p_4=25 , θ_4=0, τ_{42}=400ms, τ_{43}=100
Max throughput (g_5)	p_5=20 , θ_5=1, τ_{53}=100, τ55=30

Six adaptation actions are used in the experiments, namely i) Restart (ac_1), ii) Switch to TNH (ac_2), iii) Switch to TNL (ac_3), iv) Switch to TN (ac_4), v) Switch to TS (ac_5), and vi) Switch to T (ac_6). For these experiments, it is assumed that except ac_1 (restart), all other actions are always applicable. The precondition for restart is a "failed" state in a component. In the initial model, preconditions for the other actions were considered to change only from one state to an adjacent state; for example, to change from TNL to only TN or TNH. However, for two reasons these preconditions were removed. The first reason is that when the server changes the service level, for instance from TNH to T, a single user may not even notice this change because of the service time and the thinking time for the next request. Therefore, a jump in the service level is not observed except probably in extremely high traffic, which would lead to a crash in the system. The second reason is that these preconditions increase the system's adaptation time T_{ad}, thus taking several actions to go from one extreme service level to the other. In essence, there is a high probability that users would not see the service level switches and therefore, changes can be made to the service levels to decrease the adaptation time.

Goal Preferences

Two types of preference structure (utility function) are used in the experiments, *ordinal* and *cardinal*. For comparing ordinal and cardinal types of preferences, we assumed that the order of ac_i in both cases are identical. It means that the descending sorted list of cardinal preferences is the same as the ordinal preferences. As stated in research question RQ2, one of the goals is to compare these utilities in the simulated model. Table 6.9 shows the ordinal preferences defined for this case study. In the ordinal case, the Borda count voting method is used.

Table 6.10 shows the values of preferences in the cardinal utility case. The order of actions for each goal is identical to the ordinal case. These values were tuned in several experiments, and this set represents one of the best settings with this order.

Table 6.9: Ordinal preferences in the news application

Order/Goal	Min MTTR	Best Data Type	Best Quality	Min End2End RT	Max Throughput
4	Restart	TNH, TNL	TNH, TN	T	T
3		TN, TS	TNL	TS	TS
2		T	TS	TN	TN
1			T	TNL	TNL
0	TNH, TNL, TN, TS, T	Restart	Restart	TNH, Restart	TNH, Restart

Table 6.10: Cardinal preferences in the news application

Action/Goal	Min MTTR	Best Data Type	Best Quality	Min End2End RT	Max Throughput
Restart	10	0	0	0	0
TNH	0	3	3	0	0
TNL	0	3	1.75	0.5	0.5
TN	0	1.75	3	2.75	2.5
TS	0	1.75	1.25	3	3
T	0	0.5	1	3.75	4

In the experiments with unequal priorities, several settings were evaluated for p_i values. In these cases, the statistical results were not different. As will be mentioned in the future works section, a complete sensitivity analysis for these priorities is planned. For the experiments, PV is the set with values of $\{10, 2, 2, 4, 1\}$; p_1 is set to 10 so that its value is more than the summation of all other priorities. This is to ensure that the failure state will always be handled first. In other words, self-healing is more important than the other goals. p_2 and p_3 are equal, that means the best quality and best data type goals are equally significant for users. p_4 is the priority value of achieving the best response time and is set to be the second highest value in the PV set.

6.3.3 Design of Experiments

A factorial design was used for the experiments. Four treatments were considered for the experiments: no adaptation (**NoAdapt**), goal-ensemble adaptation with ordinal preferences (**Ordinal**), goal-ensemble adaptation with cardinal preferences (**Cardinal**), and rule-based adaptation (**Rule**). In the NoAdapt treatment, except for the manual restart, there are no actions to change the news application. For the Rule treatment the attribute-action-coupling composition pattern is employed, and the adaptation mechanism is implemented using 13 rules. The rule-based treatment was chosen based on the fact that most practical solutions use a form of this approach. By conducting these experiments, two mechanisms based on goal-centric pattern, Ordinal and Cardinal, are compared with a mechanism designed by adopting the attribute-action-coupling pattern, Rule.

In this experimental evaluation, the following research questions are taken into account:

- **CS2-RQ1:** What is the impact of adaptation on the system goals? Does it improve the goal satisfaction level comparing with the non-adaptive case?
- **CS2-RQ2:** Which type of utility is better for determining preferences? While the cardinal utility needs more effort to be extracted and tuned, does it necessarily lead to a more effective adaptation?
- **CS2-RQ3:** Does the proposed GAAM outperform the rule-based mechanism?

Each of these questions aims at comparing two terms. To make this possible an evaluation function is needed. The function, as described in Section 5.3, is aimed at assessing the deviation from the satisfied state of each goal, using a formula similar to the goal programming method [83]. The deviation for each goal is evaluated by $e_j = 1/m \sum_{i=1}^{m}(d_i^+ + d_i^-)$. The normalization factor $(1/m)$ has a value between 0 and 1. The objective is to minimize this deviation measure (zero is the ideal value). For each run k, the global evaluation function is $E_k = 1/n \sum_{j=1}^{n} p_i * u_j$.

Because workload characteristics impact the system quality and its evaluation function, load intensity is used as a blocking factor. The workload factor can accommodate three levels depending on the average IATime of requests: medium, high, and very high. All of the traffic patterns in the experiments start with a default workload (i.e., a default IATime). This workload then drops after 10 seconds to one of the preceding levels in order to simulate burst traffic.

In each experiment, there is an adaptation factor identifying treatments and a load intensity blocking factor. For each load level and treatment, the experiment has three replications. The treatments are evaluated based on three different conditions: i) with requests generated from two probability distribution functions, Exponential and Weibull, ii) with equal and unequal priorities, and iii) with requests including a timeout value. Overall, four experiments were conducted under the following conditions:

- **Exp1**: exponential traffic, equal priorities, and no-timeout
- **Exp2**: Weibull traffic, equal priorities, and no-timeout
- **Exp3**: exponential traffic, unequal priorities, and no timeout
- **Exp4**: exponential traffic, unequal priorities, and with timeout

Because the timed-out requests are dropped by the application components in Exp4, the evaluation function does not result in a comparable value with that of the other three experiments. In fact, it should also take into account a penalty for dropped requests. Designating such a penalty value is not an easy and straightforward task. Although several attempts were made to incorporate the penalty, it was decided to consider a loss ratio (the percentage of dropped requests) along with the evaluation function.

6.3.4 Obtained Results

The results are first checked using the normality and variance homogeneity tests. The variances are homogeneous, but the distribution is not normal based on the QQPlot and the Levene test. Even the data transformations did not lead to the passing of both tests. Therefore, a non-parametric one-way ANOVA method with blocking is used for statistical analysis. The Friedman test [161] performs such a test by running a Chi square on the ranked data. In this test, data in each block is initially ranked and then an analysis is performed on the ranked data. Table 6.11 shows the results for the Chi square test and the GLM ANOVA F-test in Exp1-3. These findings indicate that the treatments are not similar in these three experiments. To scrutinize the results, it is essential to look at the Box plots and pairwise comparison of the treatments.

Table 6.11: Friedman ANOVA test for news application- Row mean score difference (CHi square), and GLM F-test

	Row mean score difference		**GLM F-test**	
	Value	Probability	F value	$Pr > F$
Exp1	23.0171	<.0001	13.83	<.0001
Exp2	18.7436	0.0003	7.89	<.0001
Exp3	18.812	0.0003	7.96	<.0001

Figure 6.8 depicts the Box plots for the evaluation functions in Exp1-3. In Exp1, the difference between the range and median of the goal-based treatments (Ordinal and Cardinal) with the other two treatments (Rule and No-Adapt) is obvious. However, in Exp2 and Exp3, the Rule treatment is closer to the goal-based treatments. It appears that there is no significant difference between the goal and rule-based treatments (addressing *C2-RQ3*).

In order to statistically compare these treatments, several Dunnett tests were run. In fact, in each test one treatment is the control treatment, which forms the basis for the comparison. Table 6.12 shows the summary of these comparisons in which duplicate tests were removed. In all three experiments, Ordinal and Cardinal are among the best in

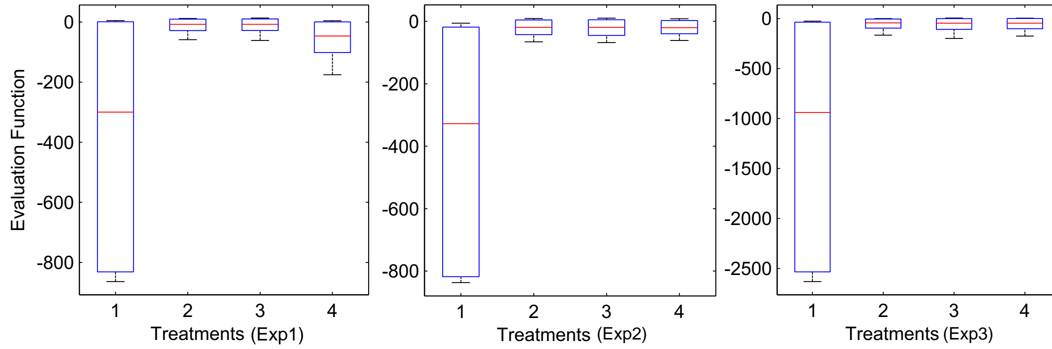

Figure 6.8: Evaluation function of treatment in Exp1-3 (1: NoAdapt, 2:Ordinal, 3:Cardinal, 4:Rule)

Table 6.12: Dunnett test for pairwise comparison of treatments (*** means significantly different)

	Treatment Comparison	**Mean Differences**	**Simultaneous 95% Confidence Limits**	**Significant ($\alpha = .05\%$)**
Exp1	cardinal-noAdapt	4.7778	(2.9832 , 6.5723)	***
	ordinal-noAdapt	4.2222	(2.4277 , 6.0168)	***
	rule-noAdapt	2.1111	(-0.3143 , 4.5365)	
	cardinal-ordinal	0.5556	(-1.239 , 2.3501)	
	cardinal-rule	4.7778	(2.3524 , 7.2032)	***
	ordinal-rule	4.2222	(1.7968 , 6.6476)	***
Exp2	cardinal-noAdapt	4.4444	(2.6291 , 6.2598)	***
	ordinal-noAdapt	4.5556	(2.7402 , 6.3709)	***
	rule-noAdapt	5.778	(2.879 , 8.676)	***
	cardinal-ordinal	-0.1111	(-1.9265 , 1.7043)	
	cardinal-rule	0.333	(-2.565 , 3.232)	
	ordinal-rule	0.333	(-2.565 , 3.232)	
Exp3	cardinal-noAdapt	4.5556	(2.7402 , 6.3709)	***
	ordinal-noAdapt	4.4444	(2.6291 , 6.2598)	***
	rule-noAdapt	6.333	(3.442 , 9.225)	***
	cardinal-ordinal	0.1111	(-1.7043 , 1.9265)	
	cardinal-rule	-0.556	(-3.447 , 2.336)	
	ordinal-rule	-0.444	(-3.336 , 2.447)	

terms of the evaluation function. In Exp1, Cardinal and Ordinal are significantly different from the Rule and NoAdapt treatments. Figure 6.8 reveals that the Cardinal and Ordinal treatments are separate and distinct from the NoAdapt treatment. However, the Rule treatment falls somewhere in between. Response to *C2-RQ1*, based on these results, is that adaptation improves satisfaction-level of goals.

The Simultaneous Confidence Level (SCL) in Table 6.12 indicates that even though the limits are completely skewed to the Rule side, these three treatments are not significantly different. Because the evaluation function of the Cardinal, Ordinal, and Rule treatments were so close in Exp2 and Exp3, the test was repeated with $\alpha = 0.01$. However, the differences are still not significant. Therefore, in response to *C2-RQ2* and *C2-RQ3*, statistical results suggest no difference. Overall, it appears that if the treatments need to be sorted using the Box plots and the SCL values, Cardinal is slightly better than the others, especially in Exp1 and Exp3. The next treatment is Ordinal (which in Exp2 is even better than Cardinal), then Rule, and finally NoAdapt treatments.

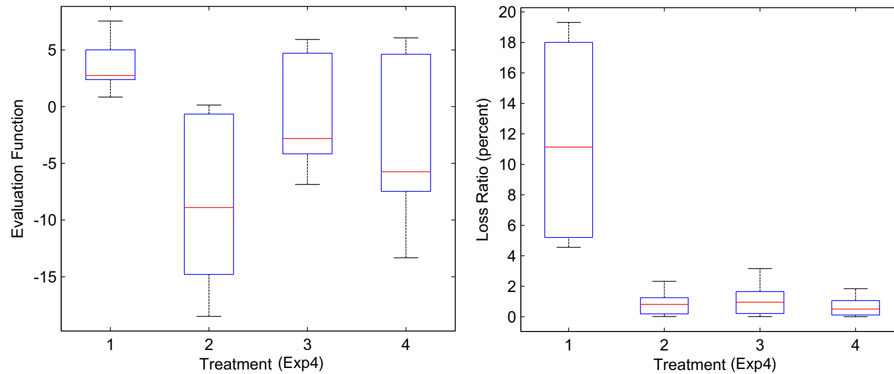

Figure 6.9: Evaluation function and loss ratio in Exp4 (1: NoAdapt, 2:Ordinal, 3:Cardinal, 4:Rule)

The fourth experiment generated some interesting results, because there is a tradeoff between dropping requests and satisfying the goals. As depicted in Figure 6.9, the NoAdapt treatment outperforms the others in terms of the evaluation function, but at the cost of dropping many more requests (median 12%). In fact, regarding both Box plots, it appears that the Ordinal treatment outperforms the others as it results in a relatively low loss ratio and a moderate evaluation function value. Therefore, Exp4 verifies the results of the previous experiments regarding to *C2-RQ1*, *C2-RQ2*, and *C2-RQ3*.

6.3.5 Lessons Learned

The experimental model has been built using a queue-server network for a news web application, but it is a generic model that can be used for other multi-tier mission-critical and enterprise applications as well. The initial experimental model was first used to evaluate GAAM in [185]. Then the same model was used to compare GAAM with Reinforcement Learning (RL) as an attribute-action-coupling mechanism [7]. Both works show that GAAM has no negative impact on the system behavior in any tested situation, and it can even improve the performance and availability in some cases. A more significant outcome of both works is that the development and tuning effort of GAAM are less than the other approaches. The is mostly owing to the less complexity in GAAM.

Three research questions (*C2-RQ1* to *C2-RQ3*) are investigated in the empirical evaluation. Obtained results suggest the following answers to these questions:

- *C2-RQ1* compares the effectiveness of adaptation mechanisms with NoAdapt treatment. The findings suggest that the adaptation mechanisms impact positively on the evaluation function formulated based on goal satisfaction levels. Therefore, these two sets of treatments are significantly different.

- *C2-RQ2* compares the ordinal and cardinal utilities to figure out which one could be more effective and which one needs less effort to be built. Both goal-ensemble mechanisms, using ordinal and cardinal utility functions, perform well in the experiments. Interestingly, the Ordinal treatment behaves better than expected. By considering the effort required to tune the cardinal preferences, the ordinal preferences are easier to elicit and lead to results that are as good as in the cardinal case. In short, in response to this question although the Ordinal and Cardinal treatments do not show a significant statistical difference, the ordinal utility is a better choice regarding the elicitation and tuning effort.

- *C2-RQ3* contrasts the goal-ensemble and rule-based adaptation mechanisms in terms of their effectiveness. The arguments in answering previous questions can be applied to this question as well. While the goal-ensemble and rule-based adaptation mechanisms are not significantly different in terms of statistical measures, formulating and refining the rules are not always straight-forward. This issue is especially remarkable in the case of having many attributes and actions .

Two important concerns of validation in these experiments are internal and external threats. This question is to what extent these concerns threaten the validity of the conducted experiments, particularly in terms of the classification provided by Shadish *et al.* [191]. Because the conducted experiments are controlled, replicated, and independent

from each other, most of the *internal validity threats* such as testing, history, maturation, and instrumentation threats do not exist in this work. This means the relationships between dependent and independent variables are not affected by other factors and time. There are some concerns that may be attributed to internal validity threats, which are not believed to be critical, and can be addressed with reasonable effort. For instance, one point that affects the internal validity of the experiment is the value of T_{ad}, the period of adaptation. A fixed value was used for this parameter, but we observed that the value can have a remarkable impact on the evaluation function. One possible solution would be to use a dynamic value, for example, as suggested by Menasce *et al.* [139].

Generally, in the experiments conducted on the news application there is no dependency on subjects, treatments, measurement methods, and the environment, which cause *external validity threats*. However, there are some issues about generalizing the model to other domains. The GAAM and the goal-ensemble mechanism are evaluated by a simulated model of a news web application. The experimental model is a generic model for multi-tier web applications and fundamental changes in the results are not expected for similar applications.

It is assumed that for the news web application there is no session state for users in the experimental model. Even though the system does not change the service level for the users using any type of service, restarting components can lead to loss of session information. This feature is not essential for a news application, but is required for some other mission-critical applications. Therefore, persistency of this information needs to be added at least for the restarting action. Therefore, by considering stateful cases, the approach is generalizable in this domain. A notable point about goals is that not only satisfying goals, but also keeping them satisfied (i.e., protecting goals) is important. As mentioned before, this issue is not critical for this experimental model and for many server-side applications, but it should be taken into account that changing goal states rapidly can cause problems for the end users. The current implementation of GAAM is memoryless, and previous states should be kept persistent for protecting goals.

6.4 Case Study 3 (CS3): Service-oriented VOIP Call Controller- CC2

With the growth of Internet, Voice over Internet Protocol (VoIP) and internet based telephony systems are gaining more popularity. Such systems, introduce various affordable services for voice, video, and messaging communications as an alternate to traditional communication systems.

6.4.1 CC2 VoIP Call Controller

"Call Controller 2" (CC2) [3], a VoIP prototype application, is chosen for the case study of building a self-adaptive software system. CC2 has Service Oriented Architecture (SOA), and is composed of several basic telephony services that are implemented and deployed on Mobicents[4] platform.

Mobicents is the first and only open source implementation of Jain Service Logic Execution Environment (JSLEE) [5] on top of JBoss Application Server (AS). Jain is a new specification for low-latency event-driven communication, and mobicents as an implementation for its SLEE is a platform to execute event-driven service applications that require low-latency communication, like telephony systems and online gaming. From users' perspective, Mobicents complements Java EE to enable easy development of services such as voice, video, instant messaging, and data in next generation of intelligent applications. Figure 6.10 shows the high-level architecture, including JBoss microcontainer and the deployed components [6].

Figure 6.10: JBoss and JSLEE [130]

Developers can create, deploy and manage services and applications integrating voice, video and data across a range of IP and communications networks on Mobicents[7]. Technically speaking, CC2 deploys its service logic on Mobicents, providing four main services to users [8]:

[3]http://groups.google.com/group/mobicents-public/web/jain-slee-example-call-controller-2?pli=1
[4]http://www.mobicents.org/index.html
[5]http://java.sun.com/products/jain/article_slee_principles.html
[6]http://code.google.com/p/mobicents/
[7]http://code.google.com/p/mobicents/
[8]http://groups.google.com/group/mobicents-public/web/jain-slee-example-call-controller-2

1. Regular call service: It is the most basic service provided by all VoIP applications. A caller can call a callee to establish a conversation.

2. Forwarding service: If a callee is unavailable, CC2 tries to forward the call to callee's backup address (if available).

3. Blocking service: If a caller is in callee's blacklist, the call will be blocked.

4. Voicemail service: If a callee is unavailable and it has no backup address but its voicemail is enabled, CC2 will record the voice message of the caller.

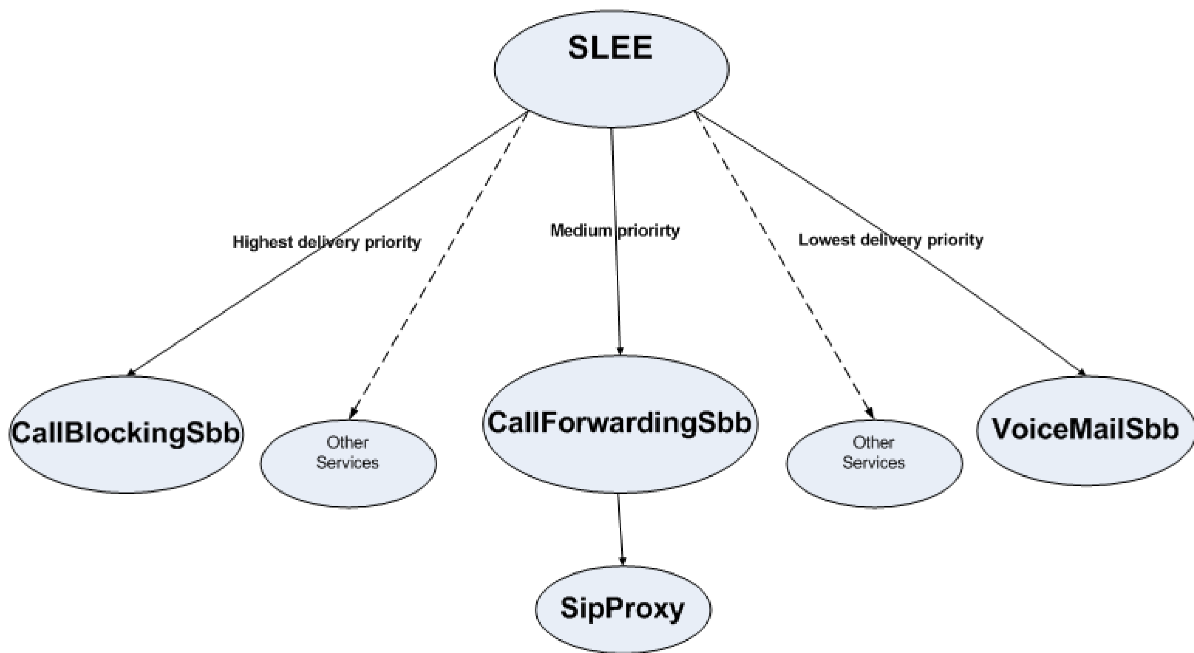

Figure 6.11: SLEE services provided by CC2 [130]

In Mobicents, services are provided by *Service Building Blocks (SBB)*. In fact, Mobicents is a container for deploying SBBs which provide different services to users. CC2 consists of three main SBBs: "*Forwarding SBB*", "*Blocking SBB*" and "*Voicemail SBB*", as depicted in Figure 6.11. Both "Regular VoIP calls service" and "Forwarding service" are provided by "Forwarding SBB". Session Initiation Protocol (SIP)[9] is the protocol for es-

[9]Session Initiation Protocol (SIP) is a signalling protocol, widely used for setting up and tearing down multimedia communication sessions such as voice and video calls over the Internet. http://www.ietf.org/rfc/rfc3261.txt

tablishing each communication, and Real-time Transport Protocol (RTP)[10] delivers audio and video packets.

Figure 6.12 illustrates a typical VoIP communication using SIP and RTP. The communication steps include:

1. A client (SIP Phone A) sends an "INVITE" SIP request to the server.
2. The server analyzes the request and redirect it to the callee (SIP Phone B).
3. If phone B accepts the call, he/she will reply a "200 OK" SIP message back to the server.
4. The server redirects "200 OK" to phone A, and phone A sends an "ACK" SIP message.
5. Phone B receive "ACK" from the server, now the communication has been established and they (phone A and B) start to use RTP to directly communicate with each other.
6. When someone wants to terminate the call, he/she sends a "BYE" message to the server.
7. The server lets the other user know the termination request, and the communication is ended.

Two different CC2 systems are used in the empirical studies. One is the original CC2 system ($CC2_{original}$), without any modifications on functional requirements. The other one is the modified CC2 system ($CC2_{modified}$), in which all functionalities of CC2 are preserved, while three user levels are augmented to the system. Thus users of $CC2_{modified}$ are classified into three levels from high to low, namely Gold, Silver, and Bronze. These user levels are based on the specified service levels in the Service Level Agreement (SLA), which is described in Section 6.4.3. The service provider is required to give higher service quality (e.g., lower response time) to higher user levels. It means Gold users have better service quality than Silver ones, and consequently Silver users better than Bronze ones.

Commonly, all users can access all services. However, under some situations, low-level users may not access some or all services in order to serve high-level users. Serving Gold and Silver users will produce more profit than serving Bronze users, of course depending on the number of users in each level. Therefore, the adaptation manager tries to maximize the satisfaction of high-level users. To reflect this, the goals related to high-level users have

[10]Real-time Transport Protocol (RTP) is a standardized packet format for delivering audio and video over the Internet. http://www.ietf.org/rfc/rfc1889.txt

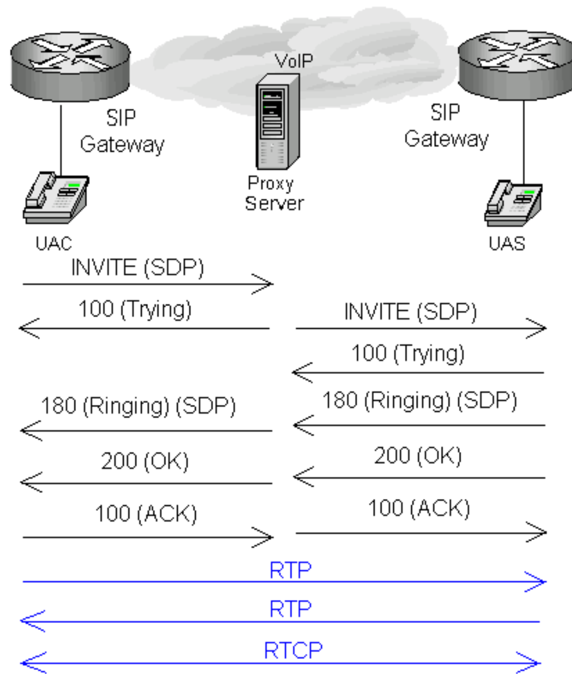

Figure 6.12: A typical VoIP communication using SIP and RTP [68]

higher weights (priorities) than the goals corresponding to low-level users. Adding the user level is due to the fact that real-world service providers often use these levels. In addition, GAAM can be evaluated in presence of more goals and actions.

Figure 6.13 illustrates the high-level architecture of the experimental system. In this figure, JBoss Application Server (AS) is the underlying platform. Mobicents is deployed on JBoss AS using features like JMX and JMS. CC2 including three SBBs is deployed on Mobicents. Each SBB is binded to a Management Bean (MBean), which can expose attributes of the SBB (i.e., a sensor), and can make changes on the SBB at run-time (i.e., an effector). The sensors and effectors are elaborated further in Section 6.4.2. These MBeans are instrumented by JMX and can be accessed via MBean server. GAAM is deployed on StarMX [13] and uses it to communicate with JMX and MBeans. This architecture is used for both $CC2_{original}$ and $CC2_{modified}$ systems.

6.4.2 Building Adaptable Software

In this case study, the adaptable VoIP call controller is built by re-engineering CC2 application by instrumenting a set of sensors and effectors. This section briefly discusses this process.

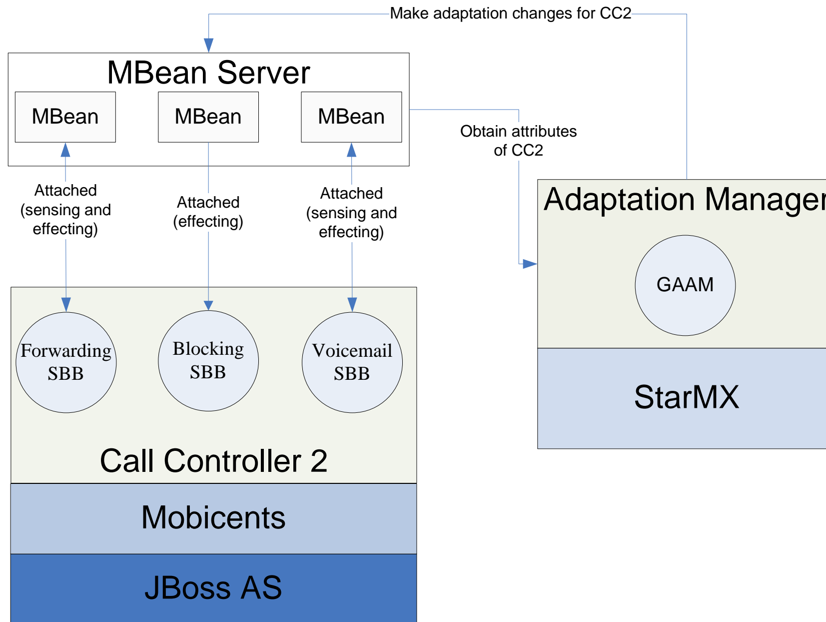

Figure 6.13: Experiment setting for CC2 system

Three main attributes of response time, throughput, and arrival rate are measured for each service in $CC2_{original}$ (see at_1 to at_7 in A.2). In $CC2_{modified}$, these attributes are monitored for each user level as well (see at_1 to at_{21} in A.3). All average response times, throughput values, and arrival rates are obtained by calculating the moving average of sensed data. Two notable points about the sensors are as follows: i) "Blocking SBB" has no sensors, because the performance of "Blocking service" does not impact the satisfaction level of users, ii) there are three performance metrics for "Regular VoIP Calls" and "Voicemail service" but only one for "Forwarding Calls". The reason is that "Regular Calls" and "Forwarding service" are both provided by "Forwarding SBB". Therefore, they share the same throughput and arrival rate.

There are also some attributes that are not used to adapt the system, but are required for evaluating the quality of adaptation. For example, the number of served users for each service is important to determine how well the system behaved in the experiments. This information can be logged, so these are not considered as sensors. Later, in Section 6.4.3, these attributes will be discussed.

The list of adaptation actions for $CC2_{original}$ and $CC2_{modified}$ systems can be found in A.4 and A.6, respectively. One may ask why "Blocking service" seems to have no role in this action list. This is because disabling "Blocking service" will not improve the performance of the system. In fact, blocking users will decrease the number of threads in

the system, so that system can serve more of other users. Thus, there is no harm to always let the “Blocking service” be enabled.

Three Management Beans (MBean) are created to be attached to three SBBs. MBean is a Java object that represents a manageable resource, such as a service or a component [89]. These MBeans, which are instrumented by JMX, can expose sensors and effectors of the manageable resource to outside via JMX interface. More specifically, these MBeans can be treated as the actual sensors and effectors of CC2.

6.4.3 Building the Adaptation Manager

The objectives of the system is to satisfy all of the adaptation goals, if possible. The main strategy is to prevent the system from entering the buckle zone, which is above the nominal capacity. In fact, the adaptation actions increase or decrease the system capacity by playing with the settings of services. When the response time is high and the throughput is low, the system is better can decrease its service levels to increase its capacity. After the capacity is increased, the response time and throughput can remain normal. All in all, the objective is to show that the system can properly increase its capacity by sacrificing some forwarding and voicemail requests from low-level users. Under a extreme heavy workload, the system may even block all the services for a short time to prevent a crash.

For the experiments, two adaptation managers are designed based on attribute-action-coupling and goal-centric patterns . The former implements a rule-based mechanisms using action policies, while the later employs goal-ensemble by GAAM. These two mechanisms are implemented with the aid of StarMX framework. The following section elaborates the details of GAAM development for this case study. It is notable that goals are also implicitly considered in the rule-based mechanism.

Goal Structure

$CC2_{original}$ has 10 leaf goals, listed in Section A.5, and $CC2_{modified}$ has 28 leaf goals, listed in Section A.7. These goals are commonly mentioned in SLA. The SLA used in the experiments can be found in Section A.1. A goal activation level is set based on its corresponding SLO in SLA. For example, one of these SLOs can be “if the response time of Regular Gold VoIP calls is greater than 5 sec, the service provider will need to pay $1 penalty per user per exceeding second to the service receiver”. This simple SLO can directly induce a goal, which is “achieve acceptable response time”, and its activation level for this goal can be “if the response time is greater than 5 sec”.

Low-level goals are decomposed from high-level goals of stakeholders. The two top-level goals in this case study are “self-optimizing” and “maximum user satisfaction”. The former

is from developers' perspective to maximize the performance of the system, while the latter one is from the users' perspective to maximize the satisfaction of users. Goals can also be presented as a hierarchy, from high-level stakeholders' abstract goals to low-level goals. Figure 6.14 illustrates the goal hierarchy in $CC2_{original}$. In $CC2_{modified}$, lower level goals are replicated for each user level.

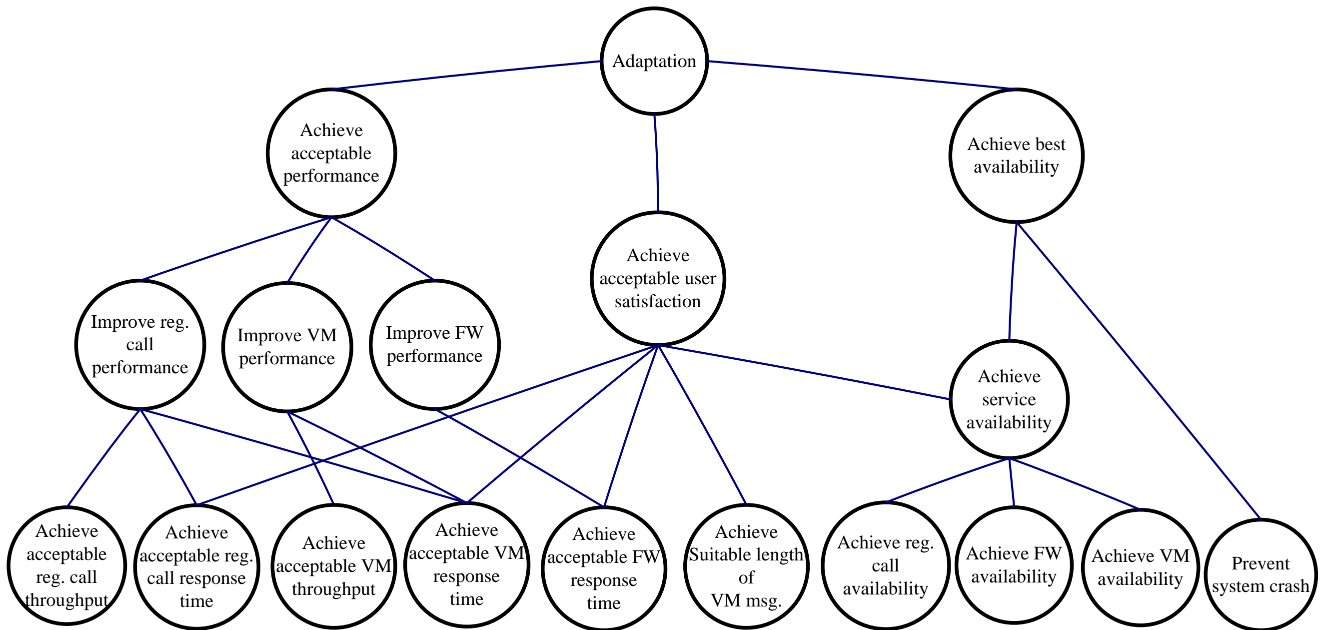

Figure 6.14: Goal hierarchy for CC2 system

Preference Elicitation for Goals

Preference elicitation for goals involves two problems: first, how to prioritize goals or assign them weights, and second, how to elicit action preferences for each goal regarding the adaptation requirements specifications.

The weight of each goal can be assigned based on its importance in SLA and QoS. For example, when the response time of normal calls exceeds 1sec the service provider will be penalized $10, whereas if the response time of forwarding calls exceeds 1sec the penalty will be $1. Then, goal "achieve acceptable response time of regular calls" has a higher weight value than "achieve acceptable response time of forwarding calls". The question is, how much higher should it be? Each goal requires a specific weight rather than just a

ranking. One feasible solution is setting up some initial weights first and then adjusting them during experiments to achieve better performance. However, if a new goal is added, all weights may need to be adjusted.

On the other hand, the action preference list for each goal can be elicited based on their impacts on satisfying the goal, that is in other words how much the actions can lead to deny (i.e., activate) the goals. If there is an analytical model of the adaptable software system, these impacts can be determined. However, in practice, for complex distributed systems this is a time-consuming and tough task, if not impossible. Instead, in the experiments on CC2 case study, dynamic analysis is used to estimate the impact of each action on goals with the aid of domain knowledge (i.e., a priori knowledge). This approach is also time-consuming but at least more feasible for this system and larger applications. Elicited goal preferences for actions and weights in this case study are listed in Section A.7.

Evaluation Function

In order to evaluate the self-adaptive VOIP call controller, it is required to take into account all the quality requirements specified by SLOs in the system SLA, and accordingly the derived adaptation goals. Certainly, every user desires to have the best possible service level designated for his/her type (e.g., Gold). If the application can provide all these in any situation, then no adaptation scenario will be required. Adaptation action makes tradeoff between service levels and the cost of services. In fact, self-adaptive software considers all stakeholders' goals to take an appropriate action. For example, when context state is normal, users can definitely access all services. However, if the workload is high and the system is slow, users might want to disable the voicemail service to lower their RT. For CC2 case study, the evaluation function (E) is defined regarding to leaf goals, and depends on both time-dependent attributes and the number of successful and failed requests. E is formulated based on the following parameters:

- Regular VoIP calls
  - Completed Regular Calls: Successful regular calls for each user level - Φ^{Reg}_{Gold}, Φ^{Reg}_{Silver}, and Φ^{Reg}_{Bronze}.
  - Regular Call Response Time: Individual Response Time for each successful Regular Call user- RT^{Reg}_{Gold}, RT^{Reg}_{Silver}, and RT^{Reg}_{Bronze}.
- Forwarding calls
  - Completed Forwarding Calls: Successful Forwarding calls for each user level - Φ^{Fw}_{Gold}, Φ^{Fw}_{Silver}, and Φ^{Fw}_{Bronze}.

– Forwarding Call Response Time: Individual Response Time for each successful Forwarding Call user - $RT^{Fw}_{Gold}, RT^{Fw}_{Silver}$, and RT^{Fw}_{Bronze}.

- Voicemail calls
  – Completed Voicemail Calls: Successful Voicemail calls for each user level - $\Phi^{VM}_{Gold}, \Phi^{VM}_{Silver}$, and Φ^{VM}_{Bronze}.
  – Voicemail Call Response Time: Individual Response Time for each successful Voicemail Call user- $RT^{VM}_{Gold}, RT^{VM}_{Silver}$, and RT^{VM}_{Bronze}.
  – Timeouts: The total number of Timeouts to force users to terminate recording voicemails due to exceeding the maximum recording time - $TO^{VM}_{Gold}, TO^{VM}_{Silver}$, and TO^{VM}_{Bronze}.

The E function considers the number of completed Regular Calls as the most important metric. The response time of Regular calls is the second most important metric. Because regular VoIP call is the most basic and frequently used service, clients expect it to be the most reliable and quickest. Forwarding service is more important than Voicemail service, since Forwarding service may lead to a successful regular call. The sample SLA of the system can be found in Appendix Section A.1. The information of this SLA will be extracted to set up the following equation. Assume that Φ^s_l is the set of completed calls for service s and user level l, then $\Delta\Phi^s_l$ is defined as follows:

$$\Delta\Phi^s_l = |\Phi^s_l| \forall \phi \in \Phi^s_l, \forall s \in \{Reg, Fw, VM\}, \forall l \in \{Gold, Silver, Bronze\}, RT^s_l > RT^s_{lSLO} \tag{6.2}$$

then E for $Exp1$ will be:

$$E_{Exp1} = \sum w_i * |\Phi^s| - \sum w_j * |\Delta\Phi^s| - \sum |TO^{VM}|, w_i \in W \tag{6.3}$$

where W is the vector of weights and there is no user level l in $Exp1$. For $Exp2$, parameters are the same, but for all user levels. In this experiment, E_{Exp2} is defined as:

$$E_{Exp2} = \sum w_i * |\Phi^s_l| - \sum w_j * |\Delta\Phi^s_l| - \sum |TO^{VM}_l| \tag{6.4}$$

The E function involves several metrics from the system, while it still misses one important metric: "what is the quality of voice in the communication?" Some research works in VoIP communication suggest setup metrics and index to quantify the quality of VoIP (e.g., see [213]). However, it was not possible to provide the essential hardware/software support to measure these metrics and indices. Incorporating the quality of VoIP in E function can be a potential direction to extend this work.

6.4.4 Design of Experiment

Experiments are designed as a one-factor with blocking. The experiment factor has the three following treatments:

- **NoAdapt** treatment: No adaptation mechanism is enabled and CC2 functions as a non-adaptable application. This is actually the control treatment, or baseline, for comparisons.
- **GAAM** treatment: This is a goal-centric adaptation mechanism based on GAAM. Each activated goal (voter) can vote for its preferred actions every 1 min ($T_{ad} = 60sec$). Therefore, due to the GAAM implementation for this case study, one adaptation action can be performed at most every T_{ad} =1 min.
- **RuleBased** treatment: This is an attribute-action coupling adaptation mechanism realized by action policies. These policies, which are defined regarding the attribute-action links, are executed on an external engine, in this case IBM ABLE [22].

The effect of workload is enormous on adaptation effectiveness and the experimental errors can be reduced by considering different types of workload as blocks. These workloads are defined based on CC2 capacity. The capacity can be defined in different ways, including "expected users point", "SLA exceed point", and "saturation point" [69]. While the evaluation function is basically defined by violating the SLA, blocks are defined regarding the saturation point. According to Haines, this is defined by considering a typical system behavior depicted in Figure 6.15 [69].

As the load increases– number of users or requests (less inter-arrival), or even the service time for requests– the resource utilization increases up to a saturation point. Likewise the throughput increases, but declines after the saturation happens. After the saturation point, requests are left pending and response time increases. If the load continues to increase after this point, the response time degrades exponentially. The point at which performance time degrades is referred to as the saturation point, or the buckle zone [69]. For this case study, the capacity and consequently workload are defined in terms of the saturation point. The saturation point and the system performance behavior can be accomplished with analyzing the system performance model or performance testing (i.e., dynamic analysis). The second approach is utilized for CC2.

The blocks are defined based on different workloads for the system. Four blocks of normal, medium, heavy, and extreme are assigned considering the capacity and saturation

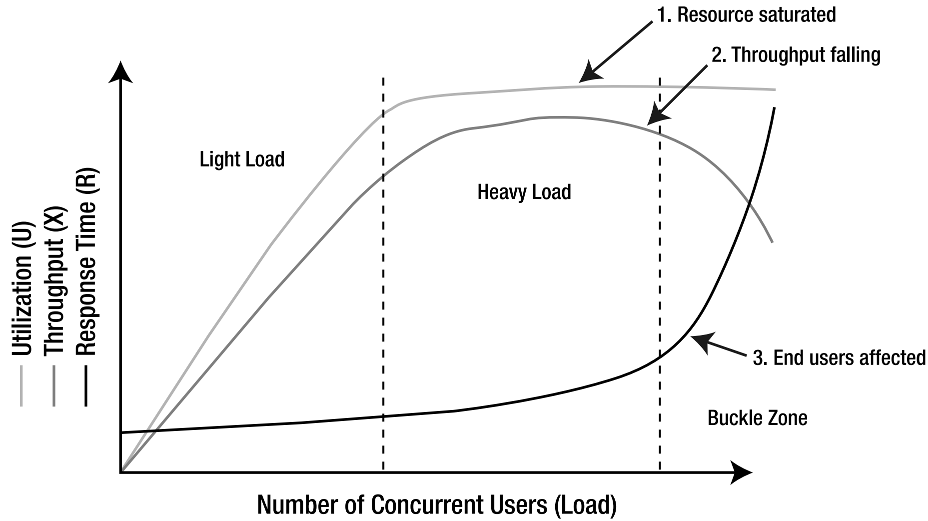

Figure 6.15: Capacity of system in terms of response time and throughput graph [69]

point of CC2. Each workload is designed according to the randomness of users' behavior to use VoIP services, regular call, forwarding and voicemail. The users may use these services in different orders randomly. The four blocks *B1* to *B4* for the experiments are corresponding to the following workloads:

- *Light workload (**B1**)*: The aim of evaluating the system under the light workload is to show that the adaptive system will not cause significantly extra overhead when the workload is below the nominal capacity of the original system. Under such a workload, the system normally handles all requests with a normal response time and normal error rate. This workload is somewhere in the middle of the light load zone in Figure 6.15.

- *Medium workload (**B2**)*: This workload is close to the border of the light load and heavy load in Figure 6.15, and is much closer to the nominal capacity of system than the light workload. However, since the capacity is not reached yet, the system should still work properly without decreasing performance. Evaluating the system under this workload is aimed at investigating whether the adaptation mechanism has a negative impact on this zone. In other words, the objective is to investigate the significance of false positive cases on the system behavior.

- *Heavy workload (**B3**)*: This workload is somewhere in the middle of the heavy load zone in Figure 6.15. This workload is around the nominal capacity of the original system. It does not cause the system to crash, but definitely leads to decrease its performance.

- *Extreme workload (**B4**)*: This workload is over the saturation point of the original system, in the buckle zone. Protecting the system from crashing under the extreme abnormal situation is one of the most remarkable advantages of adaptive systems.

In order to minimize the experimental errors due to sporadic events, three replications are conducted for each block and each treatment. This is particularly useful due to the fact that user traffic is generated based on a probability distribution and replications can decrease the effect of a specific traffic pattern the results.

Specifically, the following research questions are aimed to be answered in the experiments on $CC2_{original}$ and $CC2_{modified}$:

- **CS3-RQ1**: Does adaptation have a positive impact on CC2 operation? In other words, how do GAAM and Rule-based treatments function in comparison with No-adapt under different workloads?
- **CS3-RQ2**: What is the difference between engineering an adaptation manager by a goal-centric or attribute-action mechanism? In other words, by considering the whole engineering process and the evaluation of mechanism in operation, which approach is better?

Two sets of experiments are conducted to seek the answers of these questions, and investigate the impact of the adaptation mechanisms:

- The first set of experiments, **Exp1**, has a one-factor, three-block design with three replications for $CC2_{original}$ (with no user levels). $Exp1$ is conducted for all the three treatments (No-adapt, GAAM, RuleBased) and three blocks B1, B3, and B4. Details of the workloads for $Exp1$ is in Table 6.13.
- The second set of experiments, **Exp2**, has a one-factor, four-block design with three replications for $CC2_{modified}$ (with user levels). $Exp2$ includes two treatments (No-adapt and GAAM) and four blocks B1 to B4. Rule-based is not selected for this set of experiments because of the tremendous effort it needs to formulate the larger rule-set and to tune them. Details of the workloads for $Exp2$ appears in Table 6.14.

Table 6.13: Inter-arrival time distribution for each service in $Exp1$

	Regular call	***Forwarding***	***VoiceMail***
Light workload (B1)	Exponential distribution (μ=0.5 sec), total calls=3000	Exponential distribution (μ=5 sec), total calls=300	Exponential distribution, (μ=8 sec), total calls=200
Heavy workload (B3)	Exponential distribution (μ=0.3 sec), total calls=3000	Exponential distribution (μ=1 sec), total calls=1000	Exponential distribution, (μ=0.8 sec), total calls=1000
Extreme workload (B4)	Exponential distribution (μ=0.5 sec), total calls=5000	–	–

Table 6.14: Inter-arrival time distribution for each service in $Exp2$

	Regular call	***Forwarding***	***VoiceMail***
Light workload (B1)	Exponential distribution (μ=0.5 sec), total calls=4000	Exponential distribution (μ=5 sec), total calls=400	Exponential distribution, (μ=8 sec), total calls=240
Medium workload (B2)	Exponential distribution (μ=0.15 sec), total calls=4000	Exponential distribution (μ=1 sec), total calls=700	Exponential distribution, (μ=1.6 sec), total calls=400
Heavy workload (B3)	Exponential distribution (μ=0.075 sec), total calls=8000	Exponential distribution (μ=0.5 sec), total calls=1200	Exponential distribution, (μ=1.6 sec), total calls=400
Extreme workload (B4)	Exponential distribution (μ=0.01 sec), total calls=8000	–	–

6.4.5 Testbed

To conduct the load and stress tests, a traffic generator must be involved in the project. For this purpose, an open source SIP scenario generator, SIPp 3.1[11]. SIPp has two main modules, User Agent Client (UAC) and User Agent Server (UAS). UAC and UAS can be viewed as the caller and callee in a typical conversation. Forwarding, Regular, and Voicemail calls are three different UAC behaviors. One running instance of SIPp can generate only one UAC behavior so that multiple SIPp instances are executed to perform all behaviors. To simulate different probabilities of each service being accessed, each of these SIPps will generate traffic based on the exponential distribution with different mean values. For $Exp2$, the portions of bronze, silver, and gold users in any test case will be 50%, 30%, 20%.

Besides using a SIPp to simulate UAC behaviors, we need two more SIPp instances to behave as a UAC registrator and a UAS. The UAC registrator is responsible for keeping CC2 knowing where UAS is (i.e., its IP and port), so that CC2 can correctly redirect the message from UAC to UAS. Only one UAS, which can receive all messages from different UACs, will be run. Figure 6.16 illustrates how the system interacts with SIPps.

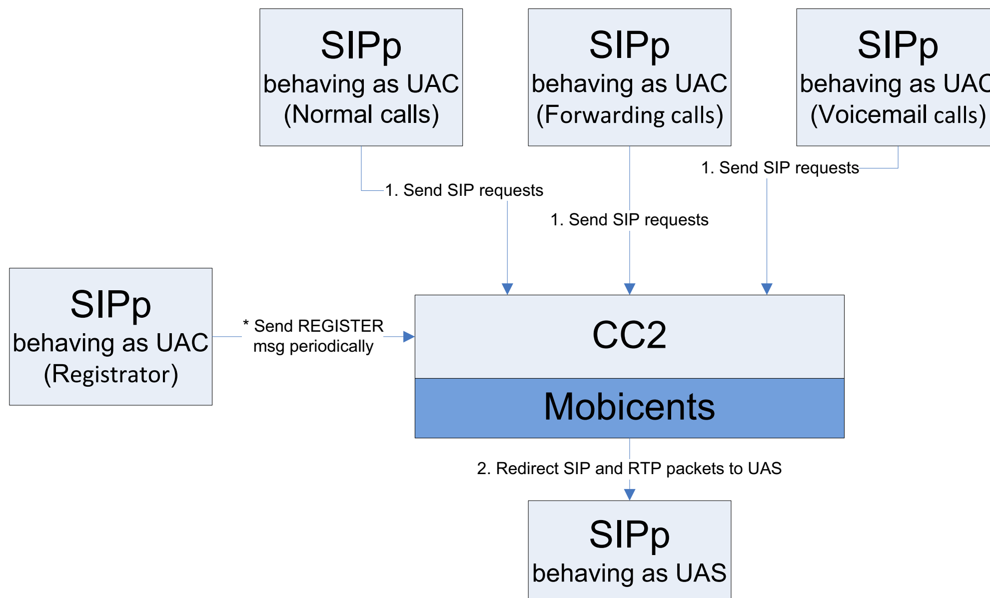

Figure 6.16: Setup environment for evaluating self-adaptive CC2

[11]http://sipp.sourceforge.net/

For $Exp2$, one workstation and one server are used to generate traffic and run the Mobicents server, respectively. The specification of the workstation is Windows XP professional SP3, Intel Pentium 4 CPU 3.4GHz, 2.00GB of RAM. The specification of the server is Windows Server 2003 Standard x64 Edition SP2, Intel Core 2 Quad CPU Q6700 @ 2.66GHz 7.92 GB of RAM. These two machines are connected via 100.0 Mbps Ethernet LAN. For $Exp1$, instead of using a distributed system, both traffic generator and server are running on one workstation. In the experiments, Mobcients server 1.2.0.BETA3 has been used.

6.4.6 Obtained Results

The first step in running ANOVA test is the normality test. QQ-plots of the utility residuals in both experiments, $Exp1$ and $Exp2$, and for each block indicate that utility values are nearly normal. Therefore, the parametric ANOVA is allowed to analyze the data with . The analysis is performed in two intra-block and inter-block ways. In the former, ANOVA and treatment contrasts are accomplished with the regard of each block separately. The latter case investigates treatment effects in the entire experiment with a blocking design. This way eliminates the blocking factor, which is workload.

Intra-block ANOVA

Detailed results of each block is available in Appendix Section A.8. First, treatments are analyzed in each single block. Table 6.15 shows the summary of ANOVA results for each block in two experiments. The p-values indicate that except for the heavy workload ANOVA does not show a significant difference ($\alpha = 5\%$).

Table 6.15: Inter-block ANOVA for CC2 system

	Block		
Experiment	**Light**	**Medium**	**Heavy**
Exp1	0.8034	-	0.0271
Exp2	0.8229	0.981	0.0298

In order to scrutinize the treatment effect, Dunnett test is run for pairwise contrasts. Table 6.18 shows the results. Under the light workload, treatments show no significant difference, but in GAAM-RuleBased contrast skewness toward GAAM is notable. Under the heavy workload, GAAM and RuleBased are significantly different from No-adapt treatment. Again, although the SCL of GAAM-RuleBased contrast skews toward GAAM, their difference is not significant.

Table 6.16: Dunnett test for pairwise comparison of treatments (*** means significantly different)

Experiment	Block	Treatment Contrast	Simultaneous 95% Confidence Limits	Significant ($\alpha = .05\%$)
Exp1	**Light**	GAAM-No-adapt	(-2524.7 , 3054)	
		RuleBased-No-adapt	(-3177.4 , 2401.4)	
		GAAM-RuleBased	(-2136.7 , 3442)	
	Heavy	GAAM-No-adapt	(2192 , 22542)	***
		RuleBased-No-adapt	(222 , 20572)	***
		GAAM-RuleBased	(-8204 , 12146)	
Exp2	**Light**	GAAM-No-adapt	(-12998 , 15447)	
	Medium	GAAM-No-adapt	(-6795 , 866)	
	Heavy	GAAM-No-adapt	(8494 , 97955)	***

The result under a medium workload (B2) shows an interesting point. There is no significant difference between GAAM and No-adapt, but there is a skew toward No-adapt. By looking at Table A.4, we can see that the performance of Regular calls for both systems have no major difference. However, more Forwarding and Voicemail requests are served in No-adapt, because the adaptation goals includes a proactive adaptation behavior. In fact, the activation criteria activates the goals before they are denied. This is a design decision to prevent bouncing between satisfaction and denial. Therefore, two of the criteria to activate a goal and to deactivate it, are not the same. In this way, the adaptation manager realizes that the workload may be over the original capacity gradually due to its trend, and it blocks some Forwarding and Voicemail requests from low-level users. This can be considered as a false positive case of the adaptive system. However, this effect is tunable and by adjusting the criteria statically or even dynamically, it can be improved.

Inter-block ANOVA

ANOVA table results in Table 6.17 show a significant difference between treatments in both experiments. This is observable from both the entire model and the treatments in Type III SS. The other outcomes of ANOVA table, not shown in Table 6.17, show that the block effect is significant (H_0 is rejected).

Tukey t-test shows that in inter-block analysis of $Exp1$ Rule-based and GAAM treatments are in a distinct group from the No-adapt treatment. It also shows that B1 and B3 (light and heavy workloads) are in two separate groups. For $Exp2$, Tukey t-test indicates that GAAM and No-adapt are in different groups, but in blocks. Light and medium blocks belong to the same group. This is because these two workloads belong to the same zone, as described before in the experiment design.

Table 6.17: ANOVA test for utility values based on GLM F-test ($\alpha = .05\%$)

	Entire model		**Treatment effect (on Type III SS)**	
	F value	$Pr > F$	F value	$Pr > F$
Exp1	57.62	<.0001	6.54	0.0120
Exp2	37.31	<.0001	9.21	0.0104

Pairwise contrasts with No-adapt treatment (i.e., control treatment) based on the Dunnett test show significant differences between adaptation and No-adapt treatments. An interesting point is that although the Dunnett test indicates a significant difference between Rule-based and No-adapt with confidence level 95%, this is not the case with confidence level 99%. This indicates GAAM behaves slightly better than Rule-based, even though the contrast does not show this significantly with 99% confidence level. This is also observable in SCL of Table 6.18. In $Exp1$, the SCL limits are skewed slightly toward GAAM treatment, although the test does not show a significant difference with confidence level 95%.

Table 6.18: Dunnett t-test for pairwise comparison of treatments (*** means significantly different)

Experiment	**Treatment Contrast**	**Simultaneous 95% Confidence Limits**	**Significant ($\alpha = .05\%$)**
Exp1	GAAM-No-adapt	(1705 , 10927)	***
	RuleBased-No-adapt	(393 , 9615)	***
	GAAM-Rule-based	(-3299 , 5923)	
Exp2	GAAM-No-adapt	(4842 , 29481)	***

Table 6.19 shows the crash data for $CC2_{original}$ and $CC2_{modified}$ in two experiments. Both Rule-based and GAAM successfully change the application to survive this load, while No-adapt crashes in both cases. Both Rule-based and GAAM can successfully detect these threats and protect themselves from crashing. The system with No-adapt crashes at the beginning until the JBoss application server restarts CC2. In GAAM, the "prevent system crash" is activated immediately when request flooding is detected. The only preferred action of this goal is blocking all requests from all users. The goal interrupts the service for a short time (e.g., T_{ad}), but prevents the system crash, which takes much more time to recover from. Correspondingly, Rule-based has a policy that blocks all users when a similar situation occurs.

Table 6.19: Experiments for extreme heavy workload (B4)

Experiment	Treatment	Crashed after #users
Exp1	No-adapt	135
	GAAM	N/A
	Rule-based	N/A
Exp2	No-adapt	58
	GAAM	N/A

6.4.7 Lessons Learned

In both Exp1 an Exp2 we can see that the type of workload plays an important role in evaluating the effect of adaptation treatments. The blocking design and replications set out to eliminate the workload factor in the analysis.

The findings lead to the following answers to the defined research questions:

- *CS3-RQ1* compares adaptation treatments, goal-ensemble, and rule-based mechanism with the no-adapt treatment. Inter- and intra-block analysis shows that adaptation treatments outperform or at least perform as effective as the No-adapt treatment. The effect of adaptation treatments is particularly notable in inter-block analysis when the blocking factor is eliminated. Therefore, statistically adaptation has a positive impact on system behavior regarding its designated quality goals and SLA. The results under extreme heavy workload (B4) also verify this statement in the conducted experiments.

- *CS3-RQ2* contrasts GAAM and Rule-based as two examples of realizing goal-centric and attribute-action coupling patterns and mechanisms. The results related to this question are only available for Exp1. The inter- and intra-block analysis show no significant difference between these two treatments, even though there is a skewness in the calculated SCL toward GAAM. The difference is not observed even with 99% confidence level. However, a significant difference has been observed in developing and tuning these mechanisms. GAAM is easier to define and tune in comparison to Rule-based mechanism. This is true especially when the number of attributes and actions increases. This was the main reason implementing Rule-based for Exp2 was complex and too time-consuming. Moreover, the maintenance and evolution of rules in the Rule-based treatment are not easy and straightforward tasks. Evolving GAAM from Exp1 to Exp2 was not a tough job in comparison with the same task for rule-based mechanism. Plenty of time was spent on the implementing the rules, but the estimated effort to tune the rule set was too high that the task was skipped in this thesis. Thus, in short, regarding the ANOVA results, we cannot say GAAM

is more effective than Rule-based, but the effort of developing, tuning and evolving of GAAM could be higher, especially in larger systems.

A threat to the validity of these findings is that the results depend on the evaluation function E, SLA, and the set of defined quality goals. The experiments do not explicitly rule out this threat, but two reasons can help us in arguing the insignificance of this issue. First, individual attributes of the system also does not show a poor performance of the adaptation mechanisms, even though weights of goals may change the outcomes in some special cases. Second, the other case studies in this chapter also endorse the positive impact of these adaptation treatments. Therefore, the possibility of significant effect of the evaluation function, as internal and external threats to validity, is not remarkable.

6.5 Summary

The goal of this chapter was to put the proposed quality-driven framework QFeam into action. The main objective was to investigate several adaptation models and mechanisms in engineering an adaptation manager. Two patterns, goal-centric and attribute-action-coupling, and two mechanisms, the goal-ensemble and rule-based mechanisms, were particularly evaluated in the conducted experiments. The initial challenge in these experiments was the lack of a commonly used testbed or benchmark. A considerable amount of time was spent on selecting case studies, simulating or re-engineering adaptable software, and preparing the platform to apply the adaptation. This part of the work has been accomplished collaboratively with the other students in the STAR lab.

Several research questions that were investigated in the experiments result in the following findings:

- The *first case study* indicated that goals can play an effective role in the engineering an adaptation manager with an attribute-action-coupling pattern. The results also show that fine-grained actions can be quite helpful in adapting a mission-critical application with low cost.
- The *second case study* mainly addressed employing the goal-ensemble and rule-based mechanisms in a typical news web application. The application-level adaptations were utilized and the results were compared for the two approaches. Although, the outcome did not indicate a significant difference between these two, a remarkably lesser effort is used in developing the system using the goal-ensemble mechanism. Moreover, modeling preferences using ordinal and cardinal utilities were investigated. Interestingly, ordinal preferences showed promising results.

- The *third case study*, again put these two mechanisms into practice, but this time in a service-oriented application, VoIP CC2 call controller. In this set of experiments, the goal-ensemble mechanism was employed in the presence of user levels. In this case study, again GAAM showed promising results.

The valuable experiences in these case studies helped improving QFeam, and evaluating the goal-ensemble mechanism in the domain of mission-critical systems. Some noteworthy points that can be considered for future experiments are as follows:

- The built goal hierarchies in the experiments included only global goals. These can be refined down to local goals for each subsystem or component. However, in order to decrease the level of complexity and effort for tuning the adaptation manager, it seems using only global goals for the deciding process is more appropriate. Therefore, for example in the goal-ensemble, a cluster of goals triggers their representative (i.e., the root of the sub-tree) to vote for the actions.

- For service-oriented applications, other fine-grained adaptation actions can be also employed. For instance, switching between alternative services with the similar functionality and different qualities is an option. This is particularly possible when a system uses an external service (e.g., a web service).

- These experiments focused only on fine-grained adaptation actions. Coarse-grained actions, such as architecture-based changes, need to be investigated in empirical studies as well. Since these actions may have severe impacts on the adaptable application, it seems using behavioral or architectural knowledge of the system is quite helpful in decision-making. In fact, this is the fourth space discussed in the adaptation conceptual model.

- These experiments included only performance and availability goals. Although there is no assumption in QFeam to cover just these quality factors, the other factors have not been investigated yet. The main strategy to evaluate built self-adaptive application was to put them under stress, load and different workloads. Failure, security attack, and resource shortage are among other conditions that can be happened for real systems.

Chapter 7

Conclusions and Future Work

"Vision without action is a daydream.
Action without vision is a nightmare."
Japanese Proverb

7.1 Thesis Summary

Engineering self-adaptive software still lacks a well-established and systematic process. There are many challenges in designing, developing, and testing such systems. This thesis highlights the adaptation manager that adjusts software at run-time in different situations. In fact, an adaptation manager continuously traces the adaptation requirements by checking the six requirements questions. Therefore, the objective of the adaptation manager is to find out when, where, what, why, and how adaptation needs to be applied, in addition to who can be responsible for it. To achieve this objective, an adaptation manager includes four processes: monitoring, detecting, deciding, and acting. Among these processes, this thesis focuses on the deciding process due to its key role in the operation of the adaptation manager. However, with respect to the strong link between the detecting and deciding processes, the former is also addressed, yet not extensively.

To address the problem under study, an engineering framework for the adaptation manager, called QFeam, has been proposed. Since adaptation requirements and self- * properties are strongly linked to quality requirements, QFeam relies on quality requirements. This framework is inspired by the ideas from the three research areas: i) requirements engineering, particularly goal-driven approaches, ii) behavior-based robotics, and iii) mechanism design and game theory, especially cooperative and collective decision-making. The two processes of building run-time adaptation model and adaptation mechanism design are emphasized in QFeam.

- *Building run-time adaptation model*: Three tenets of a typical adaptation model are quality goals, adaptation actions, and domain attributes. These three parts capture the essential concepts in the adaptation problem space. QFeam discusses the ways that entities in each of the three conceptual spaces can be elicited and modeled. Three metamodels are presented for this purpose. Then, in order to build an adaptation model, the specified entities or models in the three spaces should be connected together. Three composition patterns are discussed for this purpose: *goal-centric*, *attribute-action-coupling*, and *hybrid*. These patterns cover the approaches researchers and practitioners employ commonly in this community for engineering the adaptation manager. The notable point is that the emphasis here is on "alive" models at run-time, not models used at the development time.

- *Adaptation mechanism design*: The design of a run-time adaptation mechanism is strongly related to the adopted composition pattern. Thus, we discussed three categories of mechanisms according to the pattern used in the adaptation model. A novel mechanism was introduced in this thesis is the *goal-ensemble*, which is based on the idea of having an ensemble of goals to collectively decide about the adaptation. The collective decision-making has already been tried in some mobile robots. However, those experiences focus only on limited actions of robot movement and moreover, they use behaviors to suggest actions. The goal-ensemble benefits from the power of goal-driven requirements models in addition of the successful experiences of run-time control in robotics. A concrete model and mechanism, called GAAM, has been designed based on the goal-ensemble. GAAM employs a weighted voting game to fuse the actions suggested by each goal.

Three case studies are selected for the empirical studies: *TPC-W bookstore application*, a simulated *news web application*, and *CC2 VoIP call controller*. The main objective was to experience the problems in engineering self-adaptive software, particularly in the context of mission-critical systems. The focused quality goals in the experiments are performance and availability. These two goals have been selected due to the fact that most of SLAs emphasize these two quality factors. Moreover, the personal a priori knowledge of performance engineering was a prominent reason in selecting these two factors.

Although designing and developing the adaptation manager are targeted, due to the lack of benchmarks and evaluation platforms, the entire job of building the adaptable software has been accomplished during this research. This is a very time-consuming task regarding the variety of technologies used in the case studies. As a result, one of the side benefits of this research was gaining experience of working with dynamic adaptation techniques and monitoring in JBoss application server and Mobicents besides of gaining valuable experiences in the Matlab/Simulink environment.

The findings show that the goal-centric pattern and the corresponding mechanisms, such as the goal-ensemble, can be as effective as the common attribute-action-coupling approach. In particular, GAAM outperforms the rule-based mechanism in some cases, and by considering the less effort in developing GAAM it seems an appropriate option in this context. The validation is performed for mission-critical systems, and by considering performance and availability quality factors. However, as discussed before, no assumption has been made in the model and mechanism related to these aspects. Therefore, there would not be significant obstacles in extending the model to other quality factors, at least theoretically. Moreover, the following conclusions can be drawn from the empirical studies:

- Goals can play a prominent role in engineering the adaptation manager. This could be either at the development phase, like in TPC-W case study, or at development and run-time, like in the news application and CC2.
- The Goal-ensemble mechanism represents explicit adaptation goals in an embedded run-time model. This issue helps developers to tune and maintain the adaptation manager easier than a rule-based approach with implicit goals.
- Although preference matrices can be accurately specified using cardinal utilities, ordinal utilities can be as well as cardinal ones with less effort in eliciting and tuning.
- Application-level and fine-grained adaptation actions are normally easier to implement and can be quite effective. With respect to the engineering effort, the complexity and the risk of taking adaptation actions, in some cases fine-grained actions may be preferred to coarse-grained actions.

This research is one step towards establishing an engineering framework for self-adaptive software. Some assumptions and limitations are taken into account to narrow the scope of this thesis. The following points are notable:

- GAAM does not consider the history of previously taken actions and their observed aftereffects explicitly. Although the impacts of actions on the domain attributes are observed by the adaptation manager, there is no method to use these impacts for future rounds of adaptation. This is because GAAM does not update its knowledge at run-time. Employing a learning method could be beneficial in addressing this issue.
- Although an auxiliary concept space is denoted in the adaptation model in Chapter 3, it is not explicitly discussed in the thesis. More work is required to investigate how models of architecture can be bound to the other three spaces and be utilized by the adaptation mechanism.

- QFeam does not consider details of the monitoring and acting processes. However, in some situations these processes may affect on the deciding and detecting processes. For example, in the monitoring process noise and sensor failures can generate some erroneous attributes. On the other hand, in the acting process the failure in executing an action, particularly a composite action, or the possibility of executing two actions to achieve two separate goals, are cases that are not evaluated in this thesis.

7.2 Future Work

Self-adaptive software has a long way to go be mature and trustable, and many challenges are in front of the theoretical and practical aspects of the design and development of these systems [186]. This thesis presents several patterns and design ideas that can be realized by different solutions. Some of these solutions are discussed and a few of them are practiced in the case studies. Several potential works in continuation of this research are as follows:

- The hybrid pattern needs to be evaluated in a series of empirical studies. As mentioned before, the presented goal-centric pattern can be extended by adding another level of action fusion module to realize the hybrid pattern.
- As described in the possible solutions for realizing the goal-centric pattern, the ensemble idea can be employed for several deciding processes, too. In other disciplines, such as data clustering, this idea has also been used for aggregating several clustering schemems or for aggregating several controllers.
- The empirical studies in this thesis focus on the performance and availability quality factors. An interesting research direction is to apply the discussed mechanisms, especially the goal-ensemble mechanism, to other cases including security and reliability goals.
- Instrumenting sensors and effectors for building the adaptable software in the case studies, was a valuable experience. However, this area needs a systematic process and more enabling technologies. This is especially needed for effecting mechanisms. The fact is that sensing and monitoring have been improved much more than the effecting and dynamic change of software artifacts. The cost and the probability of failure are much higher for dynamic adaptation actions than sensing. Therefore, this specific area requires tremendous effort, particularly for application-level adaptation.
- Testing and quality assurance of self-adaptive software systems is still premature. The discussed evaluation approach based on the adaptation goals and SLA is helpful, but not enough. Defining test cases with appropriate coverage and automating the entire process require extensive research efforts.

The current thesis paves the way toward providing an engineering framework, but many concerns remain and should be investigated in theory and practice for this purpose.

APPENDICES

Appendix A

CC2 case study details

A.1 Service Level Agreement (SLA)

- Each Bronze, Silver or Gold user, who conducts a successful regular call, needs to pay 20, 40 or 60 cents for the call.
- Each Bronze, Silver or Gold user, who suffers a response time of regular call longer than 6 sec, will be paid 7, 18, or 35 cents by the VoIP service provider (SP).
- Each Bronze, Silver or Gold user, who conducts a successful forwarding call, will produce 5, 10 or 15 cents expected profit for the SP.
- Each Bronze, Silver or Gold user, who suffers a response time of forwarding call longer than 0.1 sec, will be paid 2, 4, or 6 cents by the VoIP service provider.
- Each Bronze, Silver or Gold user, who conducts a successful voicemail call, will produce 4, 8 or 12 cents expected profit for the SP.
- Each Bronze, Silver or Gold user, who suffers a response time of voicemail call longer than 1.5 sec, will be paid 2, 4, or 6 cents by the VoIP service provider.
- Each Bronze, Silver or Gold user, who receives a timeout from voicemail service, will be paid 1, 2, or 3 cents by the VoIP service provider.

A.2 Attributes for $CC2_{original}$

- Regular calls

- Average Response Time (spent in the server, not including network delay and client processing time) – at_1 (RT^{Reg})
  - Throughput – at_2 (Th^{Reg})
  - Arrival Rate – at_3 (AR^{Reg})

- Forwarding service
  - Average Response Time – at_4 (RT^{FW})
- Voicemail Service
  - Average Response Time – at_5 (RT^{VM})
  - Throughput of each user level – at_6 (Th^{VM})
  - Arrival Rate of each user level – at_7 (RT^{AR})

Window size for moving average of response time and throughput

- 200 for Response time of Regular call
- 20 for Response time of Forwarding
- 50 for Response time of VM
- 20 for Throughput of Regular call
- 5 for Throughput of VM

A.3 Attributes for $CC2_{modified}$

- Regular VoIP calls
  - Average Response Time (RT) of each user level (spent in the server, not including network delay and client processing time) – at_1 to at_3 ($RT^{Reg}_{Gold}, RT^{Reg}_{Silver}, RT^{Reg}_{Bronze}$)
  - Throughput (TH) of each user level – at_4 to at_6 ($Th^{Reg}_{Gold}, Th^{Reg}_{Silver}, Th^{Reg}_{Bronze}$)
  - Arrival Rate (AR) of each user level – at_7 to at_9 ($AR^{Reg}_{Gold}, AR^{Reg}_{Silver}, AR^{Reg}_{Bronze}$)
- Forwarding service
  - Average Response Time of each user level – at_{10} to at_{12} ($RT^{Fw}_{Gold}, RT^{Fw}_{Silver}, RT^{Fw}_{Bronze}$)

- Voicemail Service
  - Average Response Time of each user level – at_{13} to at_{15} ($RT^{VM}_{Gold}, RT^{VM}_{Silver}, RT^{VM}_{Bronze}$)
  - Throughput of each user level – at_{16} to at_{18} ($Th^{VM}_{Gold}, Th^{VM}_{Silver}, Th^{VM}_{Bronze}$)
  - Arrival Rate of each user level – at_{19} to at_{21} ($AR^{VM}_{Gold}, AR^{VM}_{Silver}, AR^{VM}_{Bronze}$)

A.4 Actions for $CC2_{original}$

- Block any services requests from users – ac_1
- Disable Forwarding – ac_2
- Shorten the max recording time of users – ac_3
- Disable Voicemail – ac_4
- Unblock A1 – ac_5
- Enable Forwarding – ac_6
- Lengthen the max recording time of users – ac_7
- Enable Voicemail – ac_8

A.5 Goals for $CC2_{original}$

- Prevent system crash g_1
- Achieve acceptable reg. call TP – g_2
- Achieve acceptable VM TP – g_3
- Achieve acceptable reg. call RT – g_4
- Achieve acceptable FW RT – g_5
- Achieve acceptable VM RT – g_6
- Achieve suitable length of VM msg – g_7
- Achieve reg call availability – g_8
- Achieve FW availability – g_9
- Achieve VM availability – g_{10}

A.6 Actions for $CC2_{modified}$

- Blocking
  - Completely block any service requests from bronze users – ac_1
  - Completely block any service requests from silver users – ac_2
  - Completely block any service requests from gold users – ac_3
  - ac_1 counter-action – ac_4
  - ac_2 counter-action – ac_5
  - ac_3 counter-action – ac_6
- Forwarding
  - Not allow bronze users' calls to be forwarded – ac_7
  - Not allow silver users' calls to be forwarded – ac_8
  - Not allow gold users' calls to be forwarded – ac_9
  - ac_7 counter-action – ac_{10}
  - ac_8 counter-action – ac_{11}
  - ac_9 counter-action – ac_{12}
- VoiceMail
  - Change the max recording time of bronze users – ac_{13}
  - Change the max recording time of silver users – ac_{14}
  - Change the max recording time of gold users – ac_{15}
  - Not allow bronze users to access voicemail service – ac_{16}
  - Not allow silver users to access voicemail service – ac_{17}
  - Not allow gold users to access voicemail service – ac_{18}
  - ac_{13} counter-action – ac_{20}
  - ac_{14} counter-action – ac_{21}
  - ac_{15} counter-action – ac_{22}
  - ac_{16} counter-action – ac_{23}
  - ac_{17} counter-action – ac_{24}
  - ac_{18} counter-action – ac_{25}

A.7 Goals, Weights and Preferences for $CC2_{modified}$

- General
  - g_1-Prevent system crash – $ac_1 \succ ac_2 \succ ac_3$ ($p_1 = 300$)
- Gold
  - g_2-Achieve acceptable reg. call TP – $ac_16 \succ ac_7 \succ ac_13 \succ ac_17 \succ ac_8 \succ ac_14 \succ ac_18 \succ ac_9 \succ ac_15$ ($p_2 = 18$)
  - g_3-Achieve acceptable VM TP – $ac_13 \succ ac_7 \succ ac_14 \succ ac_8 \succ ac_15 \succ ac_9$ ($p_3 = 16$)
  - g_4-Achieve acceptable reg. call RT – $ac_16 \succ ac_7 \succ ac_13 \succ ac_17 \succ ac_8 \succ ac_14 \succ ac_18 \succ ac_9 \succ ac_15 \succ A3$ ($p_4 = 32$)
  - g_5-Achieve acceptable FW RT – $ac_16 \succ ac_13 \succ ac_7 \succ ac_17 \succ ac_14 \succ ac_8 \succ ac_18 \succ ac_15 \succ ac_9$ ($p_5 = 14$)
  - g_6-Achieve acceptable VM RT – $ac_13 \succ ac_7 \succ ac_16 \succ ac_14 \succ ac_8 \succ ac_17 \succ ac_15 \succ ac_9 \succ ac_18$ ($p_6 = 18$)
  - g_7-Achieve suitable length of VM msg – ac_22 ($p_7 = 10$)
  - g_8-Achieve reg call availability – ac_6 ($p_8 = 54$)
  - g_9-Achieve FW availability – ac_12 ($p_9 = 14$)
  - g_{10}-Achieve VM availability – ac_25 ($p_{10} = 20$)
- Silver
  - g_{11}-Achieve acceptable reg. call TP – $ac_16 \succ ac_7 \succ ac_13 \succ ac_17 \succ ac_8 \succ ac_14$ ($p_{11} = 14$)
  - g_{12}-Achieve acceptable VM TP – $ac_13 \succ ac_7 \succ ac_14 \succ ac_8$ ($p_{12} = 12$)
  - g_{13}-g_{10}-Achieve acceptable reg. call RT – $ac_16 \succ ac_7 \succ ac_13 \succ ac_17 \succ ac_8 \succ ac_14$ ($p_{13} = 24$)
  - g_{14}-Achieve acceptable FW RT – $ac_16 \succ ac_13 \succ ac_7 \succ ac_17 \succ ac_14 \succ ac_8$ ($p_{14} = 11$)
  - g_{15}-Achieve acceptable VM RT – $ac_13 \succ ac_7 \succ ac_16 \succ ac_14 \succ ac_8 \succ ac_17$ ($p_{15} = 14$)
  - g_{16}-Achieve suitable length of VM msg – ac_21 ($p_{16} = 8$)
  - g_{17}-Achieve reg call availability – ac_5 ($p_{17} = 41$)
  - g_{18}-Achieve FW availability – ac_11 ($p_{18} = 11$)

- g_{19}-Achieve VM availability – ac_24 ($p_{19} = 15$)

- Bronze
  - g_{20}-Achieve acceptable reg. call TP – $ac_16 \succ ac_7 \succ ac_13$ ($p_{20} = 9$)
  - g_{21}-Achieve acceptable VM TP – $ac_13 \succ ac_7$ ($p_{21} = 8$)
  - g_{22}-Achieve acceptable reg. call RT – $ac_16 \succ ac_7 \succ ac_13$ ($p_{22} = 16$)
  - g_{23}-Achieve acceptable FW RT – $ac_16 \succ ac_13 \succ ac_7$ ($p_{23} = 7$)
  - g_{24}-Achieve acceptable VM RT – $ac_13 \succ ac_7 \succ ac_16$ ($p_{24} = 9$)
  - g_{25}-Achieve suitable length of VM msg – ac_20 ($p_{25} = 5$)
  - g_{26}-Achieve reg call availability – ac_4 ($p_{26} = 27$)
  - g_{27}-Achieve FW availability – ac_10 ($p_{27} = 7$)
  - g_{28}-Achieve VM availability – ac_23 ($p_{28} = 10$)

A.8 CC2 experiments detail results

Table A.1: CC2 Exp1 B1 results (light workload)

	Served Regular Call	Avg RT Regular Call	Served Forwarding	Served Voicemail	Utility Value	Achieved Utility (%)
Rule-Based	2859	2.331	252	132	58326	93.62
	2859	2.616	253	138	58273	93.54
	2872	2.765	224	151	57721	92.65
Average	2863	2.571	243	140	58107	93.27
GAAM	2924	2.831	264	165	59569	95.62
	2844	3.402	245	156	57008	91.51
	2940	2.844	232	145	59701	95.83
Average	2903	3.026	247	155	58759	94.32
NoAdapt	2834	3.352	228	155	56922	91.37
	2922	2.822	226	162	59274	95.14
	2926	2.570	170	141	59288	95.17
Average	2894	2.915	208	153	58495	93.89

Table A.2: CC2 Exp1 B3 results (heavy workload)

	Served Regular Call	Avg RT Regular Call	Served Forwarding	Served Voicemail	Utility Value	Achieved Utility (%)
Rule-Based	2099	7.34	372	185	39058	56.60
	2073	7.34	392	339	40312	58.42
	1654	10.33	346	227	31353	45.44
Average	1942	8.34	370	250	36908	53.49
GAAM	2275	5.13	211	201	44460	64.43
	1733	7.56	225	354	33621	48.73
	2011	6.83	255	283	38554	55.88
Average	2006	6.51	230	279	38878	56.35
NoAdapt	1285	11.70	426	380	24555	35.59
	1470	11.23	400	403	26493	38.40
	1672	15.00	559	450	28485	41.28
Average	1476	12.64	462	411	26511	38.42

Table A.3: CC2 Exp2 B1 results (light workload)

	Served Regular Call	Avg RT Regular Call	Served Forwarding	Served Voicemail	Utility Value	Achieved Utility (%)
NoAdapt	3646	4.84	325	207	122074	86.56
	3782	4.36	338	195	119799	84.94
	3619	4.70	333	196	127111	90.13
Average	3682	4.63	332	199	122995	87.21
GAAM	3629	4.61	315	203	115272	81.73
	3735	4.33	310	196	130859	92.79
	3605	4.08	321	194	126526	89.71
Average	3656	4.34	315	198	124219	88.08

Table A.4: CC2 Exp2 B2 results (medium workload)

	Served Regular Call	Avg RT Regular Call	Served Forwarding	Served Voicemail	Utility Value	Achieved Utility (%)
NoAdapt	3701	4.74	458	250	131291	90.75
	3745	4.24	433	194	132229	91.40
	3766	4.44	446	265	133508	92.28
Average	3737	4.47	446	237	132343	91.48
GAAM	3645	5.17	231	165	126937	87.74
	3777	4.03	141	151	130638	90.30
	3781	4.20	143	129	130560	90.25
Average	3734	4.47	172	148	129378	89.43

Table A.5: CC2 Exp2 B3 results (heavy workload)

	Served Regular Call	Avg RT Regular Call	Served Forwarding	Served Voicemail	Utility Value	Achieved Utility (%)
NoAdapt	7072	7.443	553	165	192259	67.48
	6546	8.666	550	165	152505	53.53
	6734	7.974	569	142	181667	63.76
Average	6784	8.028	557	157	175477	61.59
GAAM	7290	5.003	195	96	249649	87.62
	7180	6.291	223	94	213143	74.81
	7046	6.205	210	95	223310	78.38
Average	7172	5.833	209	95	228701	80.27

References

[1] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. A control-based framework for self-managing distributed computing systems. In *Proc. of Workshop on Self-healing Systems*, pages 3–7, 2004. 29

[2] Hasina Abdu, Hanan Lutfiyya, and Michael A. Bauer. A model for efficient configuration of management agents in distributed systems. *Performance Evaluation*, 54(4):285–309, 2003. 31

[3] M. Adams, D. Edmond, and A. ter Hofstede. The application of activity theory to dynamic workflow adaptation issues. In *Proc. of Pacific Asia Conf. on Information Systems*, pages 1836–1852, 2003. 105

[4] Y. Al-Nashif, A.A. Kumar, S. Hariri, Y. Luo, F. Szidarovsky, and G. Qu. Multi-Level Intrusion Detection System (ML-IDS). In *Proc. of Int. Conf. on Autonomic Computing*, pages 131–140, 2008. 33

[5] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice-Hall, 2001. 15, 16

[6] Wael Hosny Fouad Aly and Hanan Lutfiyya. Dynamic adaptation of policies in data center management. In *Proc. of IEEE Int. Workshop on Policies for Distributed Systems & Networks*, pages 266–272, 2007. 30

[7] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. In *Proc. of Int. Conf. on Autonomic and Autonomous Systems*, pages 175–181, 2008. 28, 85, 98, 124

[8] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA-based management of a computing utility. In *Proc. of IFIP/IEEE Int. Symp. on Integrated Network Management*, pages 855–868, 2001. 19, 33

[9] Ronald C. Arkin. *Behavior-based Robotics*. MIT Press, 1998. 51

[10] Application response measurement. http://www.opengroup.org/tech/management/arm/. 15, 16

[11] N. Arshad, D. Heimbigner, and A.L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In *Proc. of IEEE Conf. on Tools with Artificial Intelligence*, pages 39–46, 2003. 27

[12] Reza Asadollahi. Starmx: A framework for developing self-managing software systems. Master's thesis, University of Waterloo, September 2009. 98, 102

[13] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. StarMX: A framework for developing self-managing Java-based systems. In *Proc. of ICSE Workshop on Software Eng. for Adaptive and Self-Managing Systems*, pages 58–67, 2009. 50, 98, 99, 100, 129

[14] H. G. Ayad and M. S. Kamel. Cumulative voting consensus method for partitions with variable number of clusters. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 30(1):160–173, 2008. 84

[15] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni A. Di Caro, Frederick Ducatelle, Luca M. Gambardella, Niloy Ganguly, Mark Jelasity, Roberto Montemanni, Alberto Montresor, and Tore Urnes. Design patterns from biology for distributed computing. *ACM Trans. on Autonomous and Adaptive Systems*, 1(1):26–66, 2006. 16

[16] Ozalp Babaoglu, Mark Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen. *Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations*. 2005. 7, 9

[17] N. Badr, A. Taleb-Bendiab, and D. Reilly. Policy-based autonomic control service. In *Proc. of IEEE Int. Workshop on Policies for Distributed Systems & Networks*, page 99, 2004. 30

[18] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004. 114

[19] David F. Bantz, Chatschik Bisdikian, David Challener, John P. Karidis, Steve Mastrianni, Ajay Mohindra, Dennis G. Shea, and Michael Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176, 2003. 31

[20] V. Bhat, M. Parashar, Hua Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *Proc. of IEEE Int. Conf. on Autonomic Computing*, pages 15–24, 2006. 29

[21] Stefan Bieniawski and David Wolpert. Adaptive, distributed control of constrained multi-agent systems. In *Proc. of Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 1230–1231, 2004. 27

[22] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002. 21, 108, 135

[23] Ken Birman, Robbert van Renesse, and Werner Vogels. Adding high availability and autonomic behavior to web services. In *Proc. of Int. Conf. on Software Eng.*, pages 17–26, 2004. 26

[24] J. David Blaine and Jane Cleland-Huang. Software quality requirements: How to balance competing priorities. *IEEE Software*, 25(2):22–24, 2008. 24, 64

[25] B. Boehm and H. In. Identifying quality-requirement conflicts. *IEEE Software*, 13:25–35, 1996. 89

[26] C. Boutilier, R. Das, J. O. Kephart, and William E. Walsh. Towards cooperative negotiation for decentralized resource allocation in autonomic computing systems. In *Proc. of Int. Joint Conf. on Artificial Intelligence*, pages 1458–1459, 2003. 28

[27] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proc. of ACM Workshop on Self-managed systems*, pages 28–33, 2004. 26

[28] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004. 54

[29] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986. 51, 85

[30] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005. 8

[31] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *J. Software Maintenance and Evolution*, 17(5):309–332, 2005. 62

[32] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns - pattern oriented software architecture*. Wiley, 1996. 15, 16

[33] George Candea, James Cutler, and Armando Fox. Improving availability with recursive microreboots: a soft-state system case study. *Performance Evaluation*, 56(1-4):213–248, 2004. 33, 115, 117

[34] George Candea and Armando Fox. Crash-only software. In *Proc. of Hot Topics in Operating Systems Workshop*, pages 67–72, 2003. 51

[35] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(1):175–190, 2006. 19, 33

[36] V. Cardellini, M. Colajanni, and PS Yu. Dynamic load balancing on Web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999. 19

[37] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, 2001. 15, 16

[38] L. Chen and P. Pu. Survey of preference elicitation methods. Technical report, Swiss Federal Institute of Technology in Lausanne, Technical Report No. IC/200467, 2004. 89

[39] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *J. Syst. Software*, 74(1):35–43, 2005. 114

[40] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. pages 1–26, 2009. 29

[41] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proc. of Workshop on Self-adaptation and Self-managing Systems*, pages 2–8, 2006. 1, 19, 74

[42] S.W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *Proc. of ICSE Workshop on Software Eng. for Adaptive and Self-Managing Systems*, pages 132–141, 2009. 94

[43] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 2000. 58, 59

[44] Common information model standard. http://www.dmtf.org/standards/cim/. 15, 16

[45] I.J. Curiel. *Cooperative Game Theory and Applications: Cooperative Games Arising from Combinatorial Optimization Problems*. Kluwer Academic Publishers, 1997. 52

[46] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993. 54, 57

[47] D. Dawson, R. Desmarais, H. M. Kienle, and H. A. M
"uller. Monitoring in adaptive systems using reflection. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 81–88. ACM New York, NY, USA, 2008. 16

[48] Rogerio de Lemos and Jose Luiz Fiadeiro. An architectural support for self-adaptive software for treating faults. In *Proc. of Workshop on Self-healing Systems*, pages 39–42, 2002. 10

[49] M. Dilman and D. Raz. Efficient reactive monitoring. *IEEE Journal on Selected Areas in Communications*, 20(4):668–676, 2002. 31

[50] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006. 13, 23

[51] Jim Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, Department of Computer Science, Trinity College Dublin, 2004. 22, 28, 33

[52] Jim Dowling and Vinny Cahill. Self-managed decentralised systems using K-components and collaborative reinforcement learning. In *Proc. of ACM Workshop on Self-Managed Systems*, pages 39–43, 2004. 19, 33

[53] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In *Proc. of Int. Conf. on Aspect-Oriented Software Development*, pages 66–73, 2004. 73

[54] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, pages 62–70, 2006. 20, 21, 33

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. 15, 16, 100, 101

[56] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal, Special Issues on Autonomic Computing*, 42:5–18, 2003. 11

[57] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004. 26, 33, 73

[58] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proc. of Workshop on Self-healing Systems*, pages 27–32, 2002. 22, 33

[59] Erann Gat, R. Peter Bonnasso, and Robin Murphy. On three-layer architectures. In *Proc. of Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997. 51, 85

[60] John C. Georgas, André van der Hoek, and Richard N. Taylor. Architectural runtime configuration management in support of dependable self-adaptive software. In *Proc. of Workshop on Architecting Dependable Systems*, pages 1–6, 2005. 23

[61] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Formal reasoning techniques for goal models. *Lecture Notes in Computer Science*, 2800:1–20, 2003. 79

[62] E. Gjorven, F. Eliassen, and J. O. Aagedal. Quality of adaptation. In *Proc. of Int. Conf. on Autonomic and Autonomous Systems*, pages 9–14, 2006. 93, 95

[63] Martin Glinz and Roel J. Wieringa. Guest editors' introduction: Stakeholders in requirements engineering. *IEEE Software*, 24(2):18–20, 2007. 57

[64] J. Gozdecki, A. Jajszczyk, and R. Stankiewicz. Quality of service terminology in IP networks. *IEEE Communications Magazine*, 41(3):153–159, 2003. 30

[65] AR Graves and C. Czarnecki. Design patterns for behavior-based robotics. *IEEE Trans. on Systems, Man and Cybernetics, Part A*, 30(1):36–41, 2000. 97

[66] Philip Greenwood and Lynne Blair. Using dynamic aspect-oriented programming to implement an autonomic system. In *Proc. of Dynamic Aspects Workshop*, pages 76–88, 2004. 26, 73

[67] D. Reidel H. Nurmi. *Comparing voting systems*. Holland Publishing Company, 1987. 90, 91

[68] Rhys Haden. Voice networks. www.rhyshaden.com//voice.htm. 129

[69] S. Haines. *Pro Java EE 5 Performance Management and Optimization.* Apress, 2006. 30, 31, 135, 136

[70] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008. 25, 72

[71] J. Hellerstein, Y. Diao, S. Parekh, and D.M. Tilbury. *Feedback control of computing systems*. IEEE, 2004. 29, 60

[72] Michael G. Hinchey and Roy Sterritt. Self-managing software. *IEEE Computer*, 39(2):107–109, 2006. 11, 15, 16, 31

[73] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology, 2001. http://www-1.ibm.com/industries/government/doc/content/bin/auto.pdf. 10, 21

[74] Adele E. Howe. Improving the reliability of artificial intelligence planning systems by analyzing their failure recovery. *IEEE Trans. on Knowledge and Data Eng.*, 7(1):14–25, 1995. 28

[75] J. C. Hsu. *Multiple comparisons: Theory and methods*. Chapman and Hall, 1996. 111

[76] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008. 8, 23

[77] David Hutchison, Geoff Coulson, Andrew Campbell, and Gordon S. Blair. Quality of service management in distributed systems. pages 273–302, 1994. 30

[78] IBM. Autonomic computing toolkit: Developers guide. Technical Report SC30-4083-03, 2005. 15

[79] Autonomic computing 8 elements, 2001. http://www.research.ibm.com/autonomic/overview/elements.html. 9, 11

[80] Eclipse BtM (Build to Manage). www.ibm.com/developerworks/eclipse/btm. 15, 26

[81] SMART. http://www.almaden.ibm.com/software/dm/SMART/. 22

[82] Standard for software maintenance - IEEE 14764-2006 - ISO/IEC 14764, 2006. URL = http://ieeexplore.ieee.org/iel5/11168/35960/01703974.pdf. 7

[83] J. P. Ignizio. *Goal programming and extensions*. Lexington Books Lexington, Mass, 1976. 95, 120

[84] ISO/IEC 9126-1 Standard: Software Eng. -Product quality - Part 1: Quality model, Int. Standard Organization, 2001. 7, 11, 58

[85] M. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices.* ACM Press/Addison-Wesley, 1995. 25, 42

[86] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley and sons, 1991. 114

[87] Mark Jelasity, Ozalp Babaoglu, Robert Laddaga, Radhika Nagpal, Franco Zambonelli, Emin Gun Sirer, Hakima Chaouchi, and Mikhaill Smirnov. Interdisciplinary research: Roles for self-organization. *IEEE Intelligent Systems*, 21(2):50–58, 2006. 10

[88] J. Jiang, M.S. Kamel, and L. Chen. Aggregation of Multiple Reinforcement Learning Algorithms. *Int. Journal on Artificial Intelligence Tools*, 15(5):855, 2006. 84

[89] Sun Java Management eXtensions. http://jcp.org/en/jsr/detail?id=3. 15, 16, 99, 131

[90] Sun JVM Tool Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/. 15, 16

[91] Gail E. Kaiser, Janak Parekh, Philip Gross, and Giuseppe Valetto. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Proc. of Active Middleware Services*, pages 22–31, 2003. 33

[92] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proc. of Hot Topics in Operating Systems Workshop*, pages 49–54, 2005. 29

[93] Gabor Karsai, Ákos Lédeczi, Janos Sztipanovits, Gábor Péceli, Gyula Simon, and Tamás Kovácsházy. An approach to self-adaptive software based on supervisory control. In *Proc. of Int. Workshop on Self-Adaptive Software*, pages 24–38, 2001. 19, 29

[94] Gabor Karsai and Janos Sztipanovits. A model-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):46–53, 1999. 22, 25

[95] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proc. of IEEE Int. Workshop on Policies for Distributed Systems & Networks*, pages 3–14, 2003. 30

[96] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives.* Wiley, 1976. 28

[97] J. O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003. 6, 8, 10, 13, 73

[98] J. O. Kephart and W. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc. of IEEE Int. Workshop on Policies for Dist. Systems and Networks*, pages 3–13, 2004. 20, 30, 59

[99] J.O. Kephart. Keynote talk: Research challenges of autonomic computing. In *Proc. of Int. Conf. on Software Eng.*, pages 15–22, 2005. 15, 16, 23, 38

[100] M.H. Klein, R. Kazman, L. Bass, J. Carriére, M. Barbacci, and H. Lipson. Attribute-based architectural styles. In *Proc. of the IEEE/IFIP First Workshop Conf. on Software Architecture*, pages 225–243, 1999. 26

[101] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, 1999. 10, 29, 60

[102] F. Kon, F. Costa, G. Blair, and R.H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002. 21

[103] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Proc. of Future of Software Eng.*, pages 259–268, 2007. 61, 85

[104] R. O. Kuehl. *Design of Experiments: Statistical Principles of Research Design and Analysis*. Duxbury Thomson Learning, 2000. 111, 112

[105] V. Kumar, B.F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows. *Cluster Computing*, 10(4):443–455, 2007. 33

[106] Robert Laddaga. Self-adaptive software. Technical Report 98-12, DARPA BAA, 1997. 8

[107] Robert Laddaga. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14(3):26–29, 1999. 6, 24, 26

[108] Robert Laddaga. Active software. In *Proc. of Int. Workshop on Self-Adaptive Software*, pages 11–26, 2000. 6, 12, 24

[109] Robert Laddaga. Self adaptive software problems and projects. In *Proc. of IEEE Workshop on Software Evolvability*, pages 3–10, 2006. 10

[110] Robert Laddaga, Paul Robertson, and Howard E. Shrobe. Results of the 2nd Int. workshop on self-adaptive software. In *Proc. of Int. Workshop on Self-Adaptive Software*, pages 281–290, 2001. 19

[111] Robert Laddaga, Paul Robertson, and Howie Shrobe. Introduction to self-adaptive software: Applications. In *Proc. of Int. Workshop on Self-Adaptive Software*, volume 2614, pages 1–5, 2000. 25

[112] Christopher Landauer and Kirstie L. Bellman. New architectures for constructed complex systems. *Applied Mathematics and Computation*, 120:149–163, May 2001. 16

[113] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software*, pages 1–7, 2005. 18, 25, 27, 33

[114] Alexei Lapouchnian, Yijun Yu, Sotirios Liaskos, and John Mylopoulos. Requirements-driven design of autonomic application software. In *Proc. of IBM Center for Advanced Studies Conf.*, pages 7–22, 2006. 47, 58

[115] M. M. Lehman. Laws of software evolution revisited. In *Proc. of European Workshop on Software Process Technology*, pages 108–124, 1996. 8

[116] MM Lehman and JF Ramil. Towards a theory of software evolution-and its practical impact. In *Proc. of Int. Symposium on Principles of Software Evolution*, pages 2–11, 2000. 8

[117] E. Letier and A. Van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Proc. of ACM SIGSOFT Int. symposium on Foundations of software eng.*, pages 53–62, 2004. 59

[118] Karl J. Lieberherr and Jens Palsberg. Engineering adaptive software, 1993. Projest Proposal, ftp://ftp.ccs.neu.edu/pub/people/lieber/proposal.ps. 8

[119] Sam Lightstone. Foundations of autonomic computing development. In *Proc. of IEEE Int. Workshop on Eng. of Autonomic and Autonomous Systems*, pages 163–171, 2007. 48

[120] Marin Litoiu, Murray Woodside, and Tao Zheng. Hierarchical model-based autonomic control of software systems. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software*, pages 27–33, 2005. 22, 29

[121] Hua Liu and Manish Parashar. Accord: A programming framework for autonomic applications. *IEEE Trans. on Systems, Man and Cybernetics, Part C.*, 36(3):341–352, 2006. 20, 22

[122] Hua Liu, Manish Parashar, and Salim Hariri. A component-based programming model for autonomic applications. In *Proc. of Int. Conf. on Autonomic Computing*, pages 10–17, 2004. 33

[123] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. Qos aspect languages and their runtime integration. In *Proc. of Int. Workshop on Languages, Compilers, and Run-Time systems for scalable computers*, pages 303–318, 1998. 20, 33

[124] Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael A. Bauer. Issues in managing soft qos requirements in distributed systems using a policy-based framework. In *Proc. of IEEE Int. Workshop on Policies for Distributed Systems & Networks*, pages 185–201, 2001. 30

[125] Pattie Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6:49–70, 1990. 27, 51, 54, 86

[126] Pattie Maes. Modeling adaptive autonomous agents. *Artif. Life*, 1(1-2):135–162, 1994. 94

[127] J. Magee and J. Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Eng. Notes*, 21(6):3–14, 1996. 19

[128] Neil Maiden. User requirements and system requirements. *IEEE Software*, 25(2):90–91, 2008. 25, 42

[129] Marco Mamei and Franco Zambonelli. Self-organization in multi agent systems: A middleware approach. In *Proc. of Eng. Self-Organising App. Workshop*, pages 233–248, 2003. 27

[130] E. Martins. jain slee example call-controller-2, 2008. http://code.google.com/p/mobicents/. 126, 127

[131] Mathworks. Matlab simevents toolbox. http://www.mathworks.com/ products/simevents/. 114

[132] Mathworks. Matlab stateflow toolbox. http://www.mathworks.com/ products/stateflow/. 116

[133] D. A. MCALLESTER and D. ROSENBLITT. Systematic nonlinear planning. In *Proc. of AAAI*, pages 634–639, 1991. 83

[134] Julie A. McCann and Markus C. Huebscher. Evaluation issues in autonomic computing. In *Grid and Cooperative Computing Workshops*, pages 597–608, 2004. 95

[135] Julie A. McCann, Rogerio De Lemos, Markus Huebscher, Omer F. Rana, and Andreas Wombacher. Can self-managed systems be trusted?: Some views and trends. *Knowledge Eng. Review*, 21(3):239–248, 2006. 23

[136] Philip K. McKinley, Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, pages 56–64, July 2004. 9, 16, 18, 19, 31, 62

[137] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice-Hall, 2002. 58, 62, 109

[138] D. A. Menascé and M. N. Bennani. Autonomic virtualized environments. In *Proc. of Int. Conf. on Autonomic and Autonomous Systems*, page 28, 2006. 31

[139] D. A. Menascé, M. N. Bennani, and H. Ruan. On the use of online analytic performance models, in self-managing and self-organizing computer systems. In *Proc. of Self-star Properties in Complex Information Systems*, pages 128–142, 2005. 125

[140] D. A. Menascé, H. Ruan, and H. Gomaa. QoS management in service-oriented architectures. *Performance evaluation*, 64(7-8):646–663, 2007. 94

[141] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *Proc. of Int. Workshop on Software eng. for Adaptive and Self-Managing Systems*, pages 9–16, 2008. 27

[142] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of Int. Conf. on Architecture of Computing Systems*, pages 124–138, 2005. 15, 16, 19, 33

[143] Hausi A. Muller. Bits of history, challenges for the future and autonomic computing technology. In *Proc. of Working Conf. on Reverese Eng.*, pages 9–15, 2006. 15

[144] Richard Murch. *Autonomic Computing*. Prentice Hall, 2004. 23

[145] B. A. Nardi. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996. 105

[146] S. Neti and H.A. Muller. Quality Criteria and an Analysis Framework for Self-Healing Systems. In *Proc. of the Int. Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 6, 2007. 33

[147] Tomasz Nowicki, Mark S. Squillante, and Chai Wah Wu. Fundamentals of dynamic decentralized optimization in autonomic computing systems. In *LNCS*, volume 3460, pages 204–218, 2005. 28

[148] Phelim O'Doherty. Jain slee principles, 2003. http://java.sun.com/products/jain/article_slee_principles.html. 99

[149] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999. 8, 10, 13, 16, 19, 20, 23, 26, 27

[150] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proc. of Int. Conf. on Software Eng.*, pages 177–186, 1998. 19, 73

[151] C. H. Papadimitriou. Game theory and mathematical economics: a theoretical computer scientist's introduction. In *Proc. of Foundations of Computer Science*, pages 4–8, Oct. 2001. 52

[152] M. Parashar and S. Hariri. Autonomic computing: An overview. *Hot Topics, Lecture Notes in Computer Science*, 3566:247–259, 2005. 11, 22

[153] Janak Parekh, Gail Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006. 15

[154] David Lorge Parnas. Software aspects of strategic defense systems. *Communications of ACM*, 28(12):1326–1335, 1985. 27

[155] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995. 60, 64

[156] R. Patrascu, C. Boutilier, R. Das, J. O. Kephart, Gerald Tesauro, and William E. Walsh. New approaches to optimization and utility elicitation in autonomic computing. In *Proc. of Conf. on Artificial Intelligence*, pages 140–145, 2005. 28

[157] Eddy Truyen Wouter Joosen Paul Grace, Bert Lagaisse. A reflective framework for fine-grained adaptation of aspect-oriented compositions. In *Proc. of Int. Symp. on Software Composition*, pages 215–230, 2008. 73

[158] Dusko Pavlovic. Towards semantics of self-adaptive software. In *Proc. of Int. Workshop on Self-Adaptive Software*, volume 1936 of *Lecture Notes in Computer Science*, pages 65–74, 2000. 25

[159] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proc. of Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, 2001. 15, 16, 22, 26

[160] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003. 26

[161] Nigel C. Smeeton Peter Sprent. *Applied Nonparametric Statistical Methods*. Chapman & Hall/CRC, 4th edition, 2007. 121

[162] M. Pinto, L. Fuentes, ME Fayad, and JM Troya. Separation of coordination in a dynamic aspect oriented framework. In *Proc. of the Int. Conf. on Aspect-oriented software development*, pages 134–140, 2002. 19

[163] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. *Multiagent Systems Artificial Societies and Simulated Organizations*, 15:149, 2005. 47

[164] Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. Dynamic configuration of resource-aware services. In *Proc. of Int. Conf. on Software Eng.*, pages 604–613, Washington, DC, USA, 2004. IEEE Computer Society. 28

[165] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. of Int. Conf. on Aspect-Oriented Software Development*, pages 141–147, 2002. 15, 16

[166] Stefano Porcarelli, Marco Castaldi, Felicita Di Giandomenico, Andrea Bondavalli, and Paola Inverardi. A framework for reconfiguration-based fault-tolerance in distributed systems. In *Proc. of ICSE Workshop on Architecting Dependable Systems II*, Lecture Notes in Computer Science, pages 167–190, 2003. 28

[167] G. Qu and S. Hariri. *Autonomic computing: concepts, infrastructures, and applications*, chapter Anomaly-based self-protection against network attacks, pages 493–521. CRC, 2007. 32

[168] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987. 29

[169] A.S. Rao and M.P. Georgeff. BDI agents: From theory to practice. In *Proc. of Int. Conf. on Multi-Agent Systems*, pages 312–319, 1995. 47

[170] ITU-T Rec. Support of IP based services using IP transfer capabilities. *ITU-T Recommendation Y*, 1241, 2001. 30

[171] Paul Robertson and Robert Laddaga. Model based diagnosis and contexts in self adaptive software. In *Proc. of Self-* Properties in Complex Information Systems*, pages 112–127, 2005. 10, 28, 29, 33

[172] Paul Robertson and Brian Williams. Automatic recovery from software failure. *Communications of ACM*, 49(3):41–47, 2006. 19, 28

[173] Julio Rosenblatt. DAMN: a distributed architecture for mobile navigation. *Journal Exp. Theory Artificial Intelligence*, 9(2-3):339–360, 1997. 52, 54

[174] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Prentice-Hall, 1995. 27, 28, 83

[175] Boutros Saab, Xavier Bonnaire, and Bertil Folliot. Phoenix: A self adaptable monitoring platform for cluster management. *Cluster Computing*, 5(1):75–85, 2002. 31

[176] S. M. Sadjadi, P. K. McKinley, B. H. C. Cheng, and R. E. K. Stirewalt. TRAP/J: Transparent generation of adaptable Java programs. *Lecture Notes in Computer Science*, 3291:1243–1261, 2004. 15, 16, 33

[177] Seyed Masoud Sadjadi and Philip K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proc. of Int. Conf. on Dist. Computing Systems*, pages 74–83, 2004. 26, 33

[178] Mazeiar Salehie, Sen Li, Reza Asadollahi, and Ladan Tahvildari. Change support in adaptive software: A case study for fine-grained adaptation. In *Proc. of IEEE Conf. and Workshops on Eng. of Autonomic and Autonomous Systems*, pages 35–44, 2009. 4, 30, 73, 74, 98, 105

[179] Mazeiar Salehie, Sen Li, and Tahvildari Ladan. Employing aspect composition in adaptive software systems: A case study. In *Proc. of AOSD Workshop on Practices of Linking Aspect Technology and Evolution*, pages 17–21, 2009. 4, 98, 105

[180] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing: emerging trends and open problems. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software*, pages 82–88, 2005. 10, 11, 58

[181] Mazeiar Salehie and Ladan Tahvildari. A policy-based decision making approach for orchestrating autonomic elements. In *Proc. of IEEE Int. Workshop on Software Tech. & Eng. Prac.*, pages 173–181, 2005. 28, 85, 98, 110, 113

[182] Mazeiar Salehie and Ladan Tahvildari. Coordinating self-healing and self-optimizing in autonomic elements: an experiment. In *Proc. of Workshop on Software Eng. for Adaptive and Self-Managing Systems*, page 98, 2006. 2

[183] Mazeiar Salehie and Ladan Tahvildari. Action selection in self-adaptive software using social choice theory. Technical Report UW-ECE-2007-17, University of Waterloo, 2007. 116

[184] Mazeiar Salehie and Ladan Tahvildari. A quality-driven approach to enable decision-making in self-adaptive software. In *Companion to the proc. of Int. Conf. on Software Eng. (Doctoral Symposium)*, pages 103–104, 2007. 1

[185] Mazeiar Salehie and Ladan Tahvildari. A weighted voting mechanism for action selection problem in self-adaptive software. In *Proc. of IEEE Conf. on Self-Adaptive and Self-Organizing Systems*, pages 328–331, 2007. 1, 4, 25, 73, 87, 98, 124

[186] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. on Autonomous and Autonomic Systems*, 4(2):1–42, May 2009. 4, 7, 29, 38, 48, 94, 149

[187] D. C. Schmidt. Middleware for real-time and embedded systems. *Communication of ACM*, 45(6):43–48, 2002. 9, 18, 31

[188] D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible orb middleware. *IEEE Communications Magazine*, 37:54–63, 1999. 15, 16

[189] J. Scott, S. Neema, T. Bapty, and B. Abbott. Hardware/software runtime environment for dynamically reconfigurable systems. Technical Report ISIS-2000-06, Vanderbilt University, 2000. 25

[190] Giovanna Di Marzo Serugendo, Noria Foukia, Salima Hassas, Anthony Karageorgos, Soraya Kouadri Mostéfaoui, Omer F. Rana, Mihaela Ulieru, Paul Valckenaers, and Chris van Aart. Self-organisation: Paradigms and app. In *Proc. of Eng. Self-Organising App. Workshop*, pages 1–19, 2003. 10

[191] W.R. Shadish, T.D. Cook, and D.T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference.* Houghton Mifflin Harcourt, 2001. 124

[192] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline.* Prentice-Hall, 1996. 46

[193] JRat (Java Runtime Analysis Toolkit). http://jrat.sourceforge.net/. 15

[194] H. A. Simon. *The New Science of Management Decision.* Harper and Brothers, 1960. 59

[195] Morris Sloman. Policy driven management for distributed systems. *J. Network Syst. Manage.*, 2(4), 1994. 30

[196] Simple network management protocol. http://www.ietf.org/html.charters/OLD/snmp-charter.html. 15, 16

[197] Biplav Srivastava and Subbarao Kambhampati. The case for automated planning in autonomic computing. In *Proc. of Int. Conf. on Automatic Computing*, pages 331–332, 2005. 27, 83

[198] Roy Sterritt. Autonomic computing: the natural fusion of soft computing and hard computing. In *IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 5, pages 4754–4759, 2003. 21

[199] Roy Sterritt and David W. Bustard. Autonomic computing - a means of achieving dependability? In *Proc. of IEEE Symp. and Workshops on Eng. of Computer-Based Systems*, pages 247–251, 2003. 11

[200] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Eng. Informatics*, 19:181–187, Jul 2005. 7, 9, 10, 26

[201] Philip D. Straffin, Jr. *Topics in the Theory of Voting*. The UMAP Expository Monograph Series. Birkhäuser, 1980. 52, 90, 91

[202] N. Subramanian and L. Chung. Software architecture adaptability: An nfr approach. In *Proc. of Int. Workshop on Principles of Software Evolution*, pages 52–61, 2001. 25

[203] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proc. of Int. Conf. on Aspect-oriented software development*, pages 21–29. ACM New York, NY, USA, 2003. 19

[204] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation*. Pearson Prentice Hall, third edition, 2006. 9

[205] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White. A multi-agent systems approach to autonomic computing. In *Proc. of Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 464–471, 2004. 27

[206] Gerald Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30, 2007. 28

[207] S. Tuttle, V. Batchellor, M. Bodstrup Hansen, and M. Sethuraman. Centralized risk management using tivoli risk manager 4.2. Technical report, IBM Tivoli Software, December 2003. 33

[208] Gregoris Tziallas and Babis Theodoulidis. A controller synthesis algorithm for building self-adaptive software. *Information & Software Tech.*, 46(11):719–727, 2004. 29

[209] Giuseppe Valetto and Gail Kaiser. Using process technology to control and coordinate software adaptation. In *Proc. of Int. Conf. on Software Eng.*, pages 262–273, 2003. 33

[210] F. Van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action the Best Industrial Practice in Product Line Eng.: The Best Industrial Practice in Product Line Eng.* Springer, 2007. 62

[211] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* John Wiley and Sons, 2009. 56, 57, 58, 59, 63, 65, 72, 81

[212] Kunal Verma and Amit P. Sheth. Autonomic web processes. In *Proc. of Int. Conf. on Service-Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science*, pages 1–11, 2005. 26

[213] J. Q. Walker. Assessing voip call quality using the e-model. Technical report, NetIQ Corporation, 2001. 134

[214] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proc. of IEEE Conf. on Autonomic Computing*, pages 70–77, 2004. 28

[215] Web-based enterprise management standard. http://www.dmtf.org/standards/wbem/. 15

[216] A. Westerinen, J. Schnizlein, J. Strassner, Mark Scherling, Bob Quinn, Jay Perry, Shai Herzog, An-Ni Huynh, and Mark Carlson. Policy terminology. IETF, Internet Draft draftietf-policy-terminology-00.txt, 2000. 30

[217] Danny Weyns, Kurt Schelfthout, and Tom Holvoet. Architectural design of a distributed application with autonomic quality requirements. *SIGSOFT Software Eng. Notes*, 30(4):1–7, 2005. 27

[218] Jules White, Douglas C. Schmidt, and Aniruddha S. Gokhale. Simplifying autonomic enterprise java bean applications via model-driven development: A case study. In *Proc. of Int. Conf. on Model Driven Eng. Languages and Systems*, pages 601–615, 2005. 33

[219] M. H. Willebeek-LeMair, A. P. Reeves, and Y. Heights. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(9):979–993, 1993. 19

[220] M. Woodside and D. A. Menascé. Guest editors' introduction: Application-level QoS. *IEEE Internet Computing*, 10(3):13–15, 2006. 31

[221] E. Yu and J. Mylopoulos. Understanding why in software process modelling, analysis, and design. In *Proc. of Int. Conf. on Software eng.*, pages 159–168, 1994. 43

[222] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. Leite. From goals to high-variability software design. *Lecture Notes in Computer Science*, 4994:1, 2008. 33

[223] H. Zhang and S. Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, 53(3):381–407, 2004. 72