Coordinating Self-Healing and Self-Optimizing Disciplines in Autonomic Elements

By
Mazeiar Salehie and Dr. Ladan Tahvildari
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario

TECH. REPORT UW-ECE#2006-06
April 2006
© Mazeiar Salehie and Dr. Ladan Tahvildari, 2006
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>vi</td>
</tr>
<tr>
<td>1 Modelling</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Modelling Autonomic Elements and their Properties</td>
<td>3</td>
</tr>
<tr>
<td>1.3.1 A Generic Model for Autonomic Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3.2 Self-Healing</td>
<td>4</td>
</tr>
<tr>
<td>1.3.3 Self-Optimizing</td>
<td>7</td>
</tr>
<tr>
<td>2 Coordination Mechanism</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Planning-Based Coordinating</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Execution Coordinating</td>
<td>11</td>
</tr>
<tr>
<td>3 Experiments</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.2 The Experimental Model</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1 Realizing Self-Optimizing Discipline</td>
<td>14</td>
</tr>
<tr>
<td>3.2.2 Realizing Self-Healing Discipline</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Obtained Results</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Conclusion and Future Works</td>
<td>18</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Common Actions for Self-Healing. . . . . . . . . . . . . . . . . . . . . 6
1.2 Common Actions for Self-Optimizing. . . . . . . . . . . . . . . . . . . 7
3.1 SLA for Gold and Silver Services. . . . . . . . . . . . . . . . . . . . . 15
List of Figures

2.1 Potential Coordination Joint Points in an Autonomic Element. . . . . . . 10

3.1 The Experimental Autonomic System. . . . . . . . . . . . . . . . . . . 13
3.2 A Typical Autonomic Element Model. . . . . . . . . . . . . . . . . . . 14
3.3 Coordination Effect on Element Utility. . . . . . . . . . . . . . . . . . 17
3.4 Replanning for the Failure Recovery. . . . . . . . . . . . . . . . . . . 17
Abstract

There is an increasing demand for self-adaptivity in software systems. Such systems offer the promise of controlling complexity through the achievement of self-governance (autonomy) and self-management (autonomicity). In a nutshell, they are based on three principles of separation of concerns, decentralization and policy-based management. In practice, often only one concern has been taken into account to make this paradigm a realization. However having common policies and shared resources, it is required to coordinate different concerns. This research work addresses the problem of coordinating two autonomic disciplines namely, self-healing and self-optimizing, in autonomic elements. First a generic form of these two disciplines is presented, and then the effect of applying a novel coordination mechanism on an experiment will be discussed.
Chapter 1

Modelling

1.1 Introduction

In order to cope with the ever increasing complexity of managing distributed, heterogeneous computing systems, it has been proposed that systems themselves should manage their own behavior. Autonomic Computing, introduced by IBM [9], is one of such self-managing paradigms. IBM has proposed a five-layer reference architecture for autonomic systems [1], containing managed resources, touch-points, autonomic managers and orchestrators. The important point in each layer is the role of policies. Such policies may be defined globally (i.e. business policies) which will be decomposed to local level policies.

If we define the collection of an autonomic manager and its managed resources as an autonomic element, local policies specify objectives and constraints for such an element. Policy-based management of an autonomic system is not possible without coordinating autonomic elements. The coordinating are both within and across autonomic disciplines called self-CHOPs (self-Configuring, -Healing, -Optimizing, -Protecting).

Because of the decentralized nature of management in autonomic systems and the fact that each autonomic manager implements all the self-CHOPs, coordination can not always be handled at the orchestration layer of the reference architecture. This paper focuses on orchestrating across two self-CHOPs namely, self-optimizing and self-healing in autonomic elements and how it can be performed regarding designated policies.

The remainder of this paper is organized as follows. Section 1.2 gives a brief
description of the focused problem. Section 1.3 reviews the generic models for self-healing and self-optimizing. Section 2.1 describes the possible mechanisms for coordination these two disciplines. Section 3.1 reviews the experiment performed for studying coordination effects on a simple autonomic element. Finally, Section 3.4 summarizes the lessons learned and possible directions for future work.

1.2 Problem Statement

Three major ideas of autonomic computing can be summarized in the following items: 1) Separation of concern for dynamic management (mostly for of non-functional aspects), and 2) Policy-based and goal-oriented management based on flexible policies, at both local and global levels, and 3) Decentralized management of an autonomic system. These ideas are realized at the element level by a MAPE-K loop (Monitoring, Analyzing, Planning, Executing and Knowledge-base) according to IBM proposed reference architecture. But during dynamic adaptation, the ultimate decision/plan should be coordinated toward defined policies.

The four autonomic disciplines (self-CHOPs) act like four managers, each one with its own closed loop, which should behave according to policies. Policies specify goals and constraints of the system and its constituent elements. Any change in the context/self of an element is sensed by all four disciplines, so based on the symptoms each one may suggest an action plan. Now the question is which solution or combination of solutions is optimal regarding the policies.

For this paper, we focus on the problem of coordinating self-optimizing and self-healing. In the next section, we will elaborate further on details of these two disciplines. In case of a failure in a system both disciplines react simultaneously by actions (e.g. resource provisioning and restarting), which may impact the system state and dependency model. These actions are based on the local policies defined for the element, but each discipline’s action may lead to a worse or chaotic state for the other. For example a failure may be recovered by a cheap reboot (or microreboot) planned by the self-healing discipline, but self-optimizing requests a costly resource provisioning simply because the existing resources are not enough. To the best of our knowledge, there is no proposed mechanism for coordinating self-CHOPs disciplines, especially self-optimizing and healing, inside an autonomic manager.
The proposed architectures, such as IBM reference architecture, suggest to coordinate these disciplines at the system-level through two common configurations [1] namely, *Orchestrating within a discipline* and *Orchestrating across disciplines*. But regarding the decentralized management principle in Autonomic Computing, managing and decision making is better to be localized as much as possible. Coordinating these disciplines needs some local information that are difficult to handle at the global level, so there also a need for local coordination.

1.3 Modelling Autonomic Elements and their Properties

This section gives a generic model of an autonomic element in order to help us in modelling self-healing and self-optimizing. We list the common response actions for these two disciplines in order to discuss on their impacts on each other, potential conflicts and locating the coordination points. These impacts can be studied better on the basis of a categorization. Dynamic adaptation actions is broadly categorized to two classes of: *parameter adaptation* and *compositional adaptation* [11]. Parameter adaptation modifies system variables that determine behavior, while compositional adaptation exchanges algorithmic or structural aspects. We can call these two types as *soft* and *hard* adaptation. The goal is to coordinate both types of actions in two disciplines.

1.3.1 A Generic Model for Autonomic Systems

An autonomic element should model itself and its context to be aware of changes, and to take the right response. The following entities compose a generic model for such an autonomic element:

- **SD**: State diagram(s) of the system
- **DM**: Dependency model of the system
- **A**: Action set (primitive/non-primitive)
- **PS**: Policy set
- **E**: Event set
- **S**: Sensor set

*SD* gives the state model of the element in which states are connected by events in *E* and/or actions in *A*. *PS* designates goals and constraints of the system which is
subject to change during the system operation. These policies determine the strategy of selecting actions in response to changes in the context/self. $DM$ describes the structure and architecture of the system, and $S$ is the sensor set for measuring different metrics related to each autonomic discipline.

Generally, it is difficult (if not possible) to build a single state diagram, even non-deterministic, for a given system. This is mainly because of different concerns that the states must represent. For instance, states in self-healing discipline have to show whether an entity is healthy (active and connected) or there is an error, fault or failure. These states are meaningless for the self-optimizing discipline, because it is monitoring totally different aspects of the system. So, the states should be modelled by different state diagrams.

One of the current research lines is to develop tools and methodologies to incorporate planning and its underlying reasoning in the runtime decision-making process. Planning is defined as a key part of MAPE-K loop in the proposed autonomic element architecture [9], and the emphasis is on the dynamic decision making to achieve the element’s goals. In other word, planning is a way to enforcing goal-based policies.

### 1.3.2 Self-Healing

There is not a unique comprehensive definition for self-healing. Researchers have differing point of views on which areas are involved in this research area and its relationship with the existing disciplines such as fault tolerance and dependability. IBM [1] definition is: Self-healing is the ability of detecting system malfunctions and initiating policy-based corrective action without disrupting the IT environment.

At the element level, self-healing is defined by the following entities:

- $PS$: Policy set
- $E$: Event set
- $S_h$: Sensor set for self-healing
- $A_h$: Actions (primitive/non-primitive)
- $C(A_i)$: Cost of action $A_i$
- $H$: history of actions in states
- $SD_h$: State diagram of the system
- $DM_h$: Dependency model

Several entities, such as $A_h$, for self-healing discipline use a subset of the generic set. This does not mean necessarily that those entities are exclusive subsets for this discipline, but there may be some common members with the corresponding sets in
the other disciplines. From these entities, $S_h$ and $DM_h$ are the most specific subsets for self-healing. The former is because of sensing information relating to errors, faults and failures (e.g. heart beat monitoring) which mostly are useless for the other disciplines, especially for self-optimizing. History of actions ($H$) is needed because in case of unsuccessful recovery another plan of action should be selected and executed.

Arshad et al. propose a simplified ADL-based model ($S_h$) for state models of components, connectors and machines [2], and also a dependency model ($DM_h$) for system entities [3]. State models for components and connectors are one of active, inactive, connected and killed, while for machines are one of up, down or killed. Dependency model determines hard or soft dependencies between system components which, in a sense, gives the fault propagation model. Generally, the existing architecture models and languages (e.g. ADL) are not suitable for self-healing without adding additional information to the architecture (i.e. annotating fault information to dependencies). Fox et al. [6] propose building a fault map (called f-map) to model dependency of system components in case of faults. This model shows the fault propagation and ripple effects after occurring a fault in a component. However, building and maintaining such a map is not a straightforward task, especially in complex systems.

If an event $E_h^j$ changes the system state from $S_0$ to $S_1$ then we can model the self-healing closed-loop as a problem of: given $A_h$, $P_h$, $E_h$, $C(A)$, $S_0$, $S_1$ and $D$ find an action sequence $< A_1...A_i >$ to recover the system from $S_1$ to $S_0$ by minimum cost. Of course, it may not be possible to rollback to $S_0$. In this case, the system needs a new configuration by actions such as reorganizing. Table 1.1 shows the list of common actions for recovery in self-healing. Restarting may be performed in different levels of granularity. For instance it may be in form of Microreboot [6] at the component level as one of the cheapest recovery actions.

Sometimes, it may need to perform an action plan containing several actions. For instance after provisioning the new added resource should be configured for use. This configuration contains deploying of required services/servers which has the effects similar to redeploying.

According to [19] planning has the best potential of use in self-healing. Planning can be be seen as a mechanism for enforcing goal policies. Generally, a planning problem can be presented as a 5-tuple $\langle I, G, A, E, C \rangle$ where $I$ is the initial state, $G$ is the goal state(s), $A$ is the set of primitive actions, $E$ is the set of existing plans,
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Adaptation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restarting</td>
<td>Restarting services, servers or components</td>
<td>Soft adaptation</td>
</tr>
<tr>
<td>Reorganizing</td>
<td>Reorganizing the system</td>
<td>Hard adaptation</td>
</tr>
<tr>
<td>Redeploying</td>
<td>Redeploying services or servers</td>
<td>Hard adaptation</td>
</tr>
<tr>
<td>Provisioning</td>
<td>Add additional resources</td>
<td>Hard adaptation</td>
</tr>
</tbody>
</table>

Table 1.1: Common Actions for Self-Healing.

and finally $C$ is constraints either imposed by the environment or managers of the system. The planning algorithm may search in state/action space or plan space to find the action plan. In case of the plan space search, it needs $E$.

The planning entities can be mapped to the aforementioned self-healing entities. $C$ is defined by policies in $PS$, $A$ by $A_h$ and $H$ and finally $I$ and $G$ by $SD_h$ and $DM_h$. In practice, for finding the optimal action plan, the planning algorithm also needs cost of actions $C(A_i)$. Planning entities also can be categorized as domain, initial state and goal state [2], which domain is the collection of actions, utilities, state and dependency models.

Because for any state $S_i$, there may be several actions, the goal is to minimize the expected cost based on the following equation [8]:

$$EC(S_i) = C(A_1) + (1 - P(A_1|S_i))[C(A_1) + \ldots + (1 - P(A_{n-1}|S_i))[C(A_n) + (1 - P(A_n|S_i))[C_F]...]$$

(1.3.1)

$P(A_k|S_i)$ is the probability that action $A_k$ is successful if applied in state $S_i$ and $C_F$ is the cost of failing all actions. This equation assumed that cost functions $C(A_j)$ are independent of $S_i$ and order of execution of the recovery methods. Also $P(A_j|S_i)$ is independent of order of execution of the recovery methods. Simon et al. [16] showed that for the expected cost formulated by this equation will be minimized by the strategy of trying the methods in decreasing order of $P(A_j|S_i)/C(A_j)$.

The planning phase of an autonomic manager implements a continuous planning.
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Adaptation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuning</td>
<td>Tuning resources and parameters</td>
<td>Soft adaptation</td>
</tr>
<tr>
<td>Reorganizing</td>
<td>Reorganizing the system</td>
<td>Hard adaptation</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Balancing the load between servers/services/machines</td>
<td>Soft adaptation</td>
</tr>
<tr>
<td>Provisioning</td>
<td>Add additional resources</td>
<td>Hard adaptation</td>
</tr>
</tbody>
</table>

Table 1.2: Common Actions for Self-Optimizing.

For this purpose, there are two planning technique, namely *conditional planning* (contingency planning) and *replanning* (execution monitoring) [13]. The first technique gives a conditional plan with alternative paths based on sensed information during plan execution, while the second one generates only one plan in case of failure will generate an alternative plan. For planning corrective actions (recovery), one of these technique has to be used, because the corrective actions might not be successful there should be a mechanism for following up alternatives or planning new action plans.

1.3.3 Self-Optimizing

Components with *Self-Optimizing* capability can tune themselves to meet end-user or business needs [1]. Table 1.2 lists the common actions for self-optimizing in a typical autonomic system. Several action are similar to self-healing actions. Self-optimizing in an autonomic system is based on a decentralized optimization [12]. Elements perform the local optimization tasks on assigned resources and the global optimization issues will be solved only by part of element-level information (e.g. utility function value). An example of such global issues is resource provisioning which must take into account conflicts and priorities at the global level. At the element level by considering the following parameters:

- \( A_o \): Actions (primitive/non-primitive)
- \( PS \): Policy set
- \( E \): Event set
- \( S_h \): Sensor set for self-healing
- \( SD_s \): State diagram of the system
- \( U(.) \): Utility function
$DM_o$: Dependency model

$C(A_i)$: Cost of action $A_i$

we can model the self-optimizing closed-loop as a problem of: given $A_o$, $P_o$, $E_o$, $C(A_i)$ and $U(.)$ find an action sequence $<A_1...A_i>$ to optimize $U(.)$ (maximize or minimize). Because the conditions of the system (self) and the environment (context) are subject to change, the action sequence may be changed during execution.

In practice, for such a continuous optimization planning a deterministic sequence of action is not often feasible, even in existence of a perfect utility function for specific parameters. The proposed mechanisms, normally are based on a modelling or estimation of the system behavior. For instance, Litoiu et al. propose a hierarchical mechanism to control the system using model estimation and regarding performance and QoS.

Policies, particularly business policies, play an important role in self-optimizing. There are various research works in the domain of optimizing the IT-based systems regarding business policies. For example, Gilat et al. [7] define business objectives and SLA (as per day and per transaction benefit, and performance penalty), and test a business-based self-optimizing method on a simulated system. The interesting part of their work is modelling the user behavior and load breakdown between servers in a multi-tier system.
Chapter 2
Coordination Mechanism

2.1 Introduction

Two principle questions about coordinating self-healing and self-optimizing disciplines are: 1) what actions need to be coordinated, and 2) when (in which part of MAPE-K closed loop) is the appropriate time for coordinating those actions.

To answer the first question, we first should take a look at relationship between two disciplines and their actions. In [14], we have discussed about dependency between autonomic characteristics and quality factors, without focusing on their inter-relationships. Lin et. al [10] also propose a quality metrics framework for Autonomic Computing, again without discussing about relationships of self-CHOPs. For coordinating self-healing and optimizing, their impacts to the element self/context are important, and these impacts are result of the set of actions in $A_h$ and $A_o$. Categorizing these actions to two classes of soft and hard adaption actions helps us to study better their impact (As shown in Table 1.1 and Table 1.2).

Soft adaptation actions in each discipline, mostly, change variables which are specific to that discipline. For instance, tuning adjusts CPU, memory or bandwidth share to optimize an aspect of the system behavior, and these changes are not likely to change the health state of the system ($SD_h$) or $DM_h$. So, for coordinating, soft adaptation action can mostly execute simultaneously. But, hard adaptation actions have probably disturbing impacts on each other.

To answer the second question, we need to find joint-point(s) in the disciplines’ closed-loops. Because monitoring and analyzing parts of the closed loop are specifically related to the nature of each discipline, and are also before determining the
change requests cannot be options for joint-points. So, as depicted in Figure 2.1 joint points can be either in planning or executing parts of the closed loop. Two closed loops may monitor similar or different attributes of managed resources, and even after the joint point may execute parallel tasks. An example for such a case is reacting in response of a failure in the system, which self-optimizing loop tries to compensate the lack of resources by tuning while self-healing deals with recovery of the failed resource. The following subsections elaborate further these join-points.

### 2.2 Planning-Based Coordinating

The first joint-point in the planning part of the closed-loop needs a *planning-based coordination*. This coordination, in fact, is finding two optimal action plan by considering coordination actions in each plan. It is complex because each of these disciplines relies on a complex algorithm for finding the optimal action plan and adding another constraint increases the complexity and in consequence the time of coordinating. The computation time is very critical because all the decision-making process occurs continuously at run-time. Reacting by a near-optimal may be better than later optimal actions. In case of using AI planning methods, still this joint-point is not appropriate. The reason is because for self-optimizing, as mentioned by [19], it is better to use a plan-space search rather than a state-space search, while for self-healing this not always true.

One possible solution is to consider coordinated actions (pair of actions) in design time which is applicable by aspect-oriented methods. As dynamic decision making, the planning-based coordination resembles to a hierarchical planning [13] in which some actions (or operators in planning terminology) are non-primitive. For example,
an existing plan may be the normal process of tuning parameters. But the point is
the effects of some actions may be conditional, in particular in self-healing. It means
that the effect of an action depends on what is true when it is executed. For instance
in self-healing, restarting a service may not be enough for recovery because a failure
in operating system or the server. So, after executing an action, the plan may be
needed to modify.

2.3 Execution Coordinating

The second joint-point is in the execution part of the closed-loop. It is mainly a kind
of scheduling mechanism. It is more feasible to implement because two action plans
have been found already and now the problem is finding the timing of their execution
regarding the impacts of each action (soft or hard adaptation) and moreover the
policies governing the behavior of the element. For realizing such a coordination
mechanism, we need to compare cost of actions of two disciplines, to estimate the
possible changes in the system utility function (expected cost) and to predict possible
disturbances to the other discipline.

The other point is that, nature of self-optimizing is a continuous process with
proactive and reactive response, while self-healing is a once-in-while process mostly
based on reactive response. Therefore, a decision tree can be used in the second joint-
point, to separate the normal situations of the system with the cases that a problem
(fault or failure) has been occurred. We use the second joint-point in the experiment
performed in the next section.
Chapter 3

Experiments

3.1 Introduction

The goal of this experiment is studying the effect of coordinating self-healing and self-optimizing within an element. We use our previous work [15] which implements a policy-based framework for self-optimizing. The autonomic manager of each element uses utility-based policies for the self-optimizing discipline. For this paper, we use the previous implemented model in addition of incorporating a planning-based self-healing discipline and an across-discipline coordination mechanism. Our focus is on the behavior of self-healing and self-optimizing closed loops in response to changes in the system.

3.2 The Experimental Model

Figure 3.1 depicts the system model for this experiment. It is an abstract model of a simple data center containing two autonomic elements. Each element manages a multi-tier application, which means that managed resources are several machines running web servers, application servers and database servers. Global Autonomic Manager (GAO) is responsible for coordinating self-optimizing in both elements, in particular the global resource provisioning. The main global policy for this system is maximizing the profit which is defined by a utility function (this utility function also is defined at the local level). The other global policy is about business value of each of the applications running on elements, which determines the priority for resource provisioning (of course regarding maximization of the total profit). In the
experiment, we assume that autonomic elements has no resource dependency to each other except their competition on the shared resource pool.

Figure 3.1: The Experimental Autonomic System.

Figure 3.2 depicts the internal model of an autonomic manager. Utility measurement module uses a utility function U which calculates the element utility in terms of the services response time, SLA constraints and penalties. The interpretation of the utility function in this paper relates more to business objectives and SLA in [21]. In fact, local utility-based policies are used to maximize the business profit through not violating SLA for different services.

In the experiments, it has been assumed that every 50 simulation steps is equal to one day and the default user traffic (for silver and gold services) in these experiments is a ramp with slope 1. A traffic pattern closer to reality is addressed in [17], but the this function has been used because the model can be tested better due to the expected workload, especially for competition in resource provisioning.

For this experiment, we use restarting (servers and machines) and server/maching provisioning as self-healing actions, and tuning and server provisioning as self-optimizing actions. The experimental model has been implemented by IBM Agent Building and Learning Environment (ABLE) [4] in Java. ABLE is a Java framework and component library for building intelligent agents using machine learning and reasoning. GAO, autonomic managers, and elements are implemented by Java in this implementation strategy. Utility-based policies (fuzzy policies) and planning are realized by ABLE Rule Language (ARL).
3.2.1 Realizing Self-Optimizing Discipline

The autonomic manager in each element should adjust CPU shares to appropriate levels for each service based on predefined local policies. The mechanism is similar to the work of Tesauro et al. [20], which uses utility-based policies for the data center model to maximize the total utility calculated by each autonomic manager. This utility is calculated based on the current resource level, demand and predicted demand. The autonomic manager in the given model does not use a prediction algorithm, but by monitoring the resource level threshold and the system performance (response time), it determines the resource demand.

For this model, managed resources are servers (e.g. web servers) which by input requests, and total available CPU from servers, denoted as \( C_{Total} \), give the average response time, denoted as \( RT \). SLA in this experiment define two types of services, gold and silver. So there are two average response times for two services, \( RT_G \) for gold and \( RT_S \) for silver. Table 3.1 describes the SLA used in the experiments on autonomic manager. Each user gives a flat fee for daily service, gold or silver, and the system calculates a penalty for violation of the agreement on the response time.

Utility-based policies are defined using utility functions of gold and silver services. These utility functions measure utility of each service continuously based on the
response time and SLA. For the element level, the penalty is used as the utility measure, because it clearly shows the trend of profit at the element level.

We define the following fuzzy variables for policies: $z_{\text{GoldRT}}$ for $RT_G$, $z_{\text{SilverRT}}$ for $RT_S$, and $z_{\text{FreeCPU}}$ for FreeCPU. The following policy is a sample utility-based policy for poor response time of the gold service (indicated by Poor membership function - MF), while the silver service response time is excellent (indicated by Excellent MF), and there is high amount of free CPU (indicated by High MF). Preconditions for the element level are as following: penalty for the gold service ($P_G$), penalty for silver service ($P_S$), and available free CPU (FreeCPU). For fuzzy utility policies, similar to action ones, corresponding fuzzy variables have been defined as: $z_{\text{GoldPenalty}}$ as $P_G$, $z_{\text{SilverPenalty}}$ as $P_S$, and $z_{\text{FreeCPU}}$ as FreeCPU. The following utility-based policy is a sample of fuzzy utility policies used in the model:

$$
\text{if (} z_{\text{GoldPenalty}} \text{ is Zero) and (} z_{\text{SilverPenalty}} \text{ is High) and (} z_{\text{FreeCPU}} \text{ is High) then}
\text{ (} z_{\text{SilverCPUShare}} \text{ is IncFast)}
$$

The action for tuning the CPU share for silver service is to increase the share fast, which indicates with IncFast. Steady is for normal state which no action is required while IncFast and DecFast tune the CPU share respectively by increasing and decreasing it. The range of horizontal axis is $[-7,7]$, which means tuning can increase/decrease up to 7 units (or 7%) of CPU share.

### 3.2.2 Realizing Self-Healing Discipline

For the self-healing discipline, we use a planning-based mechanism. The planner is a classical planner which uses actions in order of $P(A_j|S_i)/C(A_j)$ for optimizing the cost. Each server has a heart beat monitoring and a connection monitoring mechanism to sense its state (active, inactive, connected or killed). In addition, each
machine has a heart beat monitoring mechanism for sensing its state (up, down or killed).

The autonomic manager in this experiment uses a replanning mechanism for continuous planning, which means at each time-step (in case of a fault/failure) an action plan is generated and executed. By execution monitoring the autonomic manager can recognize that any action fails or the goal state does not reach because of a new unpredicted state. In that case another action plan should be generated by replanning.

Planning is implemented using Planner4J engine in ABLE and in ARL format [18]. In ARL a planner is defined by a rule-set containing predicates, the initial state, the goal state and planning rules. The following rule is a sample planning rule for restarting the database server (as depicted in Figure 3.2, we assumed that a machine may contain only a database server):

```plaintext
gerestartDB:
parameters (Object DB, Object Machine)
precondition (db(DB) and inactive(DB) and machine(Machine) and up(Machine) and installed(DB,Machine))
effect { connected(DB); }
```

Because the incorporated planner in ABLE still does not support metric planning, we use a predefined order of actions with their cost to generate the action plan. The actions are implemented in external Java methods.

### 3.3 Obtained Results

For the experiment, we inject a failure (failure of an application server) to one of the elements and observe the reaction of its autonomic manager. This experiment is performed three times by different configurations in the autonomic manager. Figure 3.3 illustrates three utility outcomes of experimenting on an autonomic element. The application server is failed at time-step 83. In the first experiment the autonomic manager realizes only self-optimizing discipline. Because the discipline cannot tune resources to compensate the failure, it requests a server provisioning. At time-step 88 a new machine adds to the element by cost 300$ and after this time step the element almost returns to stable state, of course without recovering the failed server.

In the second experiment, The autonomic manager realizes both self-healing and
optimizing disciplines without coordinating. Both disciplines plan a change in re-
response to the failure. Self-optimizing requests a server which adds at time-step 88, 
and self-healing executes a recursive restarting plan to recover the server. Figure 3.4 
depicts the phases for the failure recovery. At the first step, planner gives the plan 
of restarting the appServer1 for cheapest and quickest recovery. But it is not suc-
cessful because the goal state has not been reached and the webserver1 is now active 
may be because of a ripple effect of the appServer1 failure. For the next step web-
Server1 is restarted but it leads to the active state of both appServer1 and webServer1 
which is still not the goal state. The next action plan is restarting machine1 which 
finally changes the state of both servers to connected state. The server is recovers at 
time-step 88 without any action cost.

The third experiment shows the utility of an autonomic manager with coordina-
tion mechanism. The coordination mechanism is implemented by scheduling actions
in the execution phase. When the self-healing discipline wants to execute a restarting action, the provisioning action of self-optimizing is postponed because it is a costly hard adaptation action. For a time period (in this experiment 10 time-steps), if the self-healing cannot solve the problem with a cheaper action (probably soft adaptation), the coordination mechanism allows the self-optimizing to execute its action. In this experiment restarting can recover the server in 5 time-steps and as seen in the Figure 3.3 with less cost in comparison with the other cases manages the change in the element.

3.4 Conclusion and Future Works

Separation of concerns enables an autonomic element to manage its different concerns simultaneously using several closed-loop disciplines. However, because these disciplines manage one set of managed resources and must obey common policies, their behavior must be coordinated. This research shows that this coordination can be performed by analyzing the nature and impact of each discipline’s actions, as well as determining the phase at which the coordination will be occurred.

The action set of each discipline contains both primitive and non primitive actions in two categories of soft and hard adaptation. The soft adaptation actions are mostly related to nature of the discipline, so they can be often executed in parallel. But the hard adaptation actions may disturb the other discipline’s actions or cause a policy violation at the element level. The experiment performed in this paper, has focused on the second problem by implementing a policy-based coordination at execution time. For the first problem, the example is redeploying a server or service on a different machine for self-healing, which generates a disturbance in the resource distribution on that machine, and in consequence changing the state for the self-optimizing discipline.

The noteworthy point about autonomic disciplines is that they cannot be modelled as self-interested agents. In fact they try to achieve different aspects of the same set of objectives defined by policies. The problem here is not necessarily resolving conflicts between these closed loops, but it is selecting the optimal plan of action to meet the policies which generally include minimizing cost and time of actions.

For future works, besides of performing more experiments for various scenarios and actions, we want to work on planning-based coordination or a combination of planning-based and execution coordination. One challenge is that the planning part of
autonomic elements is still an open research area even for one discipline. AI planning methods, still, have not been used widely in this part because in some cases, a plan should be generated with *incomplete domain models*, which may lead to problems in executing the generated action set. So, the action plans may not be deterministic, and consequently coordinating of these plans is rather tougher.

One set of systems in which the proposed coordination mechanism can work better, are recursively recoverable (RR) systems [5]. For a system to be RR, it must consist of fine grain components that are independently recoverable, such that part of the system can be repaired without touching the rest. This requires components to be loosely coupled and be prepared to be denied service from other components that may be in the process of micro-rebooting. By this mean, the impact of actions, particularly for self-healing recovery, are more predictable, and the coordination would be more feasible.
Bibliography


