# Adaptive Action Selection in Autonomic Software Using Reinforcement Learning

Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, Ladan Tahvildari
Department of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada N2L 3G1
{mamouika, msalehie, smirarab, ltahvild}@uwaterloo.ca

## Abstract

*The planning process in autonomic software aims at selecting an action from a finite set of alternatives for adaptation. This is an abstruse problem due to the fact that software behaviour is usually very complex with numerous number of control variables. This research work focuses on proposing a planning process and specifically an action selection technique based on "Reinforcement Learning" (RL). We argue why, how, and when RL can be beneficial for an autonomic software system. The proposed approach is applied to a simulated model of a news web application. Evaluation results show that this approach can learn to select appropriate actions in a highly dynamic environment. Furthermore, we compare this approach with another technique from the literature, and the results suggest that it can achieve similar performance in spite of no expert involvement.*

## 1. Introduction

The main motivation for research in autonomic/self-adaptive software stems from increasing cost of management complexity due to changing context and variable quality requirements (i.e. performance). Adaptivity aims at giving a degree of variability to software in order to achieve quality goals, and navigating it during applying *adaptation actions*. Generally, the *adaptation mechanism* is decomposed to *monitoring*, *analyzing*, *planning* and *effecting* processes [5]. Most of the existing solutions realize the adaptation processes as an external *autonomic manager*. This manager handles the adaptation for an *autonomic element*, which is implementing the application logic. It also has knowledge about the autonomic element domain, objectives in form of self-* properties (quality goals), and articulated preferences by stakeholders.

The planning process in autonomic software aims at selecting from a finite set of alternatives. It is basically, an action selection problem which has to decide how to adapt the software to changes in the environment. In spite of the rich body of the research in autonomic computing domain, this particular problem has not been sufficiently addressed [8] . In this paper, our research focuses on proposing a planning process based on "reinforcement learning" (RL) to tackle the aforementioned problem. The RL-based process explores the action space and learns which actions are more suitable for the system state.

We applied our proposed approach to a simulated model of a news web application in Matlab Simulink. We also compared our results with another approach from the literature and find out that our approach can bring about similar performance in spite of no human expert involvement after its learning period.

## 2 Problem Statement

The problem addressed in this paper is related to the planning process in an autonomic manager. The planning process uses decision policies to choose among the adaptation actions whenever the analyzing process identifies the necessity of a change. The decision policy represents the adaptation logic in an autonomic system. The task of deciding on the optimal adaptation action to execute, given the system state and context is a planning problem [2]: "how do we find an optimal decision policy for performing adaptation actions, given a (hopefully complete and correct) model of the system state, a set of available adaptation actions, and a means of evaluating the result of adaptation actions?"

In the monitoring process, a set of attributes is measured. Suppose we measure $n$ attributes, $AT_1 \dots AT_n$, from the system such as *Load*, *throughput*, and *response time*. We formulate the measured values of those attributes as a tuple $< at_1, \dots, at_n >$, where $at_i$ denotes to the measured valued of the attribute $AT_i$.

In addition to the attributes, we have a set of goals. These goals represent the objectives of the autonomic software. More formally, we have a set of $k$ binary variables $G(s) = \{g_1(s) \dots g_k(s)\}$, each of which indicates whether

0-7695-3093-1/08 $25.00 © 2008 Crown Copyright
DOI 10.1109/ICAS.2008.35

a particular goal of the system is met for a given system state. This set of variables is accompanied with a rule-base. Each rule indicates whether an specific goal of the system is met, based on the current values of the goal variables. As an example, a rule can be "if the response time is more than a certain threshold, $g_{resp} = 1$, otherwise $g_{resp} = 0$".

A set of adaptation actions $AC = \{a_1 \dots a_m\}$ is also required. These actions try to change the state of the system to satisfy the goals. An example of an adaption action is to change type of the service that a particular component of the system is providing.

For satisfying objectives and goals under different conditions of attributes, it is required to decide which actions are the most appropriate at any time. This problem can be defined as: "*Given defined goals, attributes, and adaptation actions of an autonomous system; how to build a planning process which selects an optimal action to satisfy the goals in different conditions of attributes?*"

A desired planning process needs to have several features in autonomic software. The following list captures a view of such features:

• **Multi-objective decision-making**: There are often several self-* properties (e.g self-healing and self-optimizing) as objectives (goals), and each one with several concerns (e.g. cost and availability). The planning process needs to address the whole set of objectives and the potential conflicts. Hence, it requires to know the preferences of stakeholders. One way to represent the preferences is through a vector of constant weights $PV = \{pv_1, \dots pv_k\}$, assigned by domain experts. Each $PV$ value reflects the weight of an objective. Without loss of generality, we assume $\sum_{i=1}^{k} pv_i = 100$.

• **Reactive and deliberative decision-making**: The planning process is required to be reactive regarding low-level objectives, and deliberative in order to achieve high-level objectives. This requires the explicitly presented objectives.

• **Dynamism**: This feature can be decomposed to several characteristics, namely: i) being flexible and scalable for changing goals and policies, ii) being online without knowing what exactly happens in future, and iii) taking into account uncertainty both for events and outcome of actions.

• **Timely efficiency**: The decision-making in the planning process should be efficient. This is critical in cases which late actions could lead to crash or a non-controllable state.

# 3 Proposed Approach

Our approach to solve the stated problem is based on using reinforcement learning [13] to let the software learn the proper action in different situations. In this section, we first elaborate why we choose reinforcement learning, how we map our problem to RL, and when in the software development life cycle the RL algorithm can be used. Finally, we introduce our RL-based decision maker.

## 3.1 Why Using RL?

By reviewing the characteristics of a desired planning process in Section 2, one can easily identify the similarity between what reinforcement learning promises and what an ideal planning process is set to do. In both cases we have an action selection problem; states of the system are observed and we need to choose among a set of actions. Now, the question is how much the requirements specified in Section 2 are met by reinforcement learning algorithms?

• **Multi-objective decision making**: The RL algorithm learns based on a predefined reward function. If the rewards are multi-objective (e.g. using a weighted average mechanism), the result of learning will be multi-objective. Moreover, there are many ways to extend RL algorithms to have a better support of multi-objective decision making.

• **Reactive and deliberative decision-making**: The RL-based agents are normally designed to be deliberative, however they can have reactive behaviors if designed to be so. To achieve that, we need to define our state space such that once the goals are not satisfied, the agent start to get punishments. Therefore, the agent will learn to escape from those bad states to avoid receiving punishments and hence guides the system to goal-satisfying states.

• **Dynamism**: This property is what gives an edge to RL technique. RL is an on-line learning algorithm, meaning that the agent can learn to adjust (adapt) itself to even unseen situations. In case of changing the goals, the agent will re-learn its policies to adjust to new goals. As for the uncertainty, there are no assumption about the deterministic behavior in RL models. Especially, if the environment is modelled as a Markov Decision Process (MDP) [13] the uncertainty is taken into account.

• **Time-efficiency**: Once the training is done and the agent is learnt, then the decision making is a trivial task. For example, using greedy (or $\epsilon - Greedy$) algorithm of decision making one needs $O(m)$ where $m$ is the number of actions [13]. Even when the agent is learning, each individual action does not take a long time to be chosen because of the trial and error nature of the algorithm.

Therefore the RL algorithm seems to be a promising solution to our problem. But, now the question is that how to use RL for this purpose.

## 3.2 How to Use RL?

To use RL, we need to define basic elements of RL in accordance to our problem. How to map a problem to RL has a significant impact on the performance of the algorithm and should be done with care. The following sections describes the definitions of basic elements in the RL.

#### 3.2.1 States

The states should be defined in such a way that they represent all the desired conditions that the domain can have. Our knowledge of the target system is what we monitor from the domain. Therefore, we can define the states of the system based on the values of the monitored attributes. The most obvious way to do so is to define each possible tuple of attribute values, as one state of the system. However, this approach can lead to the problem of state explosion because attributes can have continuous domains. To deal with this problem, we choose to discretize the attribute values. This dramatically decreases the state space and hence increases scalability. We discretize the attributes into two values of "good" and "bad", using expert assigned thresholds on the attributes values. We define an state of the system as tuples of kind $< \hat{at}_1 \dots \hat{at}_n >$ where each $\hat{at}_i$ is a discrete variable for the corresponding attribute $at_i$.

#### 3.2.2 Actions

The easiest element to define is the set of actions because there exist a nice mapping between RL and action selection problem in autonomic software. The set of RL actions is defined to be same as the set of adaptation actions $AC$ which is available in the autonomic element and defined in the problem.

#### 3.2.3 Reward Function

The choice of the reward function is one of the most important factors in formulating a RL-based decision-making process. The reward value is what guides the agent to learn its behavior. The value of the reward function is conventionally defined based on whether we are reaching the goal node. However, because of the multi-objective and also reactive nature of autonomous software systems (as discussed in section 3.1) we do not have any specific goal state. Instead, we can define ways of measuring to what degree our current state satisfies the defined goals. As mentioned in section 2, we have a vector of preference weights for different goals. Using this vector we can define:

$$des(s) = \sum_{i=1}^{k} pv_i * g_i(s) \tag{1}$$

where $des(s)$ value measure how "desirable" a state $s$ is according to the defined goals. In addition, there is an argument that we need to take into the account the cost of each action. It is a common practice in RL to assign a small negative value to an action ($a$) which did not lead to goal so that the agent finds the smallest path to the goal. However, in this problem and according to the fact that we do not have any specific goal we assign a penalty to each action. Note that the system can chose not to take any action in which case no penalty is assigned. If we define:

$$p(a) = \begin{cases} 0, & \text{a = "no action";} \\ c, & \text{otherwise.} \end{cases} \tag{2}$$

where $c$ is a constant. Now we can define the reward function using ( 1), ( 2) as follows:

$$r(s, a) = des(s) - p(a) \tag{3}$$

#### 3.2.4 Learning Algorithm

There are many different variations of the RL algorithms. To choose the most effective one, we have to take into the account the characteristics of our action selection problem. For example, Monte Carlo methods are only applicable for episodic problems with explicit goals which is not the case in our problem. The autonomic software is supposed to run continuously and we do not have any termination state. However, one could consider using Monte Carlo method during the testing phase of the software, when we could define different test scenarios which have clear goal states. Another family of RL algorithms is Temporal Deference (TD). In TD, we do not need to have specific termination state and hence it is more fitting to our problem.

In the family of TD algorithms, there are three major categories: SARSA, Q-Learning, and Actor-critic [13]. Among these algorithms, the first two are more common (although not necessarily because of technical reasons [13]). SARSA and Q-Learning are generally very similar as they have proved to converge in specific situations. In different problems, the speed of convergence can be slightly different between the two algorithm but we can not foresee which of them is better for our problem.

### 3.3 When to Use RL?

Although RL techniques seem promising in the area of autonomic software systems, there is one principal problem. We know that RL algorithms have a trial-and-error nature. That means at the early stages of running, a RL-equipped agent can make some clearly wrong decisions. The result of these bad decisions will guide the agent toward the correct actions in similar situations. Therefore, when the agent is faced with a new environment, it needs a learning time.

It may be argued that such a phenomenon is not acceptable in most software systems. That is true in cases where we do not afford many wrong decisions at early stages. This problem can be dealt with using three approaches.

First solution would be to consider a learning time for autonomic software. This time can be a part of the testing phase of the system. That is when the system is built, we can run it and design test cases that cause the system to go to all (or most) possible states. During this phase, high values can be assigned to "learning factor" parameter and epsilon

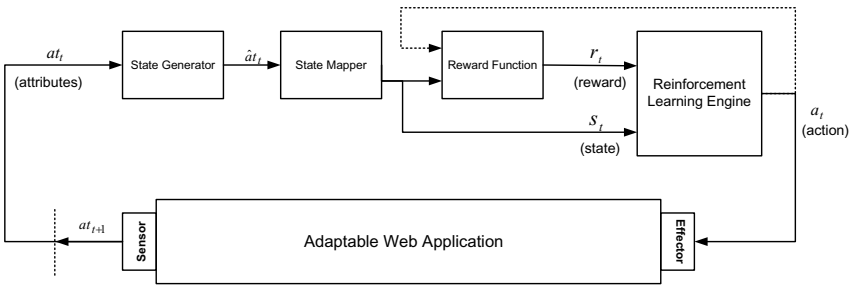


**Figure 1. Process Model of the Proposed RL-based Decision Maker**

value. Once the system converges and starts to show reasonable actions, we can deploy it and reduce the "learning factor" and epsilon value. Note that the system will continue to update its Q-Table values but slowly.

If the developers do not enjoy the luxury of a comprehensive testing phase (for example because of the market pressure), then we could use the expert knowledge. In this approach the experts (i.e. system admin or quality evaluation experts) should initialize the values of the Q-Table according to their expectations. This means they estimate intuitively what actions are suitable for what states. The system could be immediately deployed but the learning factor should be kept high (or moderate) so that the system can learn the optimal policies.

The last but not the least solution is using simulation. The new practice in software development is to have models of a software system (e.g. in Model-driven Architecture) or its behavior (e.g. performance model). Many techniques and languages have been proposed to model the architecture of softwares (e.g. Architecture Description Language) and especially model the behavior of autonomic/self-adaptive software [3]. If such models are available, we can use simulation for the learning phase. Using this approach instead of waiting for the actual system to be implemented, we can start the learning as soon as the models of the system are ready and hence do not impose any further testing time to the life cycle of the software.

### 3.4 RL-based Decision Maker

The RL-based decision maker consists of four major modules (depicted in Figure 1), namely:

**StateGenerator -** This module gets the value of the attributes and descretizes them according to predefined thresholds. The thresholds are carefully chosen such that they represent those of the goals.

**StateMapper -** This module takes $\hat{at}_i$ values from StateGenerator module and forms a single number representing the whole vector. This number represents the states of the system.

**Reward Function -** This module is responsible for computing our reward function. It first measures the $des(s)$ values as defined in Formula 1 and then computes the reward function based on Formula 3. Note that there is an implicit *performed action ($a_t$)* input to this module which is observable in dotted line in Figure 1.

**RL Engine -** This module has two responsibilities: i) it keeps and updates the Q-Table, ii) it selects the next action based on an $\epsilon - Greedy$ algorithm.

The whole system is a closed loop control system, and the aforementioned tasks are repetitively performed. Note that there is a latency between when the action is executed and when the results are observable. That is because the actions do not effect what is already in the queues. Therefore to see the result of an action, we need to wait until the existing requests in the queue are served. To do so, we use to estimate an expected value for waiting time as $w =$ "$size - of - queue$" $*$ "$average - waiting - time$" and to wait for w (simulation) seconds before choosing the next action.

## 4 Implementation

To be able to perform experiments on the proposed approach, we need two main components: *an autonomic element* equipped with monitoring and adaptation actions and *a RL-based manager*. The former constitutes our case study and the latter implements the planning process. Also note that in our case study, the action selection problem is covered only for the application-level adaptation. This means the actions are not low-level (e.g. in network or middleware level). These actions are mostly in the form of component recomposition to achieve application-level Quality of Service (QoS). QoS attributes are in close relation to self-* properties. Mukhija *et al.* discusses about this kind of adaptation with several examples [9].

### 4.1 Case Study

In [12], Salehie *et al.* have built a simulation model for a news web application and used it for experiments on de-

cision making for application level adaptation. We use this simulated model as a benchmark due to the arguments presented in section 3.1. We have slightly changed the model to comply with our needs.

This experimental model is a part of a news web application deployed on an application server. In this model, stakeholders are administrators and users which define their expectations as system's objectives. The model, in fact, is a performance model in form of network of queues (or queueing network [4]). It is an open network with infinite population, which means the model does not track one specific user behavior (series of think times and requests). The component model is a queue-server model which is implemented in SimEvents toolbox of Matlab Simulink. SimEvents provides discrete-event simulation (DES) model of computation which is suitable for a web application simulation. It is composed of three main web application components. The "front-end controller" component interprets requests (preprocess) and pass it to the "news retrieval" component to retrieve the content (i.e. text, image or video). The "view-rendering" component prepares the final HTML page for the end-user.

Each component has a priority queue based on service time for requests. The *input* set of each component is {service time, failure duration, failure probability, restart signal, and in-traffic}. The *Output set* is {average response time, number of requests served per second, component state, average load, and out-traffic}.

For traffic generation two parameters have to be taken into account, namely: *inter-arrival time* and *service time*. For the experiments in this paper, both parameters are generated by a multi-uniform probability distribution function. This means for different types of requests - for text, image and video contents - in each time interval, there is a different uniform probability of service time. The service time for "front-controller" is constant and does not depend on the type of requests. But, the randomly generated service time is used for "news retrieval" and proportionally for "view rendering" components.

As discussed, the system can perceive its state by monitoring a set of system attributes. Table 1 lists attributes of the news web application. Here, throughput is calculated based on the number of requests served per second.

| 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} |
| Component state ($at_4$) | {Active, Failed} |
| End2End Response time ($at_2$) | $\geq 0$ |
| User load (traffic)($at_3$) | $\geq 0$ |
| Throughput ($at_5$) | $\geq 0$ |

**Table 1. Attributes of News Web Application**

For the sake of simplicity of the design, we combine data type and quality in one attribute, $at_1$. If we denote Text as T, Normal and Small image mode as N and S, High and Low video resolutions as H and L, different feasible states for $at_1$ in the experiment is from the set $TNH, TNL, TN, TS, T$.

Table 2 lists the stakeholders' goals in this case study. The important point is to define the goal (what is expected) and its relation to attributes and actions in the system. Due to lack of space, we put only a general description of these goals, while more details are available in [11]. Summation of $pv_i$ has been set to 100. Max throughput goal will not be activated unless both throughput and load are high enough to pass their predefined values.

| Goals | Description |
|---|---|
| *Min MTTR* | Minimizing Mean-Time-To-Repair, $pv_1$=30 |
| *Best data type* | Having the best news data type (TNH or TNL),$pv_2$=10 |
| *Best quality* | Having the best news quality (TNH or TN) ,$pv_3$=15, |
| *Min response time* | Minimizing end-to-end response time ,$pv_4$=25 |
| *Max throughput* | Maximizing number of served requests per minute,$pv_5$=20 |

**Table 2. Goals of the News Web Application**

The adaptation actions implemented for this system consist of a set of five actions to alter the Type/Quality of the news content, and one action for restarting the components in case of failure. The first set, for example contains "switching to TNH" to give all the news contents in their best possible quality.

## 4.2 Experiment Setup

We setup the simulation model as below: we started with the Q-Table cells initialized to zero. For updating the Q-Table, we use the SARSA algorithm where the parameters are set as follows: $\gamma = 0.8$, $\alpha = 0.5$. As for the $\epsilon$, we start with a 0.1 value but gradually decrease it so that the system acts more conservatively. We run the simulation using the reward function introduced in Formula 3 for $c = 5$ and $c = 0$ (No action cost).

For each experiment, we ran the simulation model 200 times, each time for 200000 simulation steps. Note that the action time is set to 1000 second, meaning that in each run the manager can choose 200 actions.

## 4.3 Results Evaluation

This section reports the results of the performed experiments. The objectives of our experiments are to answer the following research questions:

- Can the proposed RL approach learn to behave better?
- How the performance of our approach is comparable to other approaches from literature?

### 4.3.1 Question 1: Can the algorithm learn?

Figure 2 depicts the average values of the reward function in each episode for $c = 1$ and $c = 0$. We have fitted a logarithmic curve on the observed points in order to show the

trend better. As seen in the figure, using these two reward functions the system is learned after a while, and seems it still continues to learn a better policy.

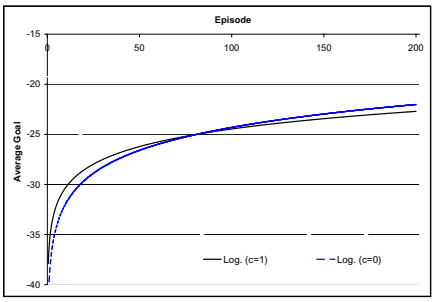


**Figure 2. Results of Learning ($c = 1$ and $c = 0$)**

This experiment also shows that both models are able to learn regardless of the action cost as time goes by. The model which uses action cost ($c = 1$) illustrates fewer and smoother sequence of actions by eliminating the action selection whenever it is possible. On the other hand, the model without action cost ($c = 0$) performs better by accumulating more rewards. As a result, we can argue that there is a trade off between service reliability and quality of service (more stable lower quality or less stable higher overall quality).

#### 4.3.2 Question 2: Is learning a good idea?

To understand whether the proposed learning-based approach is a suitable solution, we compare our results to those of two other techniques: GAAM (Goal-Action-Attribute Model) technique from [12] and a control technique where the actions are selected randomly.

The GAAM technique uses a voting mechanism for goals to choose the appropriate actions. First, it captures the knowledge of experts (e.g. system administrators) about relationships among actions and goals in form of numeric weight vectors. Then, at the operation time the technique allows each goal to vote for an action, and the action with higher votes (e.g. majority) will be selected. This model basically captures the relations between goals and attributes as well as goals and actions.

Table 3 shows some statistics about the performance of the three techniques, and Figure 3 depicts the boxplot diagram of the same result sets. In both places, "RL" corresponds to our approach, "GAAM" technique represents the mentioned approach from the literature, and "Random" is for random action selection. All the reported values are the average of 20 runs. For RL, however, we have learned the model for 200 iterations before the experiment and used the resultant Q-table. This process reflects the practical usage of the approach where the system should be learned in testing/tuning phase before the application deployment. The

| | Average | STDEV | Min | Max |
|---|---|---|---|---|
| **Random** | -32.47 | 9.50 | -45.78 | -14.67 |
| **GAAM** | -19.01 | 8.01 | -29.54 | 0.00 |
| **RL** | -23.39 | 9.71 | -43.47 | -14.99 |

**Table 3. Performance of Different Techniques (Reward Values for 20 Runs).**

performance index used for comparison is same as the introduced reward function in Furmola 3.

The obtained results suggest that both GAAM and RL techniques perform better than the random action selection in terms of the average. On the other hand, comparing RL and GAAM results in Table 3 and Figure 3 gives us an interesting point. While the average, maximum, and minimum performance of GAAM (as shown in the table) are better than those of RL, its median (as illustrated in the boxplot diagram) is not. To be able to interpret the results further we performed Kruskal-Wallis statistical test.

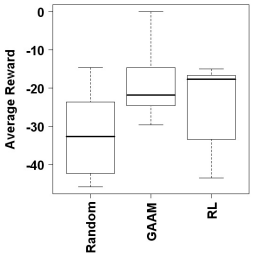


**Figure 3. Boxplot Diagram for All Techniques**

Table 4 lists the results of the test for each pair of techniques. In the significance level of 1% (i.e. $\alpha = 0.01$), both techniques are statistically better than random action selection, but they are not different from each other. The fact that RL technique has achieved a similar performance with GAAM is an accomplishment. RL is not tuned manually for this environment as opposed to GAAM for which the experts need to specify the parameters initially. The implication is that RL can be used for the error-prone and subjective task of adjusting to the goals of stakeholders.

| | GAAM vs RL | GAAM vs RAND | RL vs RAND |
|---|---|---|---|
| **chi-square** | 0.68 | 14.64 | 8.89 |
| **p-value** | 0.4113 | 0.00013 | 0.00287 |

**Table 4. Kruskal-Waillis Test Results.**

## 5 Related Works

Recently there has been an increased interest in using RL for software adaptation. In [14], Tesauro presents a hybrid approach that allows RL to bootstrap from existing management policies, which substantially reduces learning time and costliness, and he demonstrates hybrid RLs effectiveness in the context of a simple data-center prototype. Dowiling describes how RL can be used to coordinate the autonomic components in a distributed environment [2]. His

PhD thesis introduces K-Component, a framework to incorporate a decentralized set of agents which learn for adaptation. Also, Littman *et al.* have used RL to address self-healing properties in the domain of network systems [6]. They claim their system is able to learn to efficiently restore network connectivity after a failure.

On the other hand, the focused problem in this paper is similar to Action Selection Problem (ASP) in autonomous agents [7] and robotics especially behavior-based robotics [10]. Similarities can be enumerated as considering dynamism and uncertainty of the environment, as well as online decision-making. However, the problem in software is generally more complex than in robotics. This is mainly because software behavior is not obeying physical rules; in addition of having more control variables and disturbances as well as more complexity.

Several ideas in behavior-based robotics seems promising for autonomic software. Majority of existing autonomic managers use a model-based approach for adaptation. These systems normally utilize the sensor fusion model to capture events and update the model. But, the idea of behavior-based robotics is to use distributed specific task-achieving modules, called behaviors, and to apply command fusion instead of sensor fusion, as in subsumption architecture [1]. By this mean, there is no need to develop, maintain and extend a coherent monolithic model for the autonomic element and its context. More related to our work, Distributed Architecture for Mobile Navigation (DAMN) uses a voting mechanism for command fusion regarding the safety behaviors for turn and speed of the mobile robot [10]. Recently Salehie *et. al.* used voting mechanism for decision making in autonomic systems [12].

## 6 Conclusion and Future Work

In this paper, we explained how reinforcement learning can be used in decision making of an autonomic software. Our approach is proposed to tackle the action selection problem with the aid of RL and SARSA algorithm. As a proof of concept, a series of experiments have been conducted using Simulink to model the behavior of a news web application and define a set of adaptation actions. Then, the RL algorithm have been implemented and used to learn the best possible action in each state. We compared our results with one previous work in our group [12], and realized that although both techniques outperform the control technique (random), they are not statistically different. The other interesting observation is that neither of the techniques achieve high performance. This implies that further research is required on the problem of action selection in adaptive systems.

There are many ways to extend this work in future. First and foremost, the work can be extended to an actual implementation in a real life application environment. More experiments with other benchmarks could be performed and the technique could be compared to other works from literature. In terms of improving the performance of the model there are also many opportunities. One could use other methods of reinforcement learning such as Monte Carlo or Q-Learning; the parameters of the algorithm could be adjusted; better mechanisms of multi-objective action selection could be examined and so on. In short, we think the reinforcement learning mechanism is promising for the problem of action selection in an autonomic software system.

## References


[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[2] J. Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, Department of Computer Science, Trinity College Dublin, Oct. 2004.

[3] D. Gracanin, S. A. Bohner, and M. Hinchey. Towards a model-driven architecture for autonomic systems. In *IEEE Int. Conf. on the Engineering of Computer-Based Systems*, pages 500–505, 2004.

[4] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, 1991.

[5] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[6] M. L. Littman, N. Ravi, E. Fenson, and R. Howard. Reinforcement learning for autonomic network repair. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 284–285, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[7] P. Maes. Situated agents can have goals. *Robotics and Autonomous Sys.*, 6:49–70, 1990.

[8] P. K. McKinley, M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, pages 56–64, 2004.

[9] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of 18th Int. Conf. on Architecture of Computing Systems (ARCS)*, pages 124–138, 2005.

[10] J. Rosenblatt. DAMN: a distributed architecture for mobile navigation. *Journal Exp. Theory Artificial Intelligence*, 9(2-3):339–360, 1997.

[11] M. Salehie and L. Tahvildari. Action selection in self-adaptive software using social choice theory. Technical report, University of Waterloo, 2007. UW-ECE-2007-17.

[12] M. Salehie and L. Tahvildari. A weighted voting mechanism for action selection problem in self-adaptive software. In *First IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems*, pages 328–331, 2007.

[13] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[14] G. Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30, 2007.