# Reinforcement Learning-Based Dynamic Adaptation Planning Method for Architecture-based Self-Managed Software*

Dongsun Kim and Sooyong Park
Department of Computer Science and Engineering, Sogang University
Shinsoo-dong, Mapo-Gu, Seoul, Korea
{darkrsw,sypark}@sogang.ac.kr

## Abstract

*Recently, software systems face dynamically changing environments, and the users of the systems provide changing requirements at run-time. Self-management is emerging to deal with these problems. One of the key issues to achieve self-management is planning for selecting appropriate structure or behavior of self-managed software systems. There are two types of planning in self-management: off-line and on-line planning. Recent discussion has focused on off-line planning which provides static relationships between environmental changes and software configurations. In on-line planning, a software system can autonomously derive mappings between environmental changes and software configurations by learning its dynamic environment and using its prior experience. In this paper, we propose a reinforcement learning-based approach to on-line planning in architecture-based self-management. This approach enables a software system to improve its behavior by learning the results of its behavior and by dynamically changing its plans based on the learning in the presence of environmental changes. The paper presents a case study to illustrate the approach and its result shows that reinforcement learning-based on-line planning is effective for architecture-based self-management.*

## 1. Introduction

As software systems face dynamically changing environments and have to cope with various requirements at run-time, they need the ability to adapt to the environments and new requirements[12, 1]. Increasing demands for more adaptive software introduced the concept of architecture-based self-management[11] in which the software system can dynamically change its architectural configuration without human intervention. To achieve architecture-based self-management, lots of challenges should be addressed such as identifying adaptable requirements, modeling adaptable software architecture, collecting environmental data, planning software architecture, and reconfiguring software dynamically. Among the challenges, planning software architecture is one the key issues to implement autonomy in architecture-based self-management.

Planning in self-management indicates the ability to make a decision which represents behavioral or structural changes to software systems. In other words, planning is an activity in which a software system maps appropriate structure or behavior in the presence of a new situation of the environment or new user requirements. There are two different types of planning in self-management[11]: off-line and on-line planning. Off-line planning represents that mappings between situations (or states) and possible software configurations are made by software construction process with human intervention. For example, mapping strategies to invariants described in [5] is one of the off-line planning because those are specified by an architectural description language by construction process[1].

On-line planning represents that a system can dynamically and autonomously make a decision about mappings between situations and configurations. In on-line planning, generally, the software system carries out its task based on the current configuration and accumulates its experience (quantitative results of the current configuration to the current situation), and then learns the effectiveness of the configuration based on the previous experience. Using learning data, the system can determine appropriate (generally best-so-far) mappings in the presence of situation changes. By repeating this process (execution, accumulation, learning, and decision making), the system can identify better mappings (i.e. improving plans by repeated learning).

*This research was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Knowledge Economy (MKE).

[1]These plans can be injected at run-time (i.e. late binding), but those plans are eventually identified by analysis and design process with human intervention

Previous approaches so far to planning in self-management focus on an off-line planning process where adaptation plans are previous designed or predefined at construction time[11] and self-contained. Off-line planning is effective if developers can anticipate every possible mapping between situations and configurations. However, it is difficult to anticipate interrelationships between situations and actions in actual execution before the deployment of software. On the other hand, an on-line planning process enables a software system to generate plans adaptively by evaluating and learning its behavior, and exploiting the result at run-time in the presence of uncertain environments. Thus, if we can provide on-line planning capabilities to a system, the system can more autonomously adapt its configuration to the environment.

In this paper, we propose an reinforcement learning-based on-line planning approach to architecture-based self-managed systems. Specifically, our approach applies Q-learning[23]. To support reinforcement learning-based on-line planning in architecture-based software, we present several elements. Those are (a) representations, which are discovered from software requirements denoted by goals and scenarios, (b) fitness functions to evaluate the behavior of a system, (c) operators to facilitate on-line planning, and (d) a process to apply the previous three elements to actual run-time execution.

The paper is organized as follows. Section 2 describes planning approaches to architecture-based self-management. Section 3 presents our approach which consists of representation (Section 3.1), fitness (Section 3.2), operators (Section 3.3), and an on-line evolution process (Section 3.4) to achieve on-line planning in self-management. Section 4 describes a case study conducted to verify the effectiveness of our approach. Section 5 summarizes the contributions of the paper.

## 2. Planning Approaches in Architecture-based Self-Management

Planning in software systems can be represented as an interaction between an agent and an environment, as depicted in Figure 1[18]. This model shows the ideal interaction in which the agent can immediately monitor states of the environment and observe rewards from the environment after the agent gives an action to the environment. In the interaction model, a software system plays a role as an agent. The system monitors the current state ($s_t$) of the environment (where the software system operates) and observes the current reward ($r_t$) which is a special numerical value from the environment. Based on the state and observed reward, the software system (agent) selects and takes an action ($a_t$). The taken action $a_t$ influences the environment. After applying the action, the environment transitions into

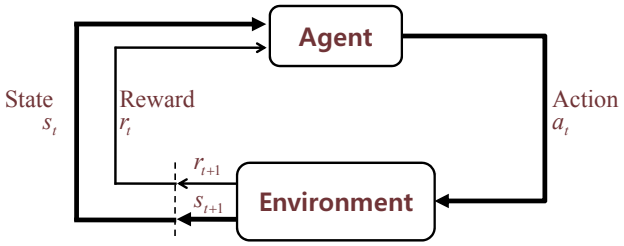


**Figure 1. The agent-environment interaction in reinforcement learning.**

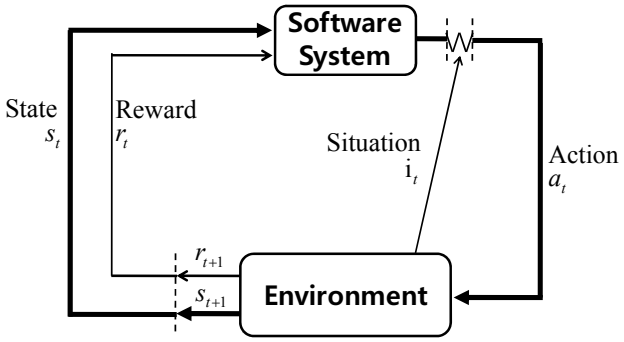


**Figure 2. The software system-environment interaction in self-management.**

the next state ($s_{t+1}$) and produces a reward. The software system(agent) selects a new action ($a_{t+1}$) based on the current state ($s_{t+1}$) and reward ($r_{t+1}$). This continual interaction is the basis realizing the autonomy of self-management.

However, the ideal model is not appropriate for actual software systems. State transition and reward observation can be delayed because it takes time until the given action affects the environment. If this timing issue is ignored, the system cannot monitor the state transition properly nor observe an appropriate reward. Hence, the systems need a modified interaction model for self-managed software as depicted in Figure 2. In the model, the difference from the model shown in Figure 1, is the existence of '**situation** $i_t$'. This allows the system to know when it must monitor the state and reward, and also when it must take the action corresponding to the state. In other words, it triggers the adaptation of self-managed systems.

One remaining issue is how this interaction model can be implemented in architecture-based self-management. Suppose that we can observe a finite set of situations that a software system can encounters and let $S$ be the set. Let $C$ be a finite set of architectural configurations that the system can take. When a situation ($s \in S$) occurs, if the system can always choose the best architectural configuration ($c \in C$, like to actions in Figure 2) among possible configurations

of the system, then we can say the system have a set of the best plans to the environment. This is can be formulated by:

$$P : S \rightarrow C \tag{1}$$

$$V(s_i, P(s_i)) = \max_{\forall c_j \in C} \{V(s_i, c_j)\}, \forall s_i \in S \tag{2}$$

where $P$ in equation (1) is a planner which maps each situation ($s \in S$) into an appropriate configuration ($c \in C$) during the system's execution. $V$ in equation (2) is a value function which calculates the value of a configuration based on the given situation $s_i \in S$. Equation (2) indicates if the configuration that the planner $P$ chosen is the best configuration, $P$ is the best planner in the environment. One of the primary goals of researchers in self-management is to find an optimal planner which can autonomously make plans even in dynamically changing environments. The rest of this section discusses two planning approaches to find an optimal planner in architecture-based self-management.

### 2.1. Off-line Planning

In architecture-based self-management, off-line planning means that decisions which define the relationship between situations and plans (i.e. configurations) are made prior to run-time. In other words, whenever a system encounters a specific situation $s_i$ from an environment each time, the system selects and executes exactly one identical configuration $c_i$. Solutions so far to self-managed systems focus on off-line planning[11]. For example, plans are made by a system administrator through console or hard-coded by a developer[14], architectural adaptation is described by mapping invariants and strategies in ADL description[5], or architectural changes are triggered by utility functions[4].

These off-line approaches can be effective if developers can identify well-defined goals, states, configurations, rewards, and their relationships along with test environments that exactly illustrate the characteristics of the actual operating environments before deployment time. However, it is very difficult to identify them due to the nature of planning[9]. However, On-line planning, which gives more effective autonomy in architecture-based self-management, presents an alternative to overcome the limitation of off-line planning.

### 2.2. On-line Planning

On-line planning in architecture-based self-management represents that a software system can autonomously choose a configuration with respect to the current situation that the system encounters. Generally, an on-line planning process has three major steps: selection, evaluation, accumulation[9, 10]. In the selection step, the system autonomously chooses a configuration which is suitable for the current situation. Generally, the configuration is chosen by the greedy selection strategy in which the best-so-far configuration is chosen. However, this strategy may lead to the problem of local optima. Hence, the system must adjust its strategy between *exploitation* and *exploration*. Exploitation represents that the system chooses a greedy configuration while exploration means that the system intentionally chooses an suboptimal or random configuration to find a better solution. It is important to adjust the ratio of two strategies because it determines the planning performance.

In the evaluation step, the system must estimate the effectiveness of the configuration which is taken in the selection step. The key issue in the evaluation step is to define the way to determine the numerical values which represent the reward of the configuration because the numerical representation enables the accumulation and comparison of the rewards. In the accumulation step, the system stores the numerical values identified in the evaluation step. The system must adjust the accumulation ratio between already accumulated knowledge and newly incoming experience. If the system uses accumulated knowledge too much in the accumulation step, it may slow down accumulation speed. On the other hand, if the system uses new experience too much in the step, it may cause ineffective accumulation. With accumulated knowledge, the on-line planner can select an appropriate action in the next selection step.

With these steps, self-managed systems can take advantage of on-line planning in the presence of dynamically changing environments. The next section describes how on-line planning can be applied to actual systems in detail.

## 3. Q-learning-based Self-Management

This section presents an approach to designing architecture-based self-managed software by applying on-line planning based on Q-learning. Generally, we need to consider three elements to apply metaheuristics such as reinforcement learning; those elements are 'representation', 'fitness', and 'operators'[7]. In reinforcement learning, the representations of states and actions (=configurations) are critical to shaping the nature of the search problem. The fitness function is used to determine which solution is better. The operators enable a system to determine neighbor solutions which can accelerate the learning process. Hence, the proposed approach provides state and action representations that the system can exploit, fitness function design for the system to determine better solutions, and operators to manipulate solutions. In addition to these three elements, the approach provides an on-line evaluation process which describes the control loop of the system at run-time.

### 3.1. Representation

The representations of states and actions are crucial for designing self-managed software using on-line planning because it defines the problem space and the solution space of the system. Their representations are usually depicted in numerical representation or symbolic code. In the former case, it is more suitable for neural networks[16] and support vector machines[3, 22]. A numerical representation is not appropriate for shaping the problem and solution space of reinforcement learning which uses discrete information, because it may cause state explosion which leads to the increase of training time of the system. Hence, the approach uses symbolic code to represent the problem(state) and solution(action) space.

Our approach provides a goal and scenario-based discovery process for more systematic state and action discovery. Goal[20, 13] and scenario-based approaches[21, 17] are widely used for the elicitation of software requirements. Also, goal and scenario discovery have been studied[15]. Goals denote the objectives that a system must achieve, for example, 'maintain high throughput in a communication system' or 'minimize response time'. Scenarios represent a sequence of events to achieve a specific goal, for example, 'the system is given user input and store them into databases'. Goals are structured hierarchically to represent the relationship of super and subgoals. A subgoal represents more specific objectives to achieve its supergoal, eventually it supports the root goal of the goal hierarchy. A goal has scenarios and they are used for discovering new subgoals by describing the related goal in detail.

The proposed approach exploits goals and scenarios to discover states and actions. Once goal and scenario structure is organized, they can be mapped to states and actions by reforming them. First, scenarios must be reformed into the pair of 'condition → behavior' or 'stimulus → reaction', e.g. 'when the battery of the system is low(condition or stimulus), turn off the additional backup storage(behavior or reaction)'. This reforming is depicted in Table 1. In this table, the discovered goals are listed in sequence (goals will be used to discover fitness functions as described in Section 3.2). The scenarios of a specific goal are listed by the goal. Each scenario is reformed into two additional columns: Condition(stimulus) and Behavior(reaction).

States are identified from the scenarios. Conditions in the scenarios can be candidates of states. A condition represents a possible state of the system, e.g. 'the system's battery is low' implies 'low-battery' or 'the system's battery is full' implies 'full-battery'. These conditions depict physical states of the system, i.e. what the environment has or shows. Thus, a group of conditions represents a dimension(type) of states, for example, 'battery-level' is a dimension of state information and it can have value, either 'low-battery' or 'full-battery'. While this information represents a long-term state of the environment, a situation represents a transient change of the environment, e.g. 'hit by wall' in an autonomous mobile robot system. Situations are triggers to begin the adaptation of the system. Situations can also be identified from condition information. The condition, which describes a transient event such as 'when the system is hit by bullet', can be transformed into a situation.

One more element which must be considered when talking about states is the current software architecture of the system. Although the architecture is not part of the environment, it can influence the planning of the system, for example, it can be used to find an admissible action set. An architecture can be denoted by symbolic notation which consists of a vector of components and connects e.g. (component:A, component:B, component:C, connector:A-B, connector:B-C). To simplify them, it can be transformed into a unique architecture identifier, e.g. arch:1a.

These three pieces of information(situation, long-term state, architecture) compose the state information. An example of elements of state information is depicted in Table 2. The state information of the system can be denoted in a vector form such as `(situation, long-term state, architecture information)`. For example, (hit-by-wall, near, low, arch:1a) and (hit-by-wall, far, full, arch:1a), which are combinations of elements from Table 2.

Actions can be identified by extracting behavior or reaction from scenarios. As depicted in Table 1, a set of behavior(reactions) is identified in a pair of conditions(or stimuli). If these pairs between conditions and reactions are fixed before deployment time, it can be considered off-line planning, i.e. static plans. The goal of this approach is on-line planning in self-management, the set of actions should be discovered separately. Similar to state information, actions can be identified by discovering action elements and grouping the elements into a type. For example, first, identify action elements such as 'stop moving' or 'enhance precision'. Then, group the elements which have the same type.

Examples of action elements and its type are shown in Table 2. With these pieces of information, each action can be represented by a vector form such as (precise maneuver, rich) or (stop, text-only) where the first dimension is 'Movement' and the second one is 'GUI'. An action element such as 'rich' in GUI or 'stop' in Movement, implies architectural changes which include adding, removing, replacing a component, and changing the topology of an architecture. Hence, each action must be mapped with a set of architectural changes. For example, the action '(precise maneuver, rich)' can be mapped with a sequence of '`[add:(visionbased_localizer), connect: (visionbased_localizer)-(pathplanner), replace: (normal_gui)-by-(rich_gui)]`'

**Table 1. Reformed goals and scenarios**

| Goal | Scenario | Condition(stimulus) | Behavior(reaction) |
|---|---|---|---|
| Goal 1 Maximize system availability | Sc. 1-1 when the battery of the system is low , turn off the additional backup storage | Cond. 1-1 the battery of the system is low | Beh. 1-1 turn off the additional backup storage |
| | Sc. 1-2 ... | Cond. 1-2 ... | Beh. 1-2 ... |
| Goal 2 ... | Sc. 2-1 ... | Cond. 2-1 ... | Beh. 2-1 ... |
| ... | ... | ... | ... |
| Goal n.m ... | Sc. n.m-1 ... | Cond. n.m-1 ... | Beh. n.m-1 ... |

**Table 2. An example of state and action information**

| Situation | State(long-term) | | Action | |
|---|---|---|---|---|
| | Type | Range | Type | Range |
| hit-by-wall | distance | {near, far} | Movement | {precise maneuver, stop, quick maneuver} |
| hit-by-user | battery | {low, mid, full} | GUI | {rich, normal, text-only} |

where `(...)` indicates a component name.

The discovery process shown in this section identifies states and actions by extracting data elements from goals and scenarios. Because goals and scenarios directly represent the objectives and user experiences, and also they show how the system influences the environment, the number of states and actions can be limited. In other words, it helps the system prevent a state explosion.

### 3.2. Fitness

The fitness function of the system is crucial because it represents the reward of the action that the system chooses. The function provides a criterion to evaluate the system's behavior. Specifically, in autonomous systems, it is necessary to know which of two actions is the better according to the current state. Hence, rewards that the fitness function generates are usually represented by numerical numbers which can be compared to each other. The fitness function is usually application specific, so it is important to discover the function from systems requirements. Our approach exploits the goal and scenario structure discovered in Section 3.1. In particular, goals are the source of fitness discovery.

Generally, goals, especially higher ones including the root goal, are too abstract to define numerical functions. Thus, it is necessary to find an appropriate goal level to define the fitness function. It is difficult to define universal rules for choosing appropriate goals which describes numerical functions of the system, but it is possible to propose a systematic process to identify the functions. The following shows the process to define the fitness function. This process must be manually conducted by system developers.

1. From the root goal, search goals which can be numerically evaluated by a top-down search strategy.

2. If an appropriate goal is found, define a function that represents the goal and mark all subgoals of the goal(i.e. subtree). Then, stop the search of the subtree.

3. Repeat the search until all leaf nodes are marked.

More than one of the fitness functions can be identified by the discovery process. In this case, it is necessary to integrate the functions into one fitness function. The following equation depicts the integrated fitness function:

$$r_t = f(t) = \sum_i w_i f_i(t) \tag{3}$$

where $r_t$ is the reward at time $t$, $f(t)$ is the (integrated) fitness function, $w_i$ is the weight of the $i$-th function, $f_i(t)$ is the $i$-th function of $t$, and $\sum_i |w_i| = 1$. Equation (3) assumes that the objective of the system is to maximize the values of all functions $f_i(t)$. To minimize a certain value, multiply $-1$ to the weight value of the function $f_i(t)$. Every function $f_i(t)$ corresponds to the observed data of the system at run-time. For example, the observed network latency $100ms$ at time $t$ is transformed into 10 by the function $f_i(t)$ and multiplied by -1 because it should be minimized.

### 3.3. Operators

Different metaheuristics use different operators. For example, genetic algorithms use crossover and mutation. The reason why algorithms use operators is to accelerate searching better solutions. In reinforcement learning, operators are used to reduce the search space, i.e. the number of actions. Operator $A(s)$ specifies an admissible action set of the observed state $s$. For example, when a mobile robot collides with the wall, actions, which are related to motion control are more admissible than those of arm manipulation. This operator is crucial because it can reduce the training time of the system.

### 3.4. On-line Evolution Process

With three elements discussed through Section 3.1∼3.3, the system can apply on-line planning based on Q-learning. Q-learning[23] is one of temporal-difference (TD) learning which is a combination of Monte Carlo ideas and dynamic programming ideas[18]. Specifically, Q-learning is an off-policy TD control algorithm. The reason why we choose Q-learning is its modeless characteristics. In contrast to model-based planners[2, 19], model-free TD methods can learn directly from raw experience without a model of the environment. Also, TD learning assumes that it updates observed data based in part on prior knowledge, without waiting for a final outcome. TD methods also assume that events of the environment are recurring. These satisfy the properties of on-line planning described in section 2. This section presents the way to exploit those elements in the on-line evolution process. The process consists of five phases: detection, planning, execution, evaluation, and learning phases.

#### 3.4.1 Detection Phase

In the detection phase, the system monitors the current state of the environment where the system operates. When detecting states, the system uses the notation presented in Section 3.1. Continual detection may cause performance degradation. Thus, it is crucial to monitor the change which actually triggers the adaptation of the system. Architecture information is not suitable for the purpose because it does not reflect environmental changes and it is changed only by the system. Also long-term states are not feasible because they are only the result of environmental changes instead of the cause. Situations can be appropriate triggers because they describe moments that the system needs adaptation. If a situation is detected, then the system observes long-term states and the current architecture of the system, and denotes them into the representation presented in 3.1. These data are passed to the next phase: the planning phase.

#### 3.4.2 Planning Phase

Using the state identified by the detection phase, the system chooses an action to adapt itself to the state. At this time, the system uses an action selection strategy. In general, Q-learning uses '$\varepsilon$-greedy' selection strategy as an off-policy strategy to choose an action from the admissible action set $A(s)$ of the current state $s$. The strategy is a stochastic process in which the system exploits prior knowledge or explores a new action. This is controlled by a value $\varepsilon$ determined by the developer, where $0 \leq \varepsilon < 1$. In this phase, the system generates a random number $r$. If $r < \varepsilon$, the system chooses an action randomly from the admissible action set. Otherwise($r > \varepsilon$), it chooses the best-so-far action by comparing the value of each action accumulated in the learning phase(see Section 3.4.5). In this manner, the stochastic strategy prevents the system from falling local optima. The chosen action is used by the execution phase.

#### 3.4.3 Execution Phase

This phase applies the action, which is chosen in the previous phase(planning phase), to the system. As the action describes architectural changes such as adding, removing, replacing components, and reconfiguring architectural topology, the system must have architecture manipulation facilities, i.e. dynamic architecture. Also, it must support the way to make a quiescent state[6]. The quiescent state[2] in architectures represents that the system(or part of the system which are relevant to the changes described by the action) is safe to execute architectural reconfiguration. If the action is not executed in the quiescent state, it may cause unsafe changes which can lead to a malfunction.

Once reconfiguration is done, the system carries out its own functionalities by using its reconfigured architecture. The system keeps executiing until it encounters a new situation or it terminates. Although this may overlap with the detection phase, the detection in this phase indicates a phase transfer from this phase to the next phase(the detection phase of course uses this detected situation).

#### 3.4.4 Evaluation Phase

After the execution phase, the system must evaluate its previous execution by observing a reward from the environment. As mentioned in section 3.2, the system continuously observes values previously defined by the fitness function. These values will be used for calculating the reward of the action taken. The values must be retrieved immediately. Otherwise, in other words, if the system is delayed in retrieving reward values, it may influence the performance or

[2]The term 'state' is different from the term mentioned in Section 3.1. In this context, the state indicates the execution state of the software architecture

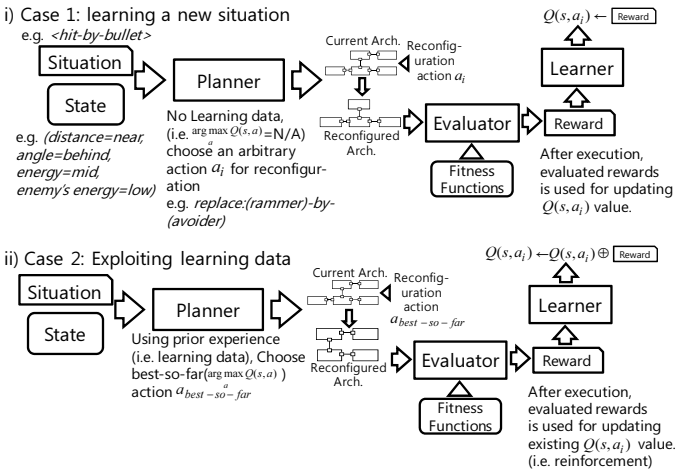


**Figure 3. Examples of applying the on-line evolution process.**

quality of the learning phase. Hence, the system must have facilities to observe the values immediately. The observed values are passed to the next phase: the learning phase.

#### 3.4.5 Learning Phase

In this phase, the system accumulates the experiences which represent the previous execution described in section 3.4.3, by using the reward observed in the evaluation phase. This phase directly uses Q-learning. The key activity of Q-learning is updating Q-values. This update process is depicted in equation (4),

$$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t)+\alpha[r_{t+1}+\gamma max_a Q(s_{t+1}, a)] \quad (4)$$

where $0 \leq \alpha \leq 1$ and $0 < \gamma \leq 1$. $\alpha$ is a constant step-size parameter and $\gamma$ is a discount factor. This update equation accumulates the result of the previous execution based on the current state, the chosen action, and the observed reward. This technique composites the current experience upon the previous experience. The term $(1-\alpha)Q(s_t, a_t)$ represents the value the system already knows and its weight for accumulation where $\alpha$ controls the weight. On the other hand, the term $\alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a)]$ represents the current reward and the expectation of the greedy action of the newly observed state where $\alpha$ controls the term's weight and $\gamma$ controls the weight of the greedy action. In this manner, the system can accumulate its experience by updating $Q(s_t, a_t) = v$ where $s_t$ is the detected state, $a_t$ is the action taken by the system at $s_t$, and $v$ is the value of the action on $s_t$. This knowledge will be used in the planning phase(see section 3.4.2) to choose the best-so-far action.

Examples that a system uses the above process are shown in Figure 3. i): When a system encounters a new situation, the planner in the system chooses an arbitrary reconfiguration action because there is no learning data about the new situation. Then, The system reconfigures its architecture based on the chosen action. During the system executes its functionality, the evaluator the execution based on the predefined fitness function and produces rewards. The learner accumulates the rewards with respect to the prior situation (planning model construction). ii): When the system encounters an already experienced situation, the planner exploits the best-so-far action based on learning data (exploitation based on the previously constructed planning model). After the execution of the reconfigured architecture based on the best-so-far action, the evaluator produces rewards. The rewards are used to reinforce current Q-values (model reinforcement). When the environment changes, the current Q-values are not appropriate the changed environment. Hence, the system must try actions with respect to situations and accumulate rewards and reinforce Q-values to adapt the changed environment (model updating).

With these facilities discussed through section 3, the approach supports software developers to construct self-managed software with on-line planning based on Q-learning. The next section shows the result of a case study which applies the approach to a robot simulator.

## 4. Case Study

This section reports on a case study which applies our approach to an autonomous system. The environment in this case study is Robocode[8] which is a robot battle simulator. Robocode provides a rectangular battle field where user-programmed robots can fight each other. It also provides APIs(Application Programming Interfaces) for robot design. A robot consists of a radar, gun, and body. The radar is responsible for locating enemy robots on the battlefield. It indicates the location of an enemy robot by providing a distance and a relative angle. The gun fires a bullet to a enemy robot. It can control the power of the bullet, but a more powerful bullet may cause a longer cooling time. The body contains a radar, a gun, and two wheels. By giving speed values to the wheels, the robot can move around in the battlefield. Basically, Robocode only provides templates to build a robot. Robot developers should construct software systems for their robots. In general, developers make firing, targeting, and maneuvering functionalities to build a robot.

The reason why we chose Robocode is that it can provide a dynamically changing environment and enough uncertainty, as well as being good for testing self-managed software with on-line planning. In particular, it is hard to anticipate the behavior of an enemy robot prior to run-time. Also, several communities provide diverse strategies for fir-

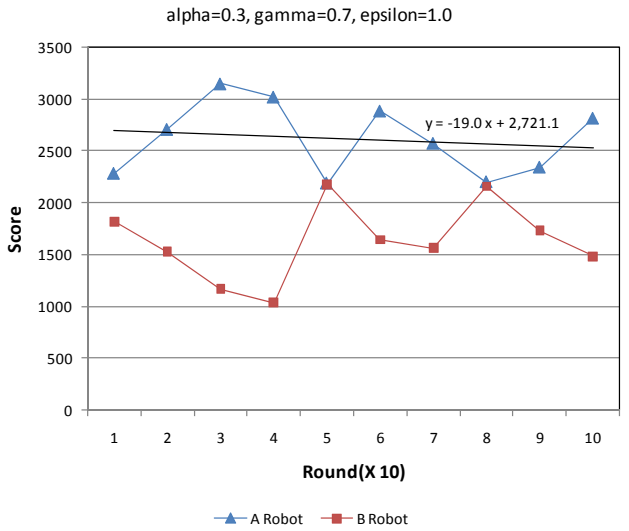


**Figure 4. Scores of 'A Robot' which has parameters: $\alpha = 0.3, \gamma = 0.7, \varepsilon = 1.0$. $\varepsilon = 1.0$ indicates the robot always randomly choose an action in the presence of any situation. (i.e. exploration only)**

ing, targeting, and maneuvering. These offer opportunities to try several reconfiguration with respect to various situations.

The rest of this section presents a robot design which applies our approach. The design includes representations, fitness functions, and operators for the robot. Also this section presents the result of the evaluation which shows the effectiveness of our approach.

### 4.1. Robot Implementation

To verify the effectiveness of the proposed approach, we implemented a robot with dynamic architecture. This robot can try lots of firing, targeting, and maneuvering strategies by changing its software architecture. Also, we identified situations, states, actions, fitness functions, and operators based on our approach. The detailed description of robot implementation is provided in Appendix A.

### 4.2. Evaluation

This section shows the effectiveness of the approach by presenting the result of robot battles in Robocode. Two experiments are conducted in this research. Learning data were initialized at the beginning of every experiment except the second experiment.

The first experiment was conducted to compare the approach with random action selection (i.e. random adaptation) which is one of generic evaluation criteria for metaheuristics[7] and to verify whether our planning approach can generates plans without an initial model of the environment. We chose a robot named 'AntiGravity 1.0' as the enemy. This robot has an anti-gravity algorithm which enables the robot to move as it has a reverse gravity engine. In other words, the algorithm supposes that the wall of the battlefield and other robots have gravity. Thus, the robot keeps enough distance between the wall[3] and other robots. The robot ('AntiGravity 1.0') is known for its good performance in battles with many other robots, in Robocode communities. For convenience, the robot described in Section 4.1 will be denoted by 'A Robot' and 'AntiGravity 1.0' will be denoted by 'B Robot'.

The result of the battle, in which 'A Robot' with random action selection ($\varepsilon = 1.0$ indicates the robot chooses actions randomly whenever the system monitors any situation. This means the robot repeats case i) in Figure 3 without exploiting prior experience) fought with 'B Robot', is shown in Figure 4. We carried out one hundred rounds, and observed scores after every ten rounds. Each score consists of a survival bonus (70 points), and bullet damage points. Scores satisfy the goals of the robots, 'Eliminate the enemy' and 'Maximize my energy', by surviving in the battle field (which offers a survival bonus) and satisfy 'Minimize the enemy's energy' by hitting the enemy with bullets (which offers bullet damage points).

The result shown in Figure 4 indicates that the robot with random action selection does not adapt to the environment (i.e. the enemy robot) even though it slightly outperforms 'B Robot'. The regression line which represents the performance of 'A robot' in Figure 4 does not increase because the robot cannot exploit prior experience to find better configurations (i.e. no planning model construction). Hence, the robot cannot make effective plans. This indicates a planner is needed to effectively adapt to a modeless environment.

On the other hand, the scores of 'A Robot', which has the on-line planning capability (i.e. $0.0 < \varepsilon < 1.0$ means the planner in the robot can exploit prior experience as depicted in Figure 3 case ii)), gradually increases as depicted in Figure 5. Also, the robot outperforms 'B Robot'. This indicates that the approach enables the robot to, at least, learn its environment and enhance its performance at run-time. In detail, the robot randomly chooses actions in the early stage of the battle as depicted in Figure 3 case i) (exploration). As the rewards of prior actions are accumulated (model construction), the robot can exploit accumulated knowledge to make a plan as shown in Figure 3 case ii) (exploitation). Consequently, the result depicted in Figure 5 shows the robot can actively explore better solutions by using the reinforcement learning-based planner in the presence of dynamically changing environments.

The second experiment was conducted to verify the ef-

[3]When a robot is hit by wall, it causes energy reduction.

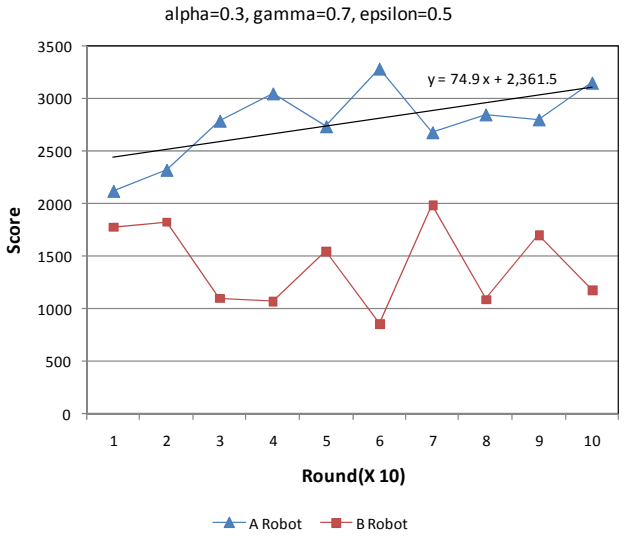


**Figure 5. Scores of 'A Robot' which has parameters: $\alpha = 0.3, \gamma = 0.7, \varepsilon = 0.5$. $\varepsilon = 0.5$ indicates the robot can make plans by using prior knowledge in the presence of each situation. (i.e. the planner adjusts the proportion of exploration and exploitation.)**

fectiveness of on-line planning in self-management. In this experiment, we used the learning data (Q-values) learned by the robot, conducted in the previous experiment (which has parameters $\alpha = 0.3, \gamma = 0.7, \varepsilon = 0.5$). Using the data, 'A Robot' which implements greedy selection (i.e. $\varepsilon = 0.0$ indicates the robot always exploits best-so-far actions in the presence of any situation.) outperforms 'B Robot' constantly as depicted in Figure 6.(a). However, when we introduced a new robot ('C Robot' which has 'Dodge' strategy), which means the environment changed, the robot with same parameters (i.e. exploitation only = off-line planner) cannot outperform the enemy as depicted in Figure 6.(b). This indicates that the robot uses off-line planning and it cannot adapt to dynamic environment changes. This shows the limitation of off-line planning which uses a fixed model for planning. To enable on-line planning, we changed $\varepsilon$ to 0.5 and the result of the battle is shown in Figure 6.(c). The result indicates that 'A Robot' can learn the behavior of 'C Robot' and gradually outperform the enemy. This indicates software systems which apply our approach can search better solutions when it encounters new situations from the dynamically changing environments. The two experiments described in this section represent the effectiveness of the approach described in Section 3 by showing that the robot implemented by the approach can learn the dynamically changing environment and outperform off-line planning.

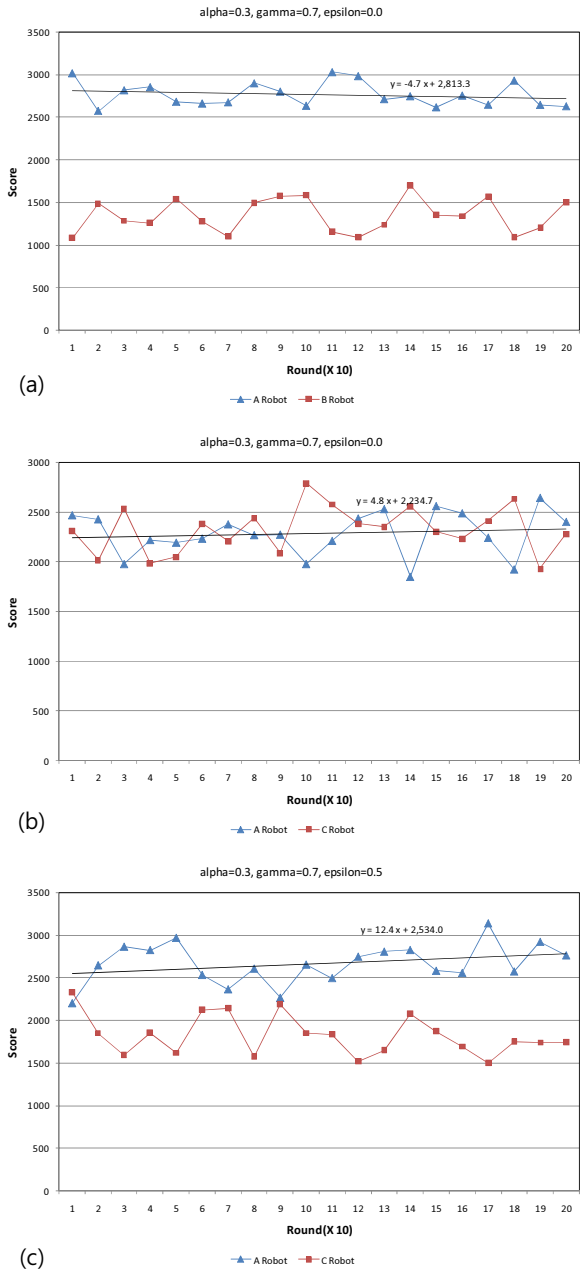


**Figure 6. (a) Exploitation only. The robot uses the previously accumulated Q-values to fight with the already experienced robot without further learning. (b) Exploitation only. the robot uses the Q-values to fight with a new robot. In other words, the robot uses off-line planning and it cannot adapt to dynamic environment changes. (c) Exploration and exploitation. In the early stages, 'A Robot' randomly explores actions and accumulates rewards from the exploration. Then, the robot exploits the knowledge learned to fight with the new robot.**

## 5. Conclusions

Planning in architecture-based self-management represents how a system can find appropriate relationships between situations that the system encounters and possible configurations that the system can take in dynamically changing environments. The paper has discussed two types of planning: off-line and on-line planning. On-line planning must be considered to deal with dynamically changing environments because off-line planning has limitation which uses fixed relationships between situations and configurations for adaptation. On the other hand, on-line planning enables a system to autonomously find better relationships between them in dynamic environments. This paper has described an approach to designing architecture-based self-managed systems with on-line planning based on Q-learning. The approach provides a discovery process of representations, fitness functions, and operators to support on-line planning. The discovered elements are organized by an on-line evolution process. A case study has been conducted to evaluate the approach. The result of the case study shows that our approach is effective for architecture-based self-management.

## A Robot Implementation

Due to space limitation, we provides an additional material about robot implementation on this URL: http://seapp.sogang.ac.kr/robotimpl.pdf

## References


[1] T. Bäck. Adaptive business intelligence based on evolution strategies: some application examples of self-adaptive software. *Inf. Sci.*, 148(1-4):113–121, 2002.

[2] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Mbp: a model based planner. In *IJCAI'01 Workshop on Planning Under Uncertainty and Incomplete Information*, 2001.

[3] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.

[4] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[6] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*, pages 79–88, 2004.

[7] M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[8] IBM. Robocode, http://robocode.alphaworks.ibm.com/home/home.html, 2004.

[9] G. Klein. Flexecution as a paradigm for replanning, part 1. *IEEE Intelligent Systems*, 22(5):79–83, 2007.

[10] G. Klein. Flexecution, part 2: Understanding and supporting flexible execution. *IEEE Intelligent Systems*, 22(6):108–112, 2007.

[11] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[12] R. Laddaga. Active software. In P. Robertson, H. E. Shrobe, and R. Laddaga, editors, *Proceedings of the First International Workshop on Self-Adaptive Software, IWSAS 2000*, volume 1936 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2000.

[13] J. Mylopoulos. Goal-oriented requirements engineering. In *12th Asia-Pacific Software Engineering Conference (APSEC 2005), 15-17 December 2005, Taipei, Taiwan*, page 3, 2005.

[14] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[15] C. Rolland, C. Souveyet, and C. B. Achour. Guiding goal modeling using scenarios. *IEEE Trans. Software Eng.*, 24(12):1055–1071, 1998.

[16] F. Rosenblatt. The perception: a probabilistic model for information storage and organization in the brain. *Neurocomputing: foundations of research*, pages 89–114, 1988.

[17] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Trans. Software Eng.*, 24(12):1072–1088, 1998.

[18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, The MIT Press, 1998.

[19] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[20] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, page 249, 2001.

[21] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. Software Eng.*, 24(12):1089–1114, 1998.

[22] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, New York., 1995.

[23] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.