A Q-Leaning-Based On-Line Planning Approach to Autonomous Architecture Discovery for Self-managed Software*

Dongsun Kim and Sooyong Park

Department of Computer Science and Engineering, Sogang University, Shinsu-Dong, Mapo-Gu, Seoul, 121-742, Republic of Korea
{darkrsw,sypark}@sogang.ac.kr

Abstract. Two key concepts for architecture-based self-managed software are *flexibility* and *autonomy*. Recent discussion have focused on flexibility in self-management, but the software engineering community has not been paying attention to autonomy as much as flexibility in self-management. In this paper, we focus on achieving the autonomy of software systems by on-line planning in which a software system can decide an appropriate plan in the presence of change, evaluate the result of the plan, and learn the result. Our approach applies Q-leaning, which is one of the reinforcement learning techniques, to self-managed systems. The paper presents a case study to illustrate the approach. The result of the case study shows that our approach is effective for 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[1]. Increasing demands for more adaptive software introduced the concept of self-managed software[2]. Self-management is the means in which the software system can change its composition dynamically without human intervention[3]. To achieve self-management, the system needs to maintain a flexible structure and decide appropriate actions in the presence of new situations. Therefore, flexibility and autonomy at run-time are the key properties of self-management.

In this paper, we focus on achieving the autonomy of architecture-based software systems because recent research already addresses the flexibility of architecture-based software systems by using reconfigurable architectures[4,5]. Specifically, we apply an on-line planning approach to self-management to deal with more autonomous behavior in self-management because the previous approaches so far to autonomy concentrate on designing an off-line planning process where adaptation plans are designed at construction time[3].

* This paper was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Commerce, Industry and Energy of Korea.

This paper proposes an Q-leaning[6]-based on-line planning approach to architecture-based self-managed software. To support on-line planning in architecture-based software, this approach presents several elements. Those are (i) representations which describe the current state of a system and possible actions that the system can take, (ii) fitness functions to evaluate the behavior of a system, (iii) operators to facilitate on-line planning, and (iv) a process to apply the previous three elements to actual run-time execution. We also present a case study to verify the effectiveness of the approach.

The paper is organized as follows. The next section gives a brief overview of planning approaches to achieve autonomy in architecture-based self-managed software. 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). 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

2.1 Off-Line Planning

In architecture-based self-management, off-line planning means that decisions which define the relationship between situations (the current state that a software system encounters) and plans (i.e. architectural reconfiguration actions, e.g. adding, remove, and replacing a component) are made prior to run-time. In other words, whenever a system encounters a specific situation *s* from an environment each time, the system selects and executes exactly one identical action *a*. Solutions so far to self-managed systems focus on off-line planning[3]. For example, plans are made by a maintainer through console or hard-coded by a developer[4], components only can restart or reinstall itself when they encounter abnormal states[7], architectural adaptation is described by mapping invariants and strategies in ADL description[8], or architectural changes are triggered by utility functions[5].

These off-line approaches can be effective if developers can identify well-defined goals, states, actions, and rewards 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] because, in real software development, developers make plans with ill-defined goals, limited numbers of states, actions and partially observable rewards, and test environments poorly describing real environments. On-line planning, which gives more effective autonomy in self-management, presents an alternative to overcome the limitation of off-line planning.

2.2 On-Line Planning

On-line planning in self-management software represents that a software system can autonomously choose an action 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 an action which is suitable for the current situation. Generally, the action is chosen by the greedy strategy in which the best-so-far action is chosen. However, this strategy may lead to the problem of local optima. Hence, the system must adjust its strategy between *exploitation* and *exploration*.

In the evaluation step, the system must estimate the effectiveness of the action 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 action 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 convergence speed to optimal planning. On the other hand, if the system uses new experience too much in the step, it may cause ineffective planning.

With these three steps, self-managed software 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 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, simulated annealing, particle swarm optimization; those elements are 'representation', 'fitness', and 'operators'[11]. In reinforcement learning, the representations of states and actions are crucial to shape 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, fitness function design, 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 they define the problem space (situations) and the solution space (architectures) of the system. Instead of intuitive approaches, our approach provides a goal and scenario-based discovery process for more systematic state and action discovery. Goal[12] and scenario-based approaches[13] are widely used for the elicitation of software requirements. Also, goal and scenario discovery have been studied[14].

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 ...

The approach exploits goals and scenarios to discover states and actions. Once goal and scenario structure[14] 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'. 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. These two pieces of information(situation, long-term state) compose the state information. An example of elements of state information is depicted in the leftmost two columns of 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.

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}

Examples of action elements and its type are shown in the rightmost column of Table 2.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 '(Movement=precise maneuver, GUI=rich)' can be mapped with a sequence of '`[add:(visionbased_localizer),connect:(visionbased_localizer)-(pathplanner),replace:(normal_gui)-by-(rich_gui)]`' 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, state and action information can reflect what the system should monitor and how the system should reconfigure its architecture in the presence of environmental changes.

3.2 Fitness

The fitness function of the system is crucial because it represents the reward of the action that the system chooses. Our approach exploits the goal and scenario structure discovered in section 3.1. In particular, goals are the source of fitness function 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 describe 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.

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:

$$r_t = f(t) = \sum_i w_i f_i(t) \tag{1}$$

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 (1) 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 (i.e. learning 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-leaning. Q-learning[6] is one of temporal-difference learning techniques which is a combination of Monte Carlo ideas and dynamic programming ideas[15]. The reason why we choose Q-learning is its modeless characteristic. This characteristic satisfies the properties of on-line planning described in section 2.2. 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. The system can dynamically adapt its architecture by executing these phases repeatedly.

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 representation of states 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. 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.

Planning Phase. Using the state identified by the detection phase, the system chooses an action to adapt itself to the state. This phase is related to the selection step described in Section 2.2 because this phase tries to select an appropriate architectural change with respect to the current situation of the environment. At this phase, the system uses an action selection strategy. In general, Q-learning uses 'ε-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 ε determined by the developer, where $0 \leq \varepsilon < 1$. When planning an action, 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. In this manner, the stochastic strategy prevents the system from falling local optima.

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. 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.

Evaluation Phase. This phase is related to the evaluation step explained in Section 2.2 because this phase determines numerical values which evaluate the previous actions (architectural changes) which can be used by the accumulation step (in our approach, the learning 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.

Learning Phase. In this phase, the system accumulates (as explained in the accumulation step in Section 2.2) the experiences obtained from the previous execution, 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 (2),

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a)] \tag{2}$$

where $0 \leq \alpha \leq 1$ and $0 < \gamma \leq 1$. α is a constant step-size parameter and γ is a discount factor. 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 to choose the best-so-far action.

4 Experiment

This section reports on a case study which applies our approach to an autonomous system. The environment in this case study is Robocode[16] which is a robot battle simulator. Robocode provides a rectangular battle field where user-programmed robots can fight each other.

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

Appendix A provides a robot design which applies our approach. The design includes representations, fitness functions, and operators for the robot.

4.1 Evaluation

This section shows the effectiveness of the approach by presenting the result of robot battles in Robocode.

An experiment was conducted to verify the effectiveness of on-line planning in architecture-based self-management. In this experiment, we implemented a robot based on our approach as described in Appendix A (the robot is denoted ‘A Robot’). Then, we trained the robot for a specific opponent robot (i.e. the opponent robot is an environment of our robot ‘A Robot’). We chose a robot named ‘AntiGravity 1.0’ as the opponent. The opponent (‘AntiGravity 1.0’) is

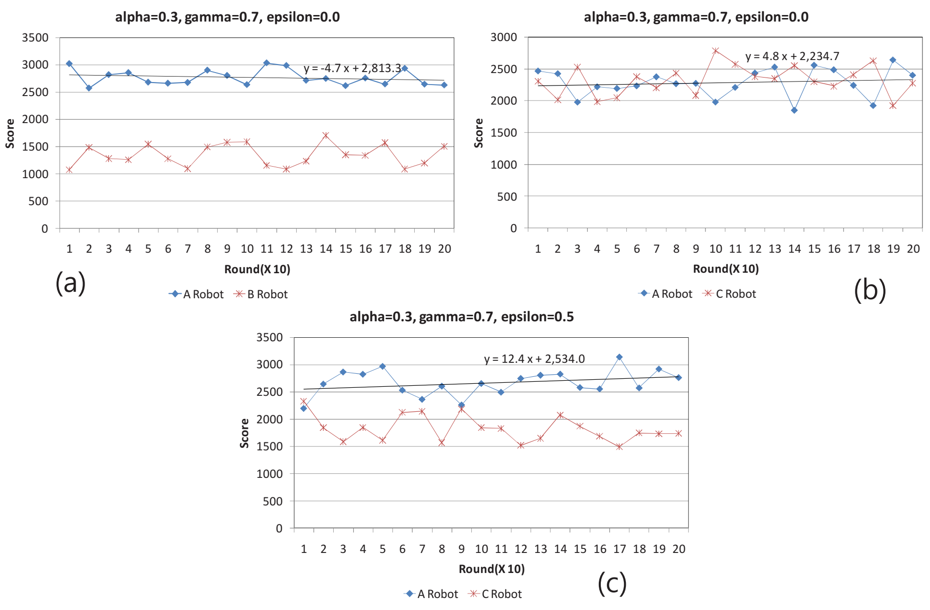

Fig. 1. (a) Exploitation of the previously explored Q-values without further learning. (b) Exploitation of the Q-values to a new robot. (c) Exploration of ‘A Robot’ in the presence of the new robot. X-axis represents rounds that two robots have fought. Y-axis represents scores that each robot obtained.

known for its good performance in battles with many other robots, in Robocode communities. ‘AntiGravity 1.0’ will be denoted by ‘B Robot’. Training is to apply the on-line evolution process described in Section 3.4 to ‘A Robot’ in the battle of two robots (‘A Robot’ and ‘B Robot’). This process made ‘A Robot’ learn the behavior of ‘B Robot’ which is an environment of ‘A Robot’. After training, we investigated the performance of ‘A Robot’ as shown in Figure 1. (a)[1]. The result shows ‘A Robot’ outperforms ‘B Robot’ and this fact means the approach can effectively make the robot learn the environment.

However, when we introduced a new robot(‘C Robot’ which has ‘Dodge’ strategy), which means the environment is dynamically changed, the robot cannot outperform the enemy as depicted in Figure 1.(b). This shows the limitation of off-line planning. To enable on-line planning, we changed ε (epsilon) to 0.5 (which means the robot can try new action with respect to the current situation with 50% probability) and the result is shown in Figure 1.(c). The result indicates ‘A Robot’ can gradually learn the behavior of ‘C Robot’ and finally outperform the enemy. In other words, it can adapt to the new environment and change its plans dynamically without **human intervention**. This indicates software systems which apply our approach can autonomously search better solutions when they encounters new situations from the environments. The experiment described in this section represents 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.

5 Conclusions

Self-managed systems have the potential to provide foundation for systematic adaptation at run-time. Autonomy in self-management is one of the key properties to realize run-time adaptation. To achieve autonomy, it is necessary to provide a planning process to the system. 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. 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. In the process, the system detects a situation, plans courses of action, executes the plan, evaluates the execution, and learns the result of the evaluation. A case study has been conducted to evaluate the approach. The result of the case study shows that on-line planning is effective for architecture-based self-management. In particular, on-line planning outperforms off-line planning in dynamically changing environments.

[1] `epsilon`=0.0 in Figure 1 indicates $\varepsilon = 0.0$ in Equation 2. In other words, the robot does not learn the environment no more and Q-value updating converges to best-so-far actions. `alpha` and `gamma` indicate a constant step-size parameter (α) and discount factor (γ) and the role of each variable is described in [6,15].

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. Laddaga, R.: Active software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, pp. 11–26. Springer, Heidelberg (2001)
2. Garlan, D., Kramer, J., Wolf, A. (eds.): WOSS 2002: Proceedings of the first workshop on Self-healing systems. ACM Press, New York (2002)
3. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society Press, Washington (2007)
4. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE 1998: Proceedings of the 20th international conference on Software engineering, pp. 177–186. IEEE Computer Society Press, Washington (1998)
5. Floch, J., Hallsteinsen, S.O., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)
6. Watkins, C.J.C.H.: Learning from Delayed Rewards. PhD thesis. Cambridge University, Cambridge (1989)
7. Shin, M.E.: Self-healing components in robust software architecture for concurrent and distributed systems. Sci. Comput. Program. 57(1), 27–44 (2005)
8. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
9. Klein, G.: Flexecution as a paradigm for replanning, part 1. IEEE Intelligent Systems 22(5), 79–83 (2007)
10. Klein, G.: Flexecution, part 2: Understanding and supporting flexible execution. IEEE Intelligent Systems 22(6), 108–112 (2007)
11. Harman, M., Jones, B.F.: Search-based software engineering. Information & Software Technology 43(14), 833–839 (2001)
12. van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: 5th IEEE International Symposium on Requirements Engineering (RE 2001), Toronto, Canada, August 27-31, 2001, p. 249 (2001)
13. Sutcliffe, A.G., Maiden, N.A.M., Minocha, S., Manuel, D.: Supporting scenario-based requirements engineering. IEEE Trans. Software Eng. 24(12), 1072–1088 (1998)
14. Rolland, C., Souveyet, C., Achour, C.B.: Guiding goal modeling using scenarios. IEEE Trans. Software Eng. 24(12), 1055–1071 (1998)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, Cambridge (1998)
16. IBM: Robocode (2004), `http://robocode.alphaworks.ibm.com/home/home.html`