AlchemistJ: A Framework for Self-adaptive Software*

Dongsun Kim, Sooyong Park

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

Abstract. The major goal of self-adaptive software is to provide a mechanism that allows a software system to dynamically change its architectural configuration during run-time to cope with requirement changes and unexpected conditions. Software which needs to handle dynamically changing internal and external environment is one of the areas in which self-adaptive software may do an important role in improving the reliability and performance of software systems. There are three main capabilities that are necessary to support self-adaptive software: the ability to monitor and recognize internal/external situations that affect behavior of the software system; the ability to determine when and what to reconfigure in the software system to handle the situations; and the ability to dynamically change the software architecture during run-time to make the reconfiguration effective. In this paper, we describe a software framework to support such capabilities to realize self-adaptive software and its experiment results.

1 Introduction

Intelligent software systems embedded in dynamic devices such as intelligent service robots should provide various capabilities in various environment continuously. This requirement makes the internal operations in software systems more complex. Due to the complexity, unpredictable errors and changes of situation often occurs at run-time[10]. Those unpredictable errors and changes of situations disturb maintaining proper performances of a system at run-time. In addition, it is a major factor which makes stable and persistent services of a system impossible.

In particular, the embedded software, which needed in more dynamic and complex changes of environment, faces the following problems:

1. *Various unexpected situations* that cannot be handled by pre-programmed actions

* This research 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.

2. *Changing user requirements* that are encountered after deploying the embedded software
3. *Faults and errors* (in embedded hardware and software) that make some parts of embedded software unusable or malfunctioned
4. *Frequent replacement and reconfiguration* of hardware components, which may cause some inconsistency problems in software that depends on the hardware components

To overcome the problems above, the embedded software need to have the ability to maintain and improve its own capabilities. The embedded software should maintain three capabilities even in complex and exceptional situation of external environment, which are changing dynamically. The followings are three capabilities to maintain and improve its own functionalities:

1. *Interfaces* for implementing recognition of environment
2. *Decision making and learning* for the adaptive behavior by using reinforcement learning
3. *Dynamic reconfiguration* by using role- and port- based software architecture

Three capabilities above cannot directly overcome all situations such as unexpected problems, changes of user's requirements, disorders in hardware and software, and inconsistency in a system. However, the capabilities offer flexibility and opportunity for software to adapt itself against environmental changes. Self-adaptive software attaches new operations, modifies the current operations, and removes unnecessary operations in order to overcome problems. In this paper, we propose a framework for self-adaptive software called 'AlchemistJ'.

We explain the framework proposed in this research in detail in section 2, and then show a case study using Robocode in section 3. We comment on other significant work in Section 4 before drawing conclusions and proposing future research in section 5.

2 Approach

2.1 Overall Architecture

AlchemistJ consists of the Monitor package('env'), the Decision maker & Learner package('learner') and the Reconfigurator package('recon'). The Monitor package contains interfaces to support observation of situations and rewards of environment. Each situation is a vector of factors which can affect behavior of the software system. Each reward is a real-numbered value that evaluates behavior of the software system. The Decision maker & Learner package contains components to support decision making and learning of software. This makes decisions based on situations and rewards given by a monitor. The Reconfigurator package provides components to maintain software architecture of the systems and reconfigure the architecture based on adaptation decisions from the Decision Maker. AlchemistJ assumes software is designed by using role- and port-based software architecture and reconfigures software architecture at run-time.

Reconfigurations leads to changes of behavior of the software system. The changed behavior affects its environment. This affection changes situations of its environment because of not only the changed behavior but also other factors(for example, human factors, randomness, etc). The software system observes a new situation and reward by using the Monitor. The Learner updates its knowledge using the observed situation and reward. The software system repeats this iterations at run-time to adapt its operations and to accumulate experience.

Fig. 1 shows the process of AlchemistJ explained above.

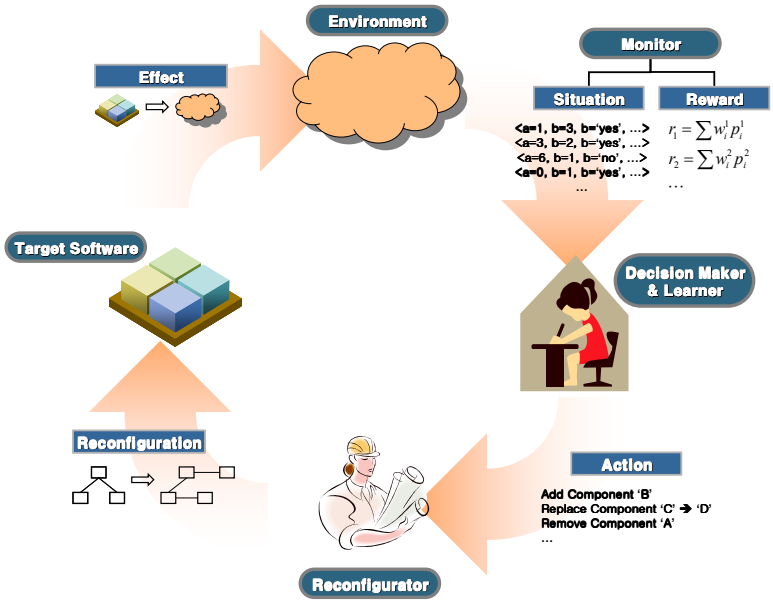

Fig. 1. The Process of AlchemistJ

2.2 The Monitor

The Monitor package defines basic interfaces for observation of environment and a process for passing situations and rewards to the Decision Maker and Learner. This package has three interfaces, 'IState', 'IReward' and 'IEnvironment'.

'IState' defines how to implement situation information. 'IState' has two methods as follows:

1. `public String getStateCode()`
2. `public void setStateCode(String code)`

'getStateCode()' returns the unique situation code which is a string of characters. 'setStateCode()' replaces the unique situation code with a given situation code. Developers should implement a monitor instance which is appropriate for a target domain to observe environment and implement a class to contain situation information using 'IState'. Each object of the class contains a situation of environment. Each situation is mapped to the unique situation code. 'IReward' defines how to express reward information. 'IReward' has one method as follows:

1. `public double getReward()`

'getReward()' returns a real-numbered reward value. This value is determined by a predefined equation. This equation should be defined by developers. AlchemistJ basically assumes software maximize rewards. If a developer is eager to minimize rewards, the Monitor should be implemented to multiply rewards by '-1'.

'IEnvironment' defines connections of the Monitor, the Decision maker & Learner and the Reconfigurator. 'IEnvironment' provides four methods:

1. `public IState getState()`
2. `public boolean giveAction(IAction action)`
3. `public IReward getReward()`
4. `public ActionSet getAdmissibleActionSet(IState s)`

Developers should implement a component in compliance with 'IEnvironment'. 'getState()' returns the current situation of environment. The current situation is a object of the class which implements 'IState'. 'giveAction()' gives an adaptation decision made by the Decision Maker to the Reconfigurator. The Reconfigurator reorganizes components of the software system and composes them into a new architecture. 'getReward()' returns rewards which are observed from environment. Each reward is a instance of the class which implements 'IReward'. 'getAdmisibleActionSet()' returns a set of possible adaptation actions. These actions describe architecture-based adaptation actions such as component replacement, component attachment and topology changes. The Decision Maker determines one of the adaptation actions as the current adaptation decision.

2.3 The Decision maker & Learner

AlchemistJ applies reinforcement learning to the Decision Making & Learning. Reinforcement learning is a mathematical formulation of interaction between an agent and its environment[13]. There are lots of specific algorithms that carry out reinforcement learning. AlchemistJ offers Q-learning implementation which is one of the specific algorithms of reinforcement learning[14][15]. Q-learning assumes a modeless environment. In other words, software cannot know deterministic model of state transitions.

The Decision Maker receives the current situation as input data from the Monitor. Then, the Decision Maker chooses an action to be applied. After applying the action to the system, The Learner receives the next situation and the immediate reward from the Monitor and updates Q-values, $Q(s, a)$. This updating is carried out by equation (1):

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)],$$
$$where \quad 0 < \alpha < 1, \quad 0 < \gamma < 1 \tag{1}$$

In equation (1), $Q(s_t, a_t)$ indicates the Q-value for the situation s_t and the taken action a_t where t is time. The term $(1-\alpha)Q(s_t, a_t)$ represents knowledge that the software system experienced and the term $\alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)]$ represents the current experience where r_{t+1} is an immediate reward from monitor and γ is a discount factor[13]. The value α controls weights of two terms.

Fig. 2 depicts a conceptual process of Q-learning in the Decision Maker & the Learner. In each state the Decision Maker takes an action and moves to the next state. In every transition, the Learner records rewards which is evaluated by the monitor.

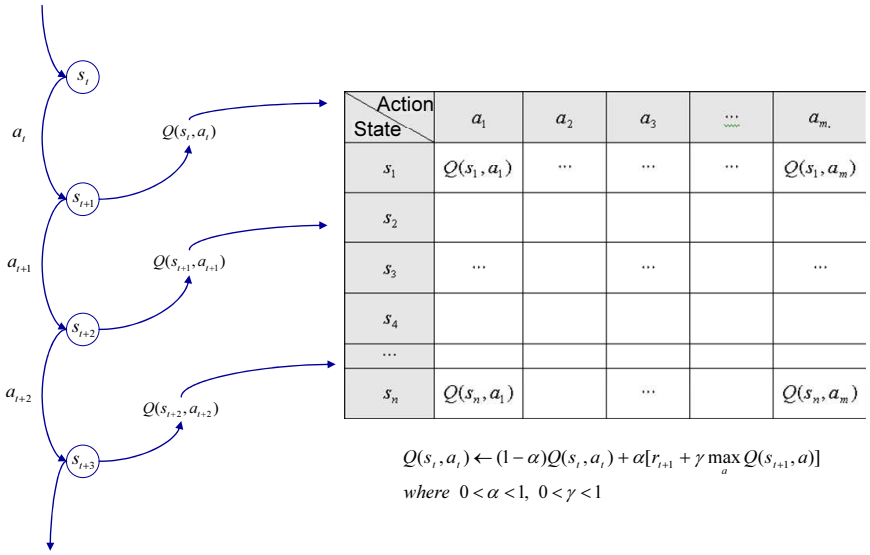

Fig. 2. Q-learning accumulates experiences of the software system by updating Q-values

'learner' package implements the Decision Maker & Learner. 'AbstractLearner' contains general functionalities for decision making & learning. This is connected with the Monitor which implements 'IEnvironment' interface. 'AbstractLearner' receives situation and reward information from the Monitor through 'IEnvironment' interface and makes decisions and learns its environment using the information. Basically, AlchemistJ offers 'QLearner' as an implementation of

'AbstractLearner'. If developers want to design a new decision maker & learner, they should implement a class that inherits 'AbstractLearner' and connect it with the Monitor.

2.4 Role- and Port-based Software Architecture

To make self-adaptation possible, the software system should contain flexible components. These components should be reorganized into architecture to operate its functionalities. For this reason, self-adaptive software development requires predefined component & architecture design principle.

AlchemistJ provides role- and port-based architecture[9][6][3] as a design principle. Assuming software is designed by role- and port-based architecture design method, AlchemistJ reconfigures target software architecture. To construct role- and port-based architecture, developers must decompose software into roles. Each role describes abstract functionalities of each independent part of software. Roles do not express concrete capabilities or strategies. In other words, they should not define 'how to do', but 'what to do'. A role can be connected to make use of another role. After defining roles, they should be composed into a role-model. This role-model is a basis of software architecture.

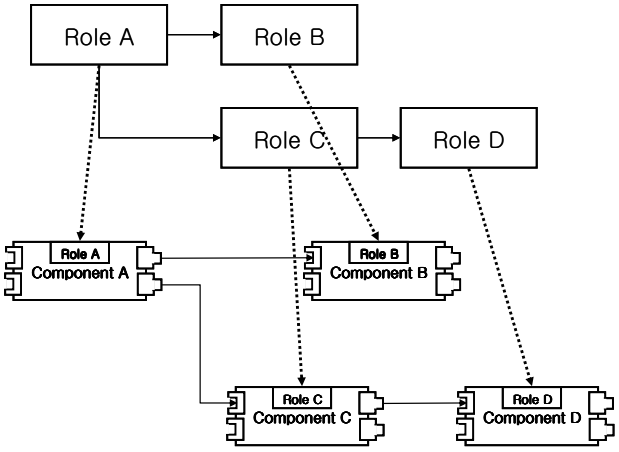

Fig. 3. Correspondence between roles and components

A role-model describes relationship among roles. The Reconfigurator of AlchemistJ makes it possible to change a role-model at run-time. A role can be attached in and removed from the role-model. Each role should correspond to a component. This component should play the corresponding role. Each component consists of component body, role description, required ports, provided

ports. The component body contains executable code that the component executes. The role description indicates a role that the component plays. One component can play only one role. The required ports describes other components(actually provided ports) that the component needs. The Provided ports describe what the component can do. One provided port should be connected with only one required port. Fig. 3 shows a sample of correspondence between roles and components.

2.5 Reconfigurator

AlchemistJ supports reconfiguration of roles and components at run-time. Role reconfiguration consists of role attachment, role removal and changes of topology. Component reconfiguration comprises component attachment because of role attachment, component removal because of role removal, changes of a component topology because of changes of role topology and component replacement. These capabilities are implemented in 'recon' package.

'Reconfigurator' maintains the role-model and components in software architecture and reorganizes components into a new architecture. Components maintained by 'Reconfigurator' should comply role- and port-based architecture. To do this, 'recon' package offers five interfaces. 'IPort' contains basic methods for ports. 'IPortRequired' and 'IPortProvided' which inherit 'IPort' describe required ports and provided ports respectively. All components should implements 'IComponent' interface which provides basic operations of components. 'IRole' contains basic operations to describe a role. Every component has an object implementing 'IRole'. 'recon' package basically provides 'SimpleRole' as an implementation of 'IRole'.

3 Experiment

3.1 Introduction to Robocode

Robocode is a platform which includes the environment for game development and a simulator and is developed by IBM. The game simulator in Robocode provides a rectangular field for a battle. In the field, robots participating battle away each other in order to destroy the opposing robot. In order to simplify the implementation, this paper supposes that only two robots can participate in a battle.

3.2 Implementation of a robot using AlchemistJ

We have implemented the robot's capability to recognize environment according to interfaces provided by 'env' package in the framework. As mentioned in Section 2.2, 'env' package provides three kinds of interfaces. First, 'RobotState' class implements 'IState' interface. 'RobotState' class contains the number of hit by

bullets and the number of hit by wall. This class converts and encodes the current situation into the unique situation code. 'RobotReward' class implements 'IReward' interface. An equation for rewards is set up as follows.

$$r_t = 0.3 \cdot \triangle E_S + (-1) \cdot 0.7 \cdot \triangle E_E \tag{2}$$

In equation (2), r_t, the reward value in time t, is calculated by summation of two terms, $\triangle E_S$ and $\triangle E_E$. $\triangle E_S$ is the difference of the robot's energy between time $t-1$ and t. $\triangle E_E$ is the difference of the enemy's energy between time $t-1$ and t. Weights of two terms are 0.3 and 0.7 respectively. Since the energy of the enemy should be minimized, we multiplied $\triangle E_E$ by (-1).

3.3 Construction of Software Architecture

We design a robot in compliance with Role- and port-based software architecture proposed in section 2.4. Fig. 4 shows the roles of the robot and the relationship among them.

The robot has five roles. 'Maneuver' is responsible for how to move. 'Radar' determines how to find an enemy robot by using a radar, and 'Targeting Strategy' determines how to target the enemy robot. 'Firing Strategy' determines how to fire at the enemy robot. 'RobotBody' has a role to relay over to the other 4 roles. In this experiment, 'Maneuver' and 'Targeting Strategy' are the target of adaptation.

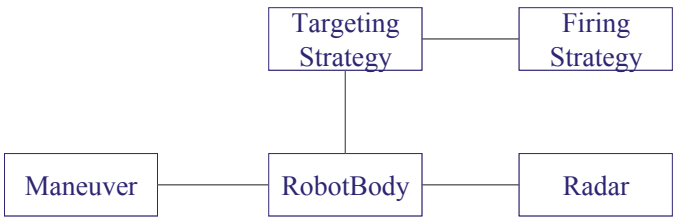

Fig. 4. The Role-model of the Robot

Based on the components shown in table 1, the robot is constituted. Fig. 5 shows the structure of the components assigned roles of the robot in the initial state. The 'LinearManeuver' component works a role as 'Maneuver' and the 'LinearTargeting' does as 'Targeting Strategy' in the initial state of the robot.

Table 1. The component list of robot architecture

Component name	Role of component	Required ports	Provided ports
NewRobot	RobotBody	RadarPort, ManeuverPort, TargetPort	N/A
Radar	Radar	N/A	RadarPortP
LinearManeuver	Maneuver	N/A	ManeuverPortP
CircularManeuver	Maneuver	N/A	ManeuverPortP
AdvancedTargeting	Targeting Strategy	FiringPort	TargetPortP
RammingTargeting	Targeting Strategy	FiringPort	TargetPortP
LinearTargeting	Targeting Strategy	FiringPort	TargetPortP
Firing	Firing Strategy	N/A	FiringPortP

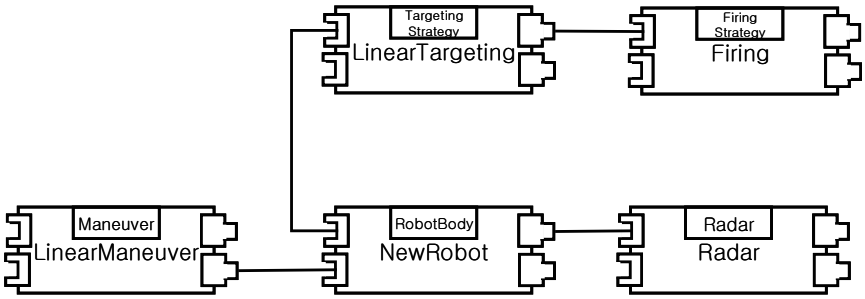

Fig. 5. The Initial Architecture of the Robot

3.4 The result from an experiment

We have performed two experiments in order to verify effectiveness of adaptation of the robot implemented by AlchemistJ. The first experiment is a battle our robot which has no adaptation capabilities(Robot X) and an enemy robot from outside. We design our robot without adaptation capabilities but this robot has 'CircularManeuver' component which plays 'Maneuver' role and 'AdvancedTargeting' component which plays 'Targeting Strategy'. 'CircularManeuver' component outperforms 'LinearManeuver', and 'AdvancedTargeting' component outperforms 'RammingTargeting' and 'LinearTargeting'. The enemy of this robot is 'Antigravity Robot' which is one of the best robots. Fig. 6 shows that 'Antigravity Robot' has a predominant winning average against a robot without adaptation capabilities. On the basis of this experiment, we designed the next experiment.

The second experiment is a battle between our robot with adaptation capabilities(Robot A) and 'Antigravity Robot'. 'Robot A' can reconfigure its architecture using components which plays 'Maneuver' role and 'Targeting Strategy' role such as 'LinearManeuver' and 'RammingTargeting'. Fig. 7 shows the result from the battle between two robots.

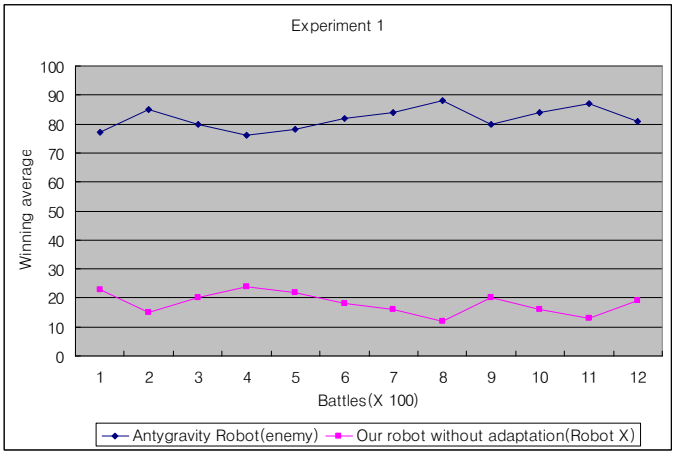

Fig. 6. The Result of the First Experiment

The result from the battle between 'Robot A' and 'Antigravity Robot' shows that 'Robot A' has a 60∼70% winning average against 'Antigravity Robot'. It means that the dynamic reconfiguration with diverse 'Maneuver' and 'Targeting Strategy' components gives better performance than the maintenance with only one moving component and one targeting component even though those outperform other components.

Our approach has the exact difference from a sequence of `if-then-else` statements. A sequence of `if-then-else` statements is hard-coded in source code. Each `if-then-else` statement consists of a condition and an action. To modify a condition or an action, a programmer must rewrite the source code, compile it, shutdown the software system to be modified, re-link the compiled component, and re-execute again. However, AlchemistJ provides the characteristics to reduce the burden of a programmer. Of course, the programmer can make a situation(condition)-behavior(action) lookup table and propose a mechanism for updating the table dynamically, but the mechanism is already another version of AlchemistJ.

4 Related Work

The researches of Richard N. Taylor(University of California, Irvine)[12][11][2], David Garlan(Carnegie Mellon University)[4][1][5], Jamie Hillman(Lancaster University)[8][7] provide the capability of dynamic reconfiguration. The capabilities of dynamic reconfiguration proposed in each framework are different, but they have similarity in terms of changing software structure in the architecture level. All of the three frameworks cannot provide decision making and learning of the

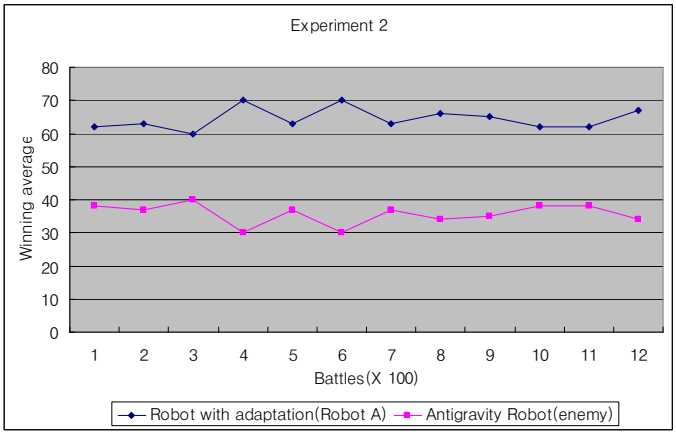

Fig. 7. The Result of the Second Experiment

adaptive behavior. Even though they mention its necessity, they cannot suggest what technology and theory to use and to implement in practice. Except Garlan's research[5], they do not provide a implementation, but conceptually mention only the necessity of it.

We proposed the AlchemistJ framework integrating three capabilities. AlchemistJ is implemented by Java, and comprises three packages to support recognition of environment by defining interfaces for the monitor, decision making and learning by Q-learning implementation, and dynamic reconfiguration using role- and port-based software architecture.

5 Conclusions

Self-adaptive software recognizes environments, and determines the adaptation behavior according to the situations it recognized. In addition, it can change its own architecture in order to perform the adaptation behavior.

We proposed AlchemistJ as a software framework to support developing self-adaptive software. This framework proposed in this paper supports the followings.

1. It prescribes interfaces to support the implementation for recognition of its environment.
2. It employs reinforcement learning to the framework in order to support decision making and learning of the adaptive behavior.
3. It prescribes role- and port-based software architecture and reconfiguration principle in order to support dynamic reconfiguration at run-time.

In order to verify operations of the framework, we have implemented an instance of self-adaptive robot software with Robocode, and examined the ability of adaptation of software from two experiments.

Further study is the development of visualization tools and measurement tools to improve the efficiency of the framework. Also we need to develop a repository for framework to maintain components. In AlchemistJ, we implemented a simple and local repository which can only gives a component corresponding a simple message. The more advanced repository should manage components with more specific description and give components via online connection.

References

1. R. J. Allen, R. Douence, and D. Garlan. Specifying dynamism in software architectures. In *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, 1997.
2. E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-Healing Systems*, 2002.
3. K. R. Dixon, T. Q. Pham, and P. K. Khosla. Port-based adaptable agent architecture. In *First International Workshop, IWSAS 2000*, 2000.
4. D. Garlan, S.-W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, October 2004.
5. D. Garlan, B. Schmerl, and J. Chang. Using gauges for architecture-based monitoring and adaptation. In *the Working Conference on Complex and Dynamic Systems Architecture*, 2001.
6. M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*, 1991.
7. J. Hillman and I. Warren. Meta-adaptation in autonomic systems. In *10th IEEE International Workshop on Future Trends of Distributed Computing Systems*, 2004.
8. J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *26th International Conference on Software Engineering*, 2004.
9. T.-H. Kim and Y.-G. Shin. Role-based decomposition for improving concurrency in distributed object-oriented software development environments. In *23rd International Computer Software and Applications Conference*, 1999.
10. R. Laddaga. Active software. In H. S. Pual Robertson and R. Laddaga, editors, *First International Workshop on Self-Adaptive Software(IWSAS 2000)*, volume 1936 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2000.
11. P. Oreizy. Issues in the runtime modification of software architectures. Technical report, Department of Information and Computer Science, University of California, Irvine, 1996.
12. P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbingner, G. Johnson, N. Medvidovid, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.
13. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, The MIT Press, 1998.
14. C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.
15. C. J. C. H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279 – 292, May 1992.