Yang QL, Lv J, Tao XP *et al.* Fuzzy self-adaptation of mission-critical software under uncertainty. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(1): 165–187 Jan. 2013. DOI 10.1007/s11390-013-1321-9

Fuzzy Self-Adaptation of Mission-Critical Software Under Uncertainty

Qi-Liang Yang[1,2] (杨启亮), *Member*, *CCF*, *IEEE*, Jian Lv[1] (吕　建), *Fellow*, *CCF*, *Member*, *ACM*
Xian-Ping Tao[1] (陶先平), *Member*, *CCF*, *IEEE*, Xiao-Xing Ma[1](马晓星), *Member*, *CCF*, *IEEE*
Jian-Chun Xing[2] (邢建春), *Member*, *CCF*, *IEEE*, and Wei Song[1,3] (宋　巍), *Member*, *CCF*, *ACM*, *IEEE*

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China*

[2]*School of National Defense Engineering, PLA University of Science and Technology, Nanjing 210007, China*

[3]*School of Computer Science and Technology, Nanjing University of Science and Technology, Nanjing 210094, China*

E-mail: yql@893.com.cn; {lj, txp, xxm}@nju.edu.cn; xjc@893.com.cn; wsong@njust.edu.cn

Received March 20, 2012; revised September 29, 2012.

Abstract Mission-critical software (MCS) must provide continuous, online services to ensure the successful accomplishment of critical missions. Self-adaptation is particularly desirable for assuring the quality of service (QoS) and availability of MCS under uncertainty. Few techniques have insofar addressed the issue of MCS self-adaptation, and most existing approaches to software self-adaptation fail to take into account uncertainty in the self-adaptation loop. To tackle this problem, we propose a fuzzy control based approach, i.e., Software Fuzzy Self-Adaptation (SFSA), with a view to deal with the challenge of MCS self-adaptation under uncertainty. First, we present the SFSA conceptual framework, consisting of sensing, deciding and acting stages, and establish the formal model of SFSA to lay a rigorous and mathematical foundation of our approach. Second, we develop a novel SFSA implementation technology as well as its supporting tool, i.e., the SFSA toolkit, to automate the realization process of SFSA. Finally, we demonstrate the effectiveness of our approach through the development of an adaptive MCS application in process control systems. Validation experiments show that the fuzzy control based approach proposed in this work is effective and with low overheads.

Keywords mission-critical software, software self-adaptation, fuzzy self-adaptation, fuzzy control, self-adaptation logic weaving

1 Introduction

Mission-critical systems refer to a class of computing systems whose breakdown or being interrupted may result in mission failure or such undesirable consequences as economic and human losses[1]. Military command and control systems, process control systems, and business financial information systems are typical examples of mission-critical systems. Mission-critical systems are becoming increasingly software intensive to render flexible and smart services during runtime. Mission-critical software (MCS) refers to the software designed for mission-critical systems, which should provide continuous and online service in mission time cycle to assure that critical missions are accomplished successfully. The ability to provide continuous availability and quality of service (QoS) is a key characteristic of MCS. An example of MCS is battlespace-environment-probing software. Once deployed, it should keep continuous availability to collect and send environmental parameters in the battlespace such as temperature, humidity and toxic density back to command systems during a mission cycle. Poor QoS (e.g., delay of data response time) or interruption in software may cause the sad failure of a battle action.

However, during runtime, MCS often encounters changes like unpredicted software aging[2], sudden changes or restrictions in computing resources (e.g., CPU utilization, power and bandwidth), variation in tasks or requirements, and network blast. These uncertain changes could bring about a heavily impact on QoS and MCS availability.

There is thus an urgent need for self-adaptation in MCS. Self-adaptation is a promising approach to settling changes in many systems. In response to changes caused by external environment or requirements, self-adaptation-enabled software can modify itself to satisfy certain objectives[3-4]. When facing unpredicted internal or external changes, MCS with self-adaptability can make decisions by itself, and adjust its states or

Regular Paper

Supported by the National Natural Science Foundation of China under Grant Nos. 60736015, 61073031, 60973044, 61003019, and the National Basic Research 973 Program of China under Grant No. 2009CB320702.

behaviors to provide continuous availability and predefined QoS. Especially, this self-adaptation process of MCS must be online and free of human oversight. Otherwise, any interrupted service or software downtime will lead to very bad consequences, such as economic costs and loss of life.

Unfortunately, due to the increasingly complex internal structures and unfavorable external environment, MCS self-adaptation has become subjected to uncertainty. Uncertainty can be observed from at least two facets: probabilistic uncertainty and fuzzy uncertainty[5]. The former refers to the uncertainty of event occurrences, and it is often related to random phenomena in the objective world. It is dealt with using the probability theory. The latter originates from the vague and inaccurate definitions of concepts existing in human thoughts, and it is always analyzed with fuzzy logic[6-7]. A further distinction between the two can be found in [8]. In this paper, we focus on fuzzy uncertainty because software is a typical handmade artifact in nature that is full of recognitions and human thoughts.

We categorize the uncertainty faced by MCS into three classes: environmental uncertainty, requirement uncertainty and internal uncertainty. Environmental uncertainty results from the external environment where MCS is deployed. MCS often runs in an unfavorable environment that often makes MCS more subject to uncertainty or fuzziness. For example, industrial control software running in plant fields is usually exposed to intense heat, high humidity, and frequent electromagnetic interference. These difficulties caused by an unfavorable environment may lead to unpredicted behaviors (e.g., data communication disconnection) of control software. The second class of uncertainty, i.e., requirement uncertainty, originates from the fuzziness of requirements expressed by users. In the phase of requirement analysis of MCS, it is found that many QoS requirements are often given using natural language consisting of uncertain and fuzzy terms. Hence, the connotation of software requirements is often intuitive, incomplete, and vague. Examples of QoS requirement descriptions include "the response time should be as small as possible" and "the control commands must be received quickly". In the above examples, words like *small* and *quickly* are both ambiguous in meanings. The last class of uncertainty, i.e., internal uncertainty, arises from MCS itself. The increasingly complicated architecture and large scale of MCS are the causes of internal uncertainty. In a very complicated and large MCS system, a self-adaptation decider is very difficult to identify how many computing nodes are in the network, what is the impact of losing a node in the system, and so on. This brings about much uncertainty in interaction, communication and decision in an MCS system. Under this situation, the self-adaptation decider may use ambiguous words like "a great number of" and "very small" to describe the above two questions. For example, the scale of military information systems has expanded from a regional area to the whole globe (e.g., Global Information Grid). This architectural complexity tends to cause difficulty in determining the impact on QoS performance when a software component in MCS self-adaptation is replaced, and then internal certainty in these systems is produced. As for the three classes of uncertainty, environment uncertainty concerns the external physical and computing environments of an MCS system, requirement uncertainty concerns users of an MCS system, and internal uncertainty concerns an MCS system itself.

Features of MCS demand that MCS uncertainty be fully understood, formally described, and rationally settled. It is of high necessity to pay attention to issues of MCS self-adaptation under uncertainty. Several researchers in software engineering have already set off in this direction of study. For instance, Cheng *et al.* suggested that traditional requirement descriptions like "the system *shall* do this ..." need to be relaxed and could be replaced with "the system *might* do this" in adaptive requirement engineering[3].

In this paper, we aim not only to explore a novel approach to achieving MCS self-adaptation but also to address fuzzy uncertainty in self-adaptation.

Software self-adaptation shares the same sense-evaluate-act feedback ideas with the control theory, so building self-adaptive software especially self-adaptive MCS from a control theory viewpoint may be more natural. This view attracts more and more discussions from the software engineering community ranging from the concept level to the implementation level. Concept-level studies propose various models for software self-adaptation based on the feedback control theory[9-11]. Implementation-level studies mostly apply classical control algorithms (mostly PID (proportional-integral-derivative)), to build up self-adaptation strategies[12-13]. Additionally, considering the discrete nature of software, Phoha *et al.* used the supervisory control theory to adjust the behavior of software systems[14].

However, control theory based approaches are still inadequate in the current. First, there is a lack of concern and systematic methods to handle software uncertainty problems. Second, classical control theory based approaches usually require mathematical models (e.g., differential equation) of software systems, whereas the accurate and mathematical modeling of software systems is fairly difficult to obtain in an uncertain environment. Third, current approaches lack intuition, which may result in a big gap between control engineering and software engineering in terms of knowledge and semantics. The design of controllers and control strategies requires too much knowledge of control theory, which is very difficult for software engineering researchers and practitioners. Fourth, the majority of current control-based approaches simply treat software as a black-box-controlled plant and do not go deeply into the interior of software. Finally, few of existing approaches can directly be directly applied to MCS due to the lack of considerations for MCS features.

Different from current control theory based studies, we propose a fuzzy control based approach to achieving software self-adaptation and dealing with the uncertainty of software, MCS in particular. This fuzzy control based software self-adaptation is termed as Software Fuzzy Self-Adaptation (SFSA). In SFSA, feedback control is used to construct an MCS self-adaptation loop, and fuzzy logic is utilized to handle uncertainty and fuzziness.

Self-adaptation is a general concept, which is often classified from four aspects: self-configuration, self-optimization, self-healing and self-protection[15]. On account of the QoS requirements of MCS, this work focuses on the self-optimization dimension of self-adaptation. The main contributions of our work are as follows:

1) An optimized SFSA conceptual framework is proposed by employing the fuzzy control theory.

2) The SFSA formal model is established, and related concepts are precisely defined using fuzzy mathematics. This model provides a mathematical foundation for the application of our approach in mission-critical systems.

3) An implementation technology of SFSA is provided. It gives programming models for SFSA loops, and employs AOP (Aspect-Oriented Programming)-based techniques for weaving SFSA loops into MCS without rewriting MCS source codes.

4) A set of supporting tools called SFSA toolkit is developed. This toolkit can automatically generate aspects in SFSA loops based on Aspect C++ to automate SFSA development. The SFSA toolkit greatly alleviates the burden of self-adaptive software engineers who have little knowledge of control theory.

The proposed SFSA approach has been successfully applied in mission-critical process control systems, such as the development of a self-adaptive MCS-FuzzyLon893OPCServer. Experimental results show that our proposed approach is effective and with low overheads.

The rest of the paper is organized as follows. Section 2 describes the background and detailed motivation of this research. Section 3 describes a running example of MCS (the Lon893OPCServer). The SFSA conceptual framework is presented in Section 4. Section 5 explores the SFSA formal model using fuzzy logic. Section 6 provides the implementation technology of SFSA. In Section 7, we validate our SFSA approach and show how the approach can be successfully applied in implementing MCS of process control. Section 8 outlines related work. Section 9 concludes the paper and presents possible directions for future research.

2 Background and Motivation

In this section, we first provide a background of fuzzy logic and fuzzy control, and then present a detailed explanation of why fuzzy control is adopted for MCS self-adaptation.

2.1 Fuzzy Logic and Fuzzy Control

Fuzzy logic is a mathematical tool for a strict and formal examination of various fuzzy phenomena in the world[6-7]. It uses the whole interval between 0 (false) and 1 (true) to describe human reasoning based fuzzy sets and utilizes natural language style rules to express expert knowledge. Fuzzy logic offers a particularly convenient means of producing a keen mapping between input and output spaces of technical systems because of the natural expression of fuzzy rules.

Fuzzy control combines feedback control with fuzzy logic to build fuzzy controllers that can mimic human decision-making processes[16-17]. Fuzzy controllers provide algorithms that can convert heuristic fuzzy rule based expert knowledge into automatic control strategies. Fuzzy control is particularly useful when the processes of plants are too complex for conventional quantitative means to analyze or when available data is qualitative, inexact or uncertain. A typical fuzzy control loop is illustrated in Fig.1, where r is a set value (a control objective), u control output, y plant output, and e the error between r and y.

2.2 Adoption of Fuzzy Control Methodology

Uncertainty and fuzziness are very serious problems

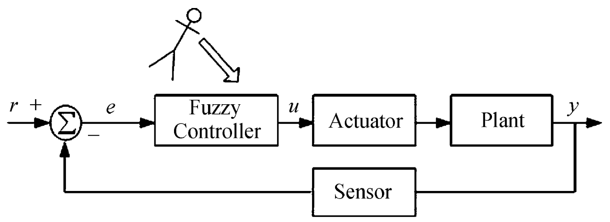


Fig.1. Typical fuzzy control loop.

faced by MCS. Therefore, on the one hand, self-adaptability must be incorporated into MCS to handle unpredictability and complexity. On the other hand, considering the need for high MCS availability and reliability, a strict mathematical tool is required for MCS to deal with fuzziness and uncertainty formally and accurately. If the two aspects simultaneously hold, QoS of MCS can be well guaranteed.

In this paper, we employ a fuzzy control method to address MCS self-adaptation issues.

Mission Critical vs Fuzzy. there are two reasons which make it seem contradictory to introduce fuzzy methodology into mission-critical systems. On the one hand, all MCS states or behaviors are deterministic and can be exactly defined and predicted. On the other hand, fuzzy methodology appears inaccurate, unreliable, and untrustworthy for the very fuzziness.

Despite the two reasons, we, however, argue that uncertainty and fuzziness are the objective problems faced by MCS and that deterministic states or behaviors are subjective MCS requirements. Furthermore, fuzzy methodology (i.e., fuzzy logic and fuzzy control) has a well-corroborated theoretical foundation and is a powerful mathematical tool for exploring and settling fuzzy issues. Introducing fuzzy methodology into MCS is just intended to ensure that MCS states or behaviors can be deterministic. Therefore, in this paper, we suggest that the two terms *mission critical* and *fuzzy* are consistent rather than contradictory. In fact, the fuzzy control theory has been applied in many mission-critical systems, such as train control systems[17] and aircraft flight control systems[19].

In the following, we give some examples to demonstrate the capability of fuzzy methodology and give an accurate description of MCS-related fuzzy issues.

Example 1 (Index Quantification of QoS). In a mission-critical process control system, users often naturally describe their requirements in such ways as "when control commands are sent to devices in plant processes, the devices shall respond *very quickly*". However, designers of this MCS may find much difficulty in precisely translating the fuzzy term *very quickly* into programming language due to the quantitative characteristics of computers. With the help of fuzzy methodology, this can be easily cracked.

Let the universe of device response time be an integer set {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Using fuzzy methodology, we can assign

$$VeryQuickly = \frac{0.7}{1} + \frac{0.9}{2} + \frac{1}{3} + \frac{0.5}{4} + \frac{0.3}{5} + \frac{0}{6} + \frac{0}{7} + \frac{0}{8} + \frac{0}{9} + \frac{0}{10}, \quad (1)$$

where the numerators represent the grades of membership. In this set, 10 being the maximal number, we thus are in a position to say that 10 is "Most Slowly". In this sense, 1, the minimal, means "Most Quickly" or "Very Very Quickly". Similarly, 5, occupying the middle position in the set, denotes "Quickly". As for 3, which is larger than 1 but smaller than 5, corresponds to the fuzzy expression "Very Quickly". Therefore, the membership assignment for "Very Quickly" is 1 when the response time is 3. Following this strain of thought, 2, which is close to 3, is assigned to 0.9, while 1 to 0.7. (1) can be treated as a vector which joins up the computations.

Example 2 (Fuzzy Error Tolerance). Interference from the environment, which leads to sensing errors, also falls under the issue of uncertainty. Suppose external circumstance temperature is sensed for self-adaptation in high-pressure air compressor control software. In the plant field, a high degree of electromagnetic interference (EMI) often results in sensing errors (e.g., the original temperature value 26 is sensed as 26.5), which heavily weakens the self-adaptation function of the software. Fuzzy methodology can in this case be used to reduce external interference errors. Here, circumstance temperature can be identified with fuzzy values in fuzzy logic: Low, Middle, and High. Each fuzzy value corresponds to a range of real values. For the example shown in Fig.2, the Middle temperature value corresponds to the range [25-27]. Therefore, every real temperature value falling in the range belongs to the Middle value. In other words, if the original temperature value 26 is sensed as 26.5, the sensed value is still Middle. This shows the error tolerance ability of fuzzy methodology.

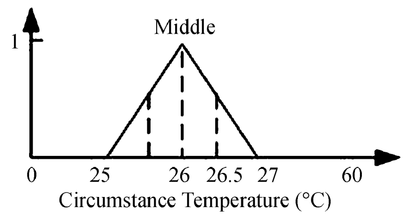


Fig.2. Example of error tolerance ability of fuzzy methodology for MCS.

2.3 Is MCS Controllable

One may point out that MCS is entirely different from traditional controlled physical systems and doubt whether MCS is controllable. Although many facts in real work[11-14] have proven that software is able to be controlled, we still want to informally discuss this question on the light of the definitions of controllability in control theory. Controllability can be generally classified as two kinds: controllability of continuous-time systems (CTS) and controllability of discrete-time systems (DTS)[19]. Since software belongs to discrete systems, we should use controllability of DTS to treat the controllability of MCS. Controllability of DTS in control theory means that the DTS plant is called *controllable* for a pair $(x(k), x^*)$ if there exists a control sequence $u(k), u(k+1), \ldots, u(l-1)$ which moves the plant from the initial state $x(k)$ to the final state x^* in a finite time interval $[k, k+l]$[19]. For most MCS systems, QoS is often described with several important performance indexes, such as response time and CPU utilization. Furthermore, performance indexes are always influenced by MCS parameters, methods and structures. Obviously, we can adjust and control these parameters, methods, and structures to optimize performance indexes and guarantee QoS. For instance, adjusting the *Maxclients* of Apache HTTP server can influence response time[12]. In other words, there theoretically exists a control sequence $u(k), u(k+1), \ldots, u(k+l)$ of adjusting parameters and changing methods in MCS, which moves MCS from an initial performance index $x(k)$ to an expected performance x^* in a finite time interval. Hence, MCS is controllable in most cases.

3 Running Example

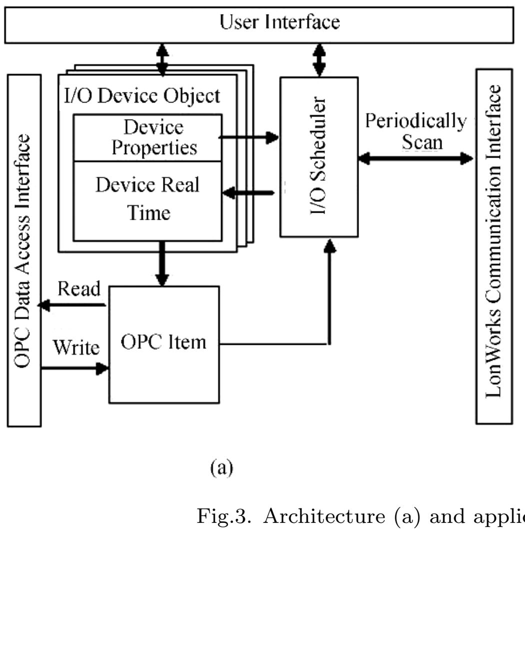
In this section, we give an example of MCS used to illustrate the SFSA approach throughout this paper.

In a previous work, we developed an MCS application called Lon893OPCServer[20] for our LonWorks fieldbus network systems. This application is mission critical and strictly conforming to object linking and embedding for process control (OPC) standard[21], an important specification for the cooperation among real-time process control software. Fig.3(a) is the implementation architecture of Lon893OPCServer. Each of the real-world I/O devices of LonWorks fieldbus is mapped into an I/O device object in the OPC server. Device properties (e.g., *ScanPeriod*) and a real-time database are important elements of these I/O device objects. According to the pre-assigned value of the *ScanPeriod* property of every I/O device object, the I/O scheduler periodically scans every real-world Lonworks I/O device via the LonWorks communication interface (i.e., a set of API functions). The *ScanPeriod* property of every I/O device greatly affects the performance, especially the CPU utilization of the OPC server station. A small *ScanPeriod* value leads to high CPU utilization. The *ScanPeriod* property of Lon893OPCServer can only be configured manually before running.

Through Lon893OPCServer, our LonWorks fieldbus can be accessed transparently by other software systems, such as HMI software and alarm software. The typical application scenario of Lon893OPCServer is shown in Fig.3(b). Lon893OPCServer runs in OPC server stations (i.e., embedded computers). It periodically scans LonWorks-based I/O devices, collects data from process systems, such as valves and pumps, and transfers these process data to HMI software that runs in operator stations for monitoring and control.

Although Lon893OPCServer has been applied in dozens of projects, it still suffers from a problem: it cannot adapt itself to the changes in its external uncertain runtime environment in OPC server stations. In the application scenario shown in Fig.3(a), Lon893OPCServer often finds itself in face of many uncertain changes in its external runtime environment.

• *Unpredicted External Behaviors Caused by the Environment.* These unpredicted external behaviors include the sudden running of assistant software tools like



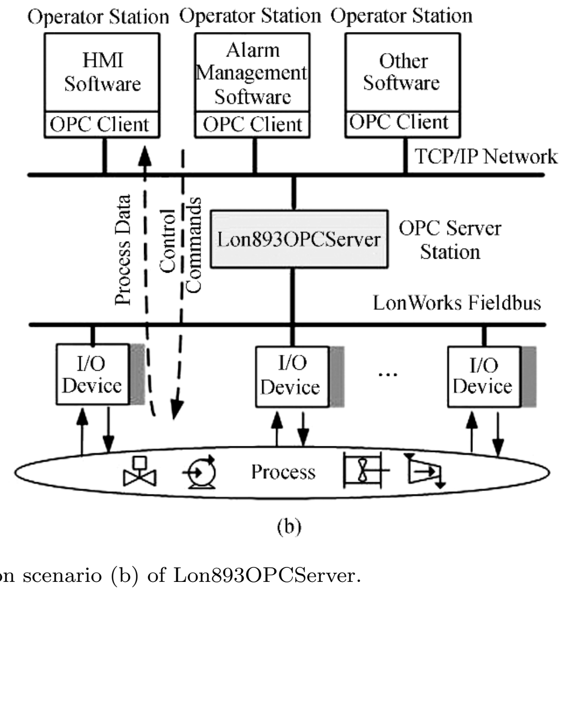


Fig.3. Architecture (a) and application scenario (b) of Lon893OPCServer.

LonWorks Manager and anti-virus software in OPC server stations, and the adding of connections of new OPC clients.

• *Random Interference.* The MCS runs in a plant environment full of various high-power electromechanical devices like generators and dehumidifiers. The start of these devices randomly produces much electromagnetic interference that pollutes the signals sensed by the MCS. These interfered signals can disrupt the stability of the OPC server.

• *Unexpected Problems with I/O Devices.* Due to factors like aging and power down, faults of I/O devices shown in Fig.3 cannot be avoided when the MCS is interacting with I/O devices. In the LonWorks network, a broken-down I/O device often consumes more CPU resources than a good one. Therefore, unexpected problems with I/O devices always cause the sudden rise of CPU utilization. As MCS, the OPC server should therefore have self-adaptation ability to react to sudden changes of CPU utilization in the station in order to guarantee that it can work uninterruptedly when parts of I/O devices fail.

All of the above uncertainty problems always consume many resources of OPC server stations (CPU utilization and memory). Resource reduction can do a severe harm to QoS of MCS, especially in embedded computers. An overly high level of CPU utilization (e.g., over 60%) of the OPC server stations often generates such unpleasant results as response time delay and instability of software systems.

In the past, due to the limited ability that supports only manual and offline configuration of the parameters (e.g., *ScanPeriod*) and the impossibility to restart Lon893OPCServer during runtime, Lon893OPCServer failed to handle the problem of sudden increase in CPU utilization. Therefore, to avoid overly high CPU utilization and ensure the performance of OPC server stations, Lon893OPCServer should have self-adaptation ability, allowing it to alleviate pressure from CPU utilization by automatically adjusting its parameters or behaviors.

As discussed earlier, we can change the CPU utilization by changing the *ScanPeriod* property of I/O devices in Lon893OPCServer. As a result of the inaccessibility of the accurate formal relations between performance (i.e., *ScanPeriod*) and resources (i.e., CPU utilization), existing model-based methods for self-adaptive software cannot meet actual needs. We therefore need systematic ways of achieving software self-adaptation that can better balance performance and resources and thereby guarantee the QoS of MCS in increasingly uncertain contexts and external environment.

4 Conceptual Framework

This section introduces the conceptual framework of SFSA and several important concepts, such as fuzzy adaptors, sensors, and actuators in the framework.

There are two methods (i.e., external adaptation and internal adaptation) for constructing architectures of self-adaptive software with respect to the separation of adaptation strategies and application codes[22]. External approaches employ independent and external adaptation engines to adapt application systems[23-24]. In contrast, internal approaches intertwine the whole set of sensors, actuators and adaptation logic with application codes[25-26]. To avoid the single-point-failure risk of external approaches, we adopt internal approaches for the SFSA framework in this paper. The proposed SFSA framework extends our previous research[27-28]. In comparison with our previous findings, the framework in this paper is more complete and optimized. What distinguishes the present study from previous ones is its internal instead of external self-adaptation architecture, and the more comprehensive definitions of such key software entities in SFSA loops as software sensors, fuzzy adaptors and software actuators. Moreover, a number of concrete examples given to illustrate the new conceptual framework also make it more reader-friendly and persuasive.

4.1 Overview of the Conceptual Framework

This work examines software self-adaptation from the perspective of concern separation and divides self-adaptive software into two parts: self-adaptation logic and application logic. The former uses the sense-decide-act loop to make the software adaptive to external or internal changes. The latter presents domain-specific application to meet the function requirements of users. It consists of several software components. This separation of concerns allows programmers to focus on the development of domain-specific business logic in normal ways. Once self-adaptation ability is required from software, self-adaptation logic can be weaved into the target application logic without influencing old program codes.

This paper, considering the complexity of MCS and the uncertainty of its context, borrows ideas from fuzzy control theory to implement self-adaptation logic and regards application logic as the controlled part. Occasionally, we name the application logic as the target software. Fig.4 illustrates the conceptual framework of SFSA.

As shown in Fig.4, self-adaptation logic is a special automatic control system based on fuzzy logic. Various sensors encapsulated in the software continuously mea-

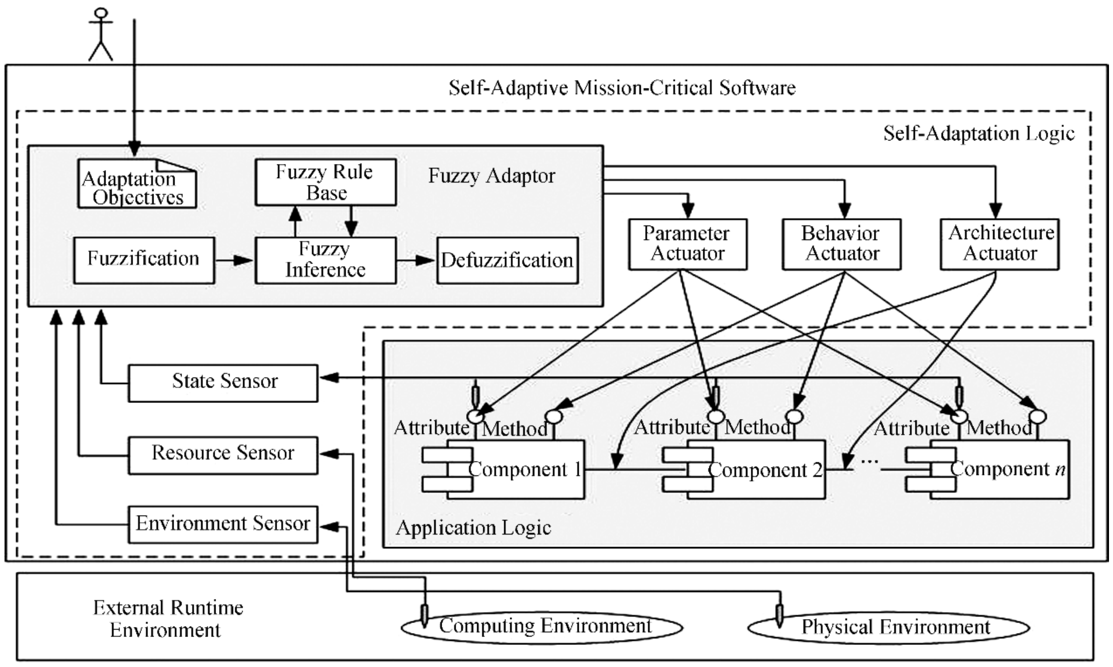


Fig.4. Conceptual framework of SFSA.

sure changes caused by the software and its external runtime environment. These measurements are delivered to a fuzzy adaptor, which uses fuzzy logic to make decisions according to the measurements, takes appropriate control actions to prevent critical situations, and ultimately guarantees QoS of MCS. Through actuators, actions decided by fuzzy adaptors are put on the application logic to regulate MCS parameters, behaviors and architecture. The fuzzy self-adaptation loop can be naturally divided into three stages, namely sensing, deciding and acting stages. The three stages are described in depth afterwards.

4.2 Sensing Stage

In the sensing stage, changes caused by the internal states and external runtime environment of the software are sensed in real time through software sensors for decisions to be made in the fuzzy self-adaptation loop.

Software Sensors. Software sensors are software entities that collect data for a specific internal state of the software system or the external runtime environment.

In our framework, there are three classes of sensors employed for sensing various factors, namely, state, resource and environment sensors. State sensors monitor software elements related to the internal states or attributes of software components. These software elements may be global variables, properties of class and other parameters. Resource sensors monitor computing resources in the software runtime environment, such as CPU utilization, memory in operating systems and throughout and bandwidth of networks. Environment sensors monitor changes in the physical environment related to target software systems. Environmental data may be air temperature and cell power voltage in computing devices. Additional physical sensors or transmitters are required to sense data from the physical environment. Hence, environment sensors in our SFSA framework can be regarded as software entities in software systems that map physical sensors or transmitters in the physical environment.

Example 3. Consider the running example Lon893-OPCServer. To achieve certain self-adaptation objectives, we can deploy various sensors to monitor the software:

• a state sensor to sense the value of *ConnectionNum* $\in [0, 1\,000]$, an attribute of the Lon893-OPCServer to specify the number of current OPC connections initialized by OPC clients (see Fig.3(b));

• a resource sensor to monitor *CPUUtilization* $\in [0, 100]$, which denotes the CPU utilization of an OPC server station;

• an environment sensor to monitor *Power* $\in [0, 50]$, which denotes the cell power voltage of an OPC server station.

Fuzzification. Quantitative and crisp data must first be converted into qualitative fuzzy linguistic values so that they can be interpreted and understood by fuzzy adaptors. This process is known as fuzzification. Fuzzy sets, linguistic variables and other concepts and methods in fuzzy logic are the tools for realizing the fuzzification of crisp data from sensors. The input variable (sensed by software sensors) and the output variable (changed by software actuators) of fuzzy adaptors all need to be fuzzified.

Example 4. In Lon893OPCServer, in order to perform fuzzy inference, the input variable *CPUUtilization* and the output variable *ScanPeriod* must be fuzzified into fuzzy values denoted with fuzzy sets to conduct fuzzy inference. In this case, *CPUUtilization* and *ScanPeriod* are called linguistic variables. The fuzzification process of the two parameters is described below.

• The crisp range of *CPUUtilization* is [0, 100] (in percentage). We identify the fuzzy range of *CPUUtilization* as the set $A = \{$Very Low, Low, Mid, High, Very High$\}$, and $a \in A$ is called the linguistic value. Membership functions are used to quantify linguistic values and thereby automate the fuzzy inference process. The fuzzy membership functions of the above five linguistic values are shown in Fig.5(a). A triangular function is used to describe the four fuzzy values (i.e., Very Low, Low, Mid, and High) and a trapezoidal function is used to describe the fuzzy linguistic value Very High. Through the fuzzy membership functions, a crisp input value is mapped into one or more fuzzy linguistic values with a degree of certainty. For example, in Fig.5(a), the crisp value of *CPUUtilization* (16%) is mapped into one fuzzy linguistic value Low with the certainty 0.933, and another linguistic value Mid with the certainty 0.067 at the same time. In practice, we find that when the CPU utilization of an OPC server station is over 60%, its performance often goes downwards. Therefore, we assign the fuzzy set Very High to 1 when *CPUUtilization* equals or surpasses 60%, the level of certainty which the value of *CPUUtilization* belongs to.

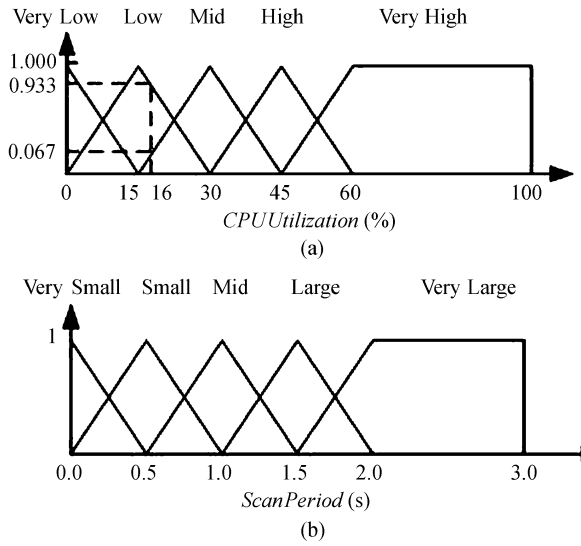


Fig.5. Fuzzy membership functions for the variables (a) *CPUUtilization* and (b) *ScanPeriod*.

• Considering the practical requirements for Lon893OPCServer, the crisp range of *ScanPeriod* is specified as [0, 3] (in seconds). In the same way, we also identify the fuzzy range of *ScanPeriod* as the set $B = \{$Very Small, Small, Mid, Large, Very Large$\}$. The fuzzy membership functions of the five linguistic values are shown in Fig.5(b). A triangular function is used to describe the former four linguistic values, and a trapezoidal function is used to describe the fifth linguistic value Very Large. When *ScanPeriod* $\geqslant 2$, the certainty, which the value of *ScanPeriod* belongs to the fuzzy set Very Large, is 1.

Additionally, self-adaptation objectives, as a kind of special input of fuzzy adaptors, also need to be fuzzified. The fuzzification process is similar to the above.

4.3 Deciding Stage

The deciding stage of the SFSA loop uses results of the sensing stage and fuzzy knowledge rules to make decisions for self-adaptation.

Fuzzy Adaptor. A fuzzy adaptor is a software entity used to make decisions using fuzzy inference algorithms for adapting application logic.

Fuzzy adaptors play a core role in the deciding stage, which involves five parts: fuzzification, self-adaptation objectives, fuzzy rule base, fuzzy inference, and defuzzification. For the sake of easy understanding, we have discussed fuzzification in the sensing stage, and will further discuss defuzzification in the acting stage. According to the observed and fuzzified data obtained in the sensing stage, fuzzy adaptors can make decisions with a certain fuzzy inference mechanism under the guidance of the knowledge contained in the fuzzy rule base.

Self-Adaptation Objectives. Self-adaptation objectives are goals toward which the target software is directed during the dynamic process of self-adaptation. Each self-adaptation process of software should be aimed towards the following goals: enhancing QoS, guaranteeing performance or cutting maintenance costs, and so forth. Some self-adaptation objectives can be an online set or modified through user interfaces, while others can be solidified and hidden into a fuzzy rule base before the running of the target software. In most cases, self-adaptation objectives can only be infinitely approached but never reached. The self-adaptation process of software is also a process that dynamically converging to self-adaptive objectives.

Example 5. Consider the running example Lon893OPCServer. We know that the software cannot adapt itself to the sudden increase of CPU utilization in the runtime environment. We expect that the software can automatically change some of its important parameters or behaviors to reduce its CPU utilization when the total CPU utilization of the environment is high because high CPU utilization often results in bad performance (e.g., delay of control instructions). Therefore, one of the fuzzy self-adaptation objectives of Lon893OPCServer can be defined as follows:

$O1$: automatically adjusting its CPU utilization to maintain a running environment with proper CPU utilization.

Additionally, this specified self-adaptation objective is not required to be tuned online, so we can hide it in the fuzzy rule base.

Fuzzy Rule Base. A fuzzy rule base is used to hold the knowledge of how to best adapt the target software system in the form of a set of word-based rules. It is a collection of if-then fuzzy self-adaptation rules. In an if-then rule, the antecedent is composed of the sensed variables, and the consequent is composed of the control variables. The form of a fuzzy rule base can be described as:

$$\begin{aligned}&R_1 : \text{if } A \text{ is } A_1, \text{then } B \text{ is } B_1,\\ &R_2 : \text{if } A \text{ is } A_2, \text{then } B \text{ is } B_2,\\ &\vdots\\ &R_n : \text{if } A \text{ is } A_1, \text{then } B \text{ is } B_n,\end{aligned}$$

where A is the fuzzy input linguistic variable of fuzzy adaptors, B the fuzzy output linguistic variable, and A_i and B_i $(i = 1, 2, \dots, n)$ the individual fuzzy linguistic values of A and B.

To construct a fuzzy self-adaptation rule base, it is very necessary to know how to get this experience and knowledge. We propose two approaches to obtaining the software adaptation knowledge of software systems.

• The first approach is to interview the designers of the software system to which the adaptation ability will be added. The designers have a clear knowledge of the internal mechanisms and implementation principles of the software system. We can use a carefully designed questionnaire to get a fuzzy model of the software process.

• The second approach is to ask skilled administrators or operators of software systems. Although the administrators (operators) may not know the input-output relations with sufficient precision, they can manage or control such a system quite successfully without having any quantitative models in mind. In effect, a human administrator employs — consciously or subconsciously — a set of if-then rules to manage and control the software process.

Example 6. According to the self-adaptation objective defined in Example 5 and our observations on the dynamic characteristics of Lon893OPCServer, we can establish the fuzzy self-adaptation rules displayed in Table 1 for the fuzzy adaptor.

Table 1. Fuzzy Self-Adaptation Rules

ID	*CPUUtilization* (Premise)	*ScanPeriod* (Consequent)
R_1	Very Low	Very Small
R_2	Low	Small
R_3	Mid	Mid
R_4	High	Large
R_5	Very High	Very Large

Table 1 consists of five if-then rules. *CPUUtilization* is the input variable, and *ScanPeriod* is the output variable. Two detailed example rules in Table1 are described as follows:

R_1: If *CPUUtilization* is Very Low, then *ScanPeriod* is Very Small.

R_5: If *CPUUtilization* is Very High, then *ScanPeriod* is Very Large.

Fuzzy Inference Mechanisms. Fuzzy inference mechanisms evaluate which self-adaptation rules are relevant to current input and decide what actions for the target software should be taken. In the SFSA framework, we use the minimum inference engine[17] to realize fuzzy inference, which can reduce computing complexity and guarantee the real-time response ability of fuzzy adaptors. Using this method, the fuzzy value specified in the consequent of a fuzzy rule (e.g., R_1) is clipped off at a height corresponding to the degree of truth of the rule's antecedent. After the evaluation of fuzzy rules, all decided fuzzy values regarding the same output linguistic variable are combined using the fuzzy union (i.e., maximum) operation. The resulting combined fuzzy set is the final outcome of the inference step. This fuzzy inference process can be formally expressed as

$$\mu_B^t(y) = \bigvee_{x \in X} [\mu_A^t(x) \wedge (\mu_A(x) \wedge \mu_B(y))], \ x \in X, \ y \in Y,$$

where $\mu_A(x)$ and $\mu_B(y)$ are the membership functions of fuzzy linguistic A and B respectively, $\mu_A^t(x)$ the fuzzy value of A at time t, and $\mu_B^t(y)$ the fuzzy value of B decided by the inference mechanism at time t.

Example 7. Consider the running example Lon893OPCServer. Suppose the value of *CPUUtilization* measured from the sensor is 16% at time t_0. In the fuzzification phase, in terms of the membership functions for *CPUUtilization* defined in Fig.5(a), the crisp value 16% simultaneously corresponds to two fuzzy values, Low and Mid ($\mu_{\text{Low}}(16) = 0.933$ and $\mu_{\text{Mid}}(16) = 0.0667$; see left part of Fig.6). The detailed process of the fuzzy inference is described as below.

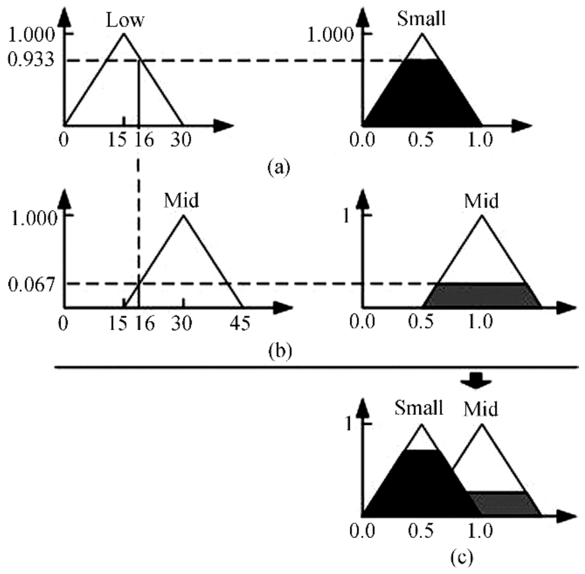


Fig.6. Fuzzy self-adaptation inference process. (a) If *CPUUtilization* is Low, then *ScanPeriod* is Small. (b) If *CPUUtilization* is Mid, then *ScanPeriod* is Mid. (c) Inference result combined using the fuzzy union operation.

• The two fuzzy values, Low and Mid, activate the two rules, R_2 and R_3, in the fuzzy rule base listed in Table 1. According to the minimum inference engine method, R_2 recommends that the fuzzy value of the output variable *ScanPeriod* is Small with the certainty 0.933, and R_3 recommends that the fuzzy value of *ScanPeriod* is Mid with the certainty 0.0667.

• The two recommendations are combined using the fuzzy union operation, which is shown with the shaded region at Fig.6(c).

This part presents an example of fuzzy adaptors with rather simple fuzzy control mechanisms. In fact, in our conceptual framework of SFSA, a fuzzy adaptor can be thought as an abstract container, and many complicated fuzzy control mechanisms such as fuzzy-PID control policies can be encapsulated into it. What fuzzy mechanisms are selected depends on the specific application scenario.

4.4 Acting Stage

The acting stage converts decisions made by fuzzy adaptors in the deciding stage into executable actions and further puts these actions on application logic to adapt the target software. The stage involves two steps. The first step, called defuzzification, interprets fuzzy linguistic values into crisp values so that they can be recognized by software entities in computing space. Meanwhile, the second step employs the crisp decisions to adjust attributes, behaviors or structures of application logic.

Defuzzification. Decisions made by fuzzy inference are in the form of fuzzy linguistic values. These fuzzy values cannot be directly used to adapt the application logic. Defuzzification is required to convert the fuzzy conclusions drawn by the inference mechanism into crisp values for actuators. There are several different defuzzifying methods proposed in the literature. In our framework, we use the center average defuzzification method[17] due to its popularity. The method computes the crisp values as follows:

$$y^{\mathrm{crisp}} = \frac{\sum_i y_i \mu(y_i)}{\sum_i \mu(y_i)}, \tag{1}$$

where y^{crisp} is the defuzzified crisp output, y_i the center of the membership function (i.e., where it reaches its peak) of the consequent of rule i, and $\mu(y_i)$ the implied fuzzy set for rule i (i.e., it is the conclusion implied by rule i).

Example 8. In terms of the results of Example 7, we have $y_1 = 0.5$, $\mu(y_1) = 0.933$, $y_2 = 1$, and $\mu(y_2) = 0.0667$. Using the center average defuzzification method, we have $y^{\mathrm{Crisp}} = 0.355$. The crisp value 0.355 is used to modify the *ScanPeriod* parameter in Lon893OPCServer. When current CPU utilization is 16%, our fuzzy self-adaptation approach decides that the value of *ScanPeriod* should be 0.355s.

Software Actuator. An actuator is a software entity that acts to change the parameters, methods and structure of application logic.

Our SFSA framework classifies actuators into three categories: parameter actuators, behavior actuators, and structure actuators. Parameter actuators are used to dynamically modify some attributes or configuration parameters (e.g., time limit) of components. Behavior actuators are used to change the behaviors of components by changing the method of calling. When a fuzzy adaptor finds that calling a method of a component cannot meet the requirements of self-adaptation objectives, it will automatically change to call another alternative method through a behavior actuator. Structure actuators are software entities used to change connectors among components or replace some components to improve the QoS of software.

Example 9. For Lon893OPCServer, a fuzzy adaptor can change the software through actuators.

• A parameter actuator is used to adjust the *ScanPeriod* attribute of the *ODevice* class.

• A behavior actuator is used to change the local data-refreshing behavior. When CPU utilization of the OPC server station is very high, the actuator calls the *StopDataRefresh* method to stop the refreshing behavior and release the pressure on CPU.

• A structure actuator is used to change the connector between the OPC server and its client. When the fuzzy adaptor finds the network bandwidth very low, to avoid data loss, it breaks the connector between the server and the client to stop data transmission through the network and save the data from devices on the local disk.

5 SFSA Formal Model Using Fuzzy Logic

In the previous section, we have discussed our fuzzy control-based approach for achieving software self-adaptation in a conceptual and intuitive way. In this section, we model software fuzzy self-adaptation in a mathematical and strict way to show that our approach has solid mathematical foundations and can therefore be confidently applied in MCS. We will give several formal definitions of fuzzy self-adaptation concepts in Subsection 5.1 and formally describe the process of software fuzzy self-adaptation using fuzzy theory in Subsection 5.2.

5.1 Definitions Concerning SFSA

Before discussing software fuzzy self-adaptation, we will firstly deal with the definition of software self-adaptation. Part of our idea was inspired by the model proposed by Subramanian[29].

Definition 1 (Software Self-Adaptation). *Self-adaptation of a software system* (S) *is a process caused by changes* (δ_C) *in its running contexts* (U) *involving its external environment* (U_E) *and its internal runtime states* (U_S), *and directed by certain pre-set objectives* (O) *to result in a new software system* (S') *that meets the needs of the new contexts* (U').

Formally, software self-adaptation can be considered as a function:

$$f_{\text{Self-adapation}} : U \times U' \times O \times S \to S', \tag{2}$$

where $meet(S', need(U'))$ *and* $U = U_E \cup U_S$.

Therefore, a software self-adaptation process involves three tasks: sensing the changes $\delta_C = U' - U$, deciding the change δ_S to be made to the software system S on the basis of δ_C and the objective $o \in O$, and acting the change δ_S on the target software S to generate the new software system S'.

Definition 2 (Fuzzy Self-Adaptive Software System). *A fuzzy self-adaptive software system can be formally defined as a tuple as follows*:

$$F3S = \langle FSAL, AL, IF, E\rangle \tag{3}$$

• *FSAL means fuzzy self-adaptation logic, where* $FSAL = \langle FuSensor,\ FuAdaptor,\ FuActuator\rangle$. *A fuzzy self-adaptation logic of software is composed of a set of fuzzy sensors, a set of fuzzy adaptors, and a set of fuzzy actuators.*

• *AL means application logic.*

• *IF means software interface.*

• *E means runtime environment.*

Definition 3 (Software Fuzzy Self-Adaptation). *Software fuzzy self-adaptation is a kind of software self-adaptation, where self-adaptation decisions are made using fuzzy logic and the expert knowledge set* K. *Interactive relations among software elements, such as fuzzy sensors, fuzzy adaptors, fuzzy actuators and application logic, are described with fuzzy relations.*

Formally, software fuzzy self-adaptation is a kind of nonlinear map, which can be viewed as a function:

$$f_{\text{Fuzzy-self-adapation}} : U \times U' \times O \times K \times S \xrightarrow{FuzzyLogic} S', \tag{4}$$

where $meet(S', need(U'))$ *and* $U = U_E \cup U_S$. (4) *can be further separated into the following functions*:

• *ContextSensing* : $U \times U' \to \delta_C$, *used to sense the changes of contexts.*

• *Fuzzifying* :$\delta_C \to F(\delta_C)$, $O \to F(O)$, *used to convert crisp values of changes and objectives into fuzzy values, where* $F(\cdot)$ *means fuzzifying operators.*

• *Deciding*: $F(\delta_C) \times F(O) \times K \times S \xrightarrow{FuzzyLogic} F(V)$, $V = V_P \cup V_M \cup V_S$, *used to make decisions through fuzzy logic.*

• *Defuzzifying*: $F(V) \to V$, *used to convert fuzzy decision results into crisp acting values.*

• *Acting*: $V \times S \to S'$, *used to act results of self-adaptation decisions on application logic to adjust parameters, behaviors or structures.*

All the notations in Definition 3 are listed in Table 2.

Table 2. Notations in Definition 3

Notation	Explanation
U	Set of running contexts of software
U'	Set of new running contexts of software
U_E	External environment of software
U_S	Set of internal states of software
δ_C	Set of changes of contexts
O	Set of self-adaptation objectives
K	Set of fuzzy rule-based expert knowledge
S	Software system
S'	New software system
$F(\cdot)$	Fuzzifying operators
V	Set of actions on application logic
V_P	Set of actions to modify parameters of application logic
V_M	Set of actions to modify methods of application logic
V_S	Set of actions to modify structure of application logic

5.2 Modeling the SFSA Loop

In this subsection, we model the SFSA loop using

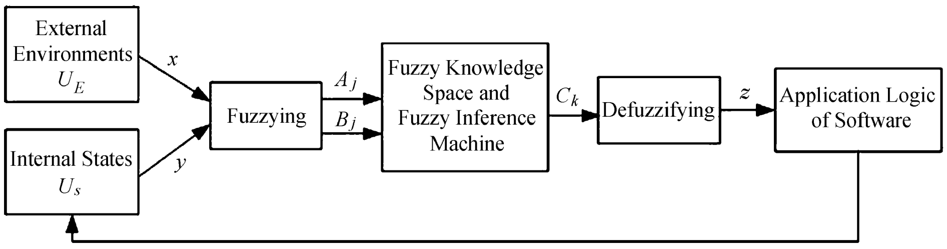

External Environments U_E
Internal States U_s
x
y
Fuzzying
A_j
B_j
Fuzzy Knowledge Space and Fuzzy Inference Machine
C_k
Defuzzifying
z
Application Logic of Software

Fig.7. Abstract illustration of a software fuzzy self-adaptation loop.

fuzzy mathematics. We present an abstract illustration (Fig.7) for our conceptual framework in Section 4. According to the fuzzy logic theory[17], an MIMO (multi-input and multi-output) fuzzy system can always be decomposed into several MISO (multi-input and single output) fuzzy systems. Therefore, for ease of understanding, we suppose that the fuzzy self-adaptation loop has two input variables, x and y, and one output variable, z, in the abstract model. The variables in the abstract self-adaptation loop are

- $x \in U_E$, an external environment variable;
- $y \in U_S$, an internal state variable of the software;
- $z \in V$, an action variable for modifying S;
- $A_i \in F(U_E)$ $(i = 1, 2, 3, \ldots, I)$, a fuzzy linguistic value of x, which is a fuzzy set on the universe of discourse U_E;
- $B_j \in F(U_S)$ $(j = 1, 2, 3, \ldots, J)$, a fuzzy linguistic value of y, which is a fuzzy set on the universe of discourse U_S; and
- $C_k \in F(V)$ $(k = 1, 2, 3, \ldots, K)$, a fuzzy value of z, which is a fuzzy set on the universe of discourse V.

There are three key stages in the abstract self-adaptation loop: 1) converting software context space into fuzzy space, 2) inferring fuzzy self-adaptation based on the knowledge base, and 3) converting fuzzy results into action space.

5.2.1 Converting Software Context Space into Fuzzy Space

The software context sensed by software of sensors is composed of crisp values. These values must be converted into fuzzy values in fuzzy space before fuzzy decisions are made on self-adaptation. The conversion process is realized with membership functions (Fig.8).

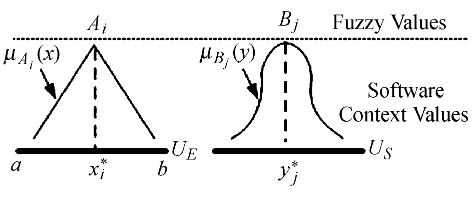


Fig.8. Fuzzifying software context space using membership functions.

As shown in Fig.8, x_i^* is a crisp value of $x \in U_E$ at a certain moment and A_i is a fuzzy linguistic value: *About* x_i^*. A triangular function $\mu_{A_i}(x) : R \to [0, 1]$ can be used to map x_i^* into A_i. The function describes the certainty that a crisp value of x is close to x_i^*. The membership value at the number x_i^* should be a maximum, so $\mu_{A_i}(x_i^*) = 1$. Furthermore, the membership function should be decreased as x moves away from x_i^*. The mathematical formula for the triangular function is as follows:

$$\mu_{A_i}(x) = \begin{cases} \dfrac{x-a}{x_i^*-a}, & \text{if } x \in (a, x_i^*], \\ \dfrac{b-x}{b-x_i^*}, & \text{if } x \in (x_i^*, b], \\ 0, & \text{if } x \notin [a, b]. \end{cases} \tag{5}$$

In our model, in addition to the triangular function, membership functions can also choose other standard shapes: Gaussian and trapezoidal. For example, as shown in Fig.8(b), a Gaussian function $\mu_{B_j}(x) : R \to [0, 1]$ is used to fuzzify a crisp value $y_j^* \in U_S$ into a fuzzy linguistic B_j. The mathematical formula for the Gaussian function is as follows:

$$\mu_{B_j}(y) = e^{-\frac{(y-y_j^*)^2}{2\delta^2}}. \tag{6}$$

5.2.2 Inferring Fuzzy Self-Adaptation

The process of self-adaptation inference depends on the knowledge base established with fuzzy if-then rules.

Definition 4 (Fuzzy Self-Adaptation Rule). *Fuzzy self-adaptation rules electronically solidify expert experience on how to adjust software to achieve certain goals.*

Formally, a fuzzy self-adaptation rule in our formal model is defined as follows:

$$R^{(l)} : \text{IF } x \text{ is } A_i \text{ and } y \text{ is } B_j, \text{ THEN } z \text{ is } C_k,$$

where $i = 1, 2, 3, \ldots, I$; $j = 1, 2, 3, \ldots, J$; $l = 1, 2, 3, \ldots, L$ and $\max(L) = I \times J$. This rule can be expressed through a fuzzy implication relation:

$$R^{(l)} = (A_i \text{ and } B_j) \to C_k,$$

so we have

$$\begin{aligned}\mu_{R^{(l)}}(x,y,z) =&\mu_{(A_i \text{ and } B_j \to C_k)}(x,y,z)\\ =&(\mu_{A_i}(x) \text{ and } \mu_{B_j}(y)) \to \mu_{C_k}(z).\end{aligned} \tag{7}$$

Here, we adopt Mamdani implication[17] (i.e., minimal implication) to compute fuzzy implication in a fuzzy rule. Consequently,

$$\begin{aligned}\mu_{R^{(l)}}(x,y,z) =&(\mu_{A_i}(x) \text{ and } \mu_{B_j}(y)) \wedge \mu_{C_k}(z)\\ =&(\mu_{A_i}(x) \wedge \mu_{B_j}(y)) \wedge \mu_{C_k}(z).\end{aligned} \tag{8}$$

Therefore, for a fuzzy self-adaptation rule base $R_{\text{self-adaptation}}$ made up of L rules, the total fuzzy implication is

$$R_{\text{self-adaptation}} = \bigcup_{l=1}^{L} R^{(l)}$$

or

$$\mu_{R_{\text{self-adaptation}}}(x,y,z) = \max_{l=1}^{L}(\mu_{R^{(l)}}(x,y,z)). \tag{9}$$

Definition 6 (Fuzzy Self-Adaptation Inference). *Fuzzy self-adaptation inference is a computing process supported by a fuzzy rule base, where a group of given fuzzy input values A' and B' from the software context is used to decide a fuzzy output value C' for adjusting parameters, behaviors or structures of MCS and thereby achieve self-adaptation goals. The inference process is formally expressed as*

$$C' = (A' \text{ and } B') \circ R_{\text{self-adaptation}}, \tag{10}$$

where "$\circ$" is a fuzzy composition operator.

In our formal model, we use the popular max-min composition method[17] to make fuzzy self-adaptation inference. Hence, we have

$$\mu_{C'}(z) = \max_{\substack{x \in U_E \\ y \in U_S}} \min(\mu_{A' \times B'}(x,y), \mu_{R_{\text{self-adaptatioin}}}(x,y,z)). \tag{11}$$

5.2.3 Converting Inference Results into Action Space

The conclusions drawn by fuzzy self-adaptation inference are fuzzy values and therefore cannot be used directly to change software in the real world. For example, it is impossible to use the fuzzy value Very Large to modify a software parameter because the value of the parameter must be crisp and numerical in the computing world. Therefore, converting fuzzy inference results into action space is a necessary step in the fuzzy self-adaptation loop. This conversion process is called defuzzification. Several different defuzzification methods have been proposed in the literature. In our model, we adopt the popular Center Average Defuzzification method[17], which has already been introduced in Subsection 4.3.

6 Implementation Technology and Tools Supporting SFSA

In the earlier sections, we have proposed the SFSA conceptual framework in order to achieve MCS self-adaptation and presented a formal model for a strict description of the fuzzy self-adaptation loop. However, there still exist several problems: how to realize and engineer the SFSA conceptual framework into real-world software systems and what tools to use in weaving fuzzy self-adaptation logic into target MCS. These questions will be answered in this section.

In recent years, there have been a few studies addressing the implementation problem of building self-adaptation loops into software. Some use middleware or object-oriented (OO) technologies to build sensors or effectors[30-31], and others employ aspect-oriented programming (AOP) technologies to weave, in either a partial or full manner, policy aspects, sensor aspects and effector aspects into software[32-35].

In this paper, based on the characteristics of MCS, we propose an SFSA implementation technology involving programming models of SFSA loop based on AOP and provide a set of supporting tools (i.e., the SFSA toolkit) for automating the development process of software fuzzy self-adaptation. Different from other studies, our AOP-based programming models for SFSA is based on AspectC++[①] because, to the best of our knowledge, many MCS systems are programmed with C/C++ language. It can weave pure control loops into MCS. Compared with other component-based or pure OO-based approaches, AOP is more lightweight, promoting the execution efficiency of MCS. The general architecture of the proposed SFSA implementation technology is illustrated in Fig.9.

Our SFSA implementation technology borrows the idea of concern separation. It treats fuzzy self-adaptation logic as separated modules independent from target software systems. AOP technologies are introduced to support this separation. The fuzzy self-adaptation loop is encoded not in the target software system but in the separated modules (i.e., aspects) that externalize the crosscutting composition of the self-adaptation loop with the target software. When needed, this aspect code is instrumented into the target software system using a weaver (a special compiler and associated runtime library). After the weaving, fuzzy

[①]http://www.aspectc.org, March 2011.

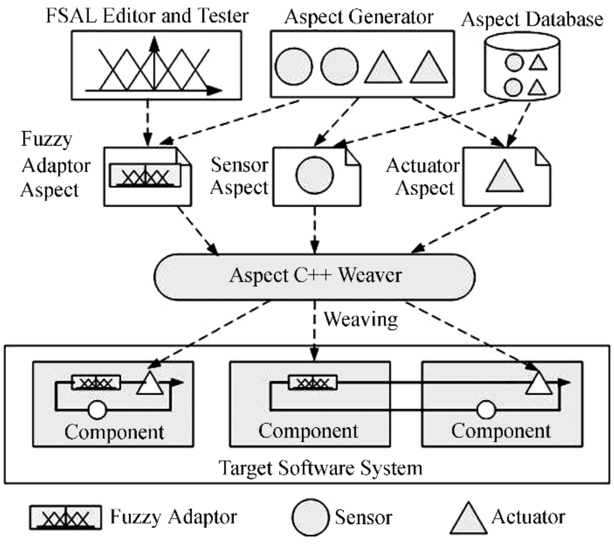


Fig.9. General architecture of SFSA implementation technology and its supporting tools.

self-adaptation loops, consisting of fuzzy adaptors, sensors and actuators, are constructed in the target software, which means that fuzzy self-adaptation ability is added into the target software.

Thanks to AOP, the source codes of the target software does not have to be rewritten, and the fuzzy self-adaptation loop is made visible and explicit in the target software with the help of aspects.

The basic elements and concepts in the implementation framework shown in Fig.9 are elaborated as below.

FSAL Editor and Tester. The FSAL (fuzzy self-adaptation logic) editor and tester is a software tool for editing and testing fuzzy self-adaptation logic, and it can automatically convert the compilation of fuzzy self-adaptation logic into a fuzzy adaptor aspect.

Since fuzzy logic may be new to many software engineers, here we will provide the FSAL editor and tester tool to reduce the difficulty that they might be confronted with in the process of developing fuzzy self-adaptive software. This tool has friendly GUI supporting key fuzzy operations, such as fuzzification, fuzzy self-adaptation inference and defuzzification. Specifically, it can define fuzzy linguistic variables and terms, compile fuzzy self-adaptation rules, and specify fuzzy self-adaptation inference algorithms. While finishing the editing of FSAL, the tool can transparently test the edited FSAL to check the correctness of the logic. Finally, according to the current compilation of FSAL, the tool can generate a fuzzy adaptor aspect meeting the specifications of AspectC++. We have realized the FSAL editor and tester tool by extending an open-source fuzzy inference system, Fuzzy Lite[2], with AOP abilities.

Aspect Generator. Aspect generator is a software environment for editing (C++) aspects. In our implementation framework, sensor aspects and actuator aspects in the fuzzy self-adaptation loop are edited and generated with the software tool. Moreover, fuzzy adaptor aspects can be edited directly and generated in the environment if users are familiar with fuzzy mathematics. In the framework, Eclipse with the ACDT plug-in[3] or Microsoft Visual Studio .Net with AspectC++ AddIn[3] can be employed as the aspect generator.

Fuzzy Adaptor Aspect. A fuzzy adaptor (FA) aspect is a programming module responsible for realizing fuzzy self-adaptation logic. The code of a fuzzy adaptor aspect depends on the type of the fuzzy self-adaptation logic specified in the FASL editor and tester. Definitions of input and output variables, parameters of membership functions, fuzzy self-adaptation rules, operation algorithms of fuzzy self-adaptation inference and defuzzification, and other core elements related to fuzzy self-adaptation inference are all encapsulated in an FA aspect. In our framework (Fig.9), an FA aspect can be converted automatically using the FASL editor and tester. It can also be edited and generated manually using the aspect generator. At the implementation level, we realize a fuzzy adaptor into a class using OO methods and then introduce the FA class into an aspect to produce an FA aspect, which not only provides a clear program structure but also results in the good scalability of FA aspects. Additionally, an FA aspect woven in the target software is executed with main or other threads of the target software specified by *pointcuts* in the aspect.

Fig.10 shows the basic programming model of an FA aspect. The second element in this structure, named

```
aspect Fuzzy Adaptor {
    pointcut controlling()=execution ("% ControlPosition(...)");
    advice controlling(): after(){
    OXFuzzyAdaptor fa;
    FuzzyInputs=fa.fuzzify (GlobalRTDatabase SenseValues);
    fa.ConstructRuleBase();
    FuzzyOutputs=fa.FuzzyInference (FuzzyInput);
        CrispOutputs=fa.Defuzzify(FuzzyOutputs);
        GlobalRTDatabase.ActuateValues=CrispOutputs;
    }
}
```

Fig.10. Basic programming model of an FA aspect.

[2]http://code.google.com/p/fuzzy-lite, March 2011.

[3]http://www.aspectc.org, March 2011.

pointcut *controlling*, has the effect of identifying the execution of *ControlPosition* operation in the target software as a joining point (lines 2 and 3). *ControlPosition* denotes the name of a specified function (method) or variable statement in the source code of the target software. In the *controlling after* advice, a fuzzy adaptor object *fa* is instanced to make self-adaptation decisions by performing basic fuzzy operations, such as fuzzifying, constructing a rule base and defuzzifying (lines 5∼10). We also realize a global real-time database (i.e., GlobalRTDatabase) to exchange data among fuzzy adaptor objects, sensor objects and actuator objects (lines 6 and 11).

Sensor Aspect. Sensor aspects are software modules for sensing software contexts in runtime. A special code to probe software internal states (e.g., values of global variables) or software external environment (e.g., CPU utilization of OS and temperature of the physical environment) is encapsulated in a sensor aspect. Our implementation framework has an aspect database that provides common sensor aspects, such as the CPU utilization sensor aspect and the memory sensor aspect, for software fuzzy self-adaptation. If a sensor is not available in the aspect database, it can be generated using the aspect generator. Similar to the implementation of FA aspects, a sensor aspect is often realized by introducing a sensor class into an aspect. In our implementation framework, a general programming model of sensor aspects includes one aspect named pointcut *sensing*, which identifies the execution of *SensePosition* operation in the target software as a joining point (lines 2 and 3 in Fig.11). Similar to *ControlPosition*, *SensePosition* denotes another specified function (method) or variable statement in the source code of the target software.

```
1:  aspect Sensor{
2:     pointcut sensing()=execution ("%SensePosition-
3:     (...)");
4:     advice sensing: after(){
5:     OXSensor sensor;
6:     GlobalRTDatabase.SenseValues
7:     =sensor.GetValues();
8:     }
9:  }
```

Fig.11. Basic programming model of a sensor aspect.

Actuator Aspect. An actuator aspect is a special software module for adjusting the parameters, behaviors and structures of the target software. Similar to sensor aspects, common actuator aspects are also encapsulated and stored in an aspect database for easy use. Moreover, the aspect generator is an alternative tool for editing and generating actuator aspects. The general programming model of an actuator aspect consists of one named pointcut *actuating*, which identifies the execution of *ActuatePosition* operation in the target software as a joining point (lines 2 and 3 in Fig.12). Similar to *SensePosition*, *ActuatePosition* is the name of a specified function (method) or variable statement in the source code of the target software.

```
1:  aspect Actuator{
2:     pointcut actuating()=execution ("% ActuatePosi-
3:     tion(...)");
4:     advice actuating: after(){
5:     OXActuator actuator;
6:     OutputValues=GlobalRTDatabase.ActuateValues;
7:     actuator.Output (OutputValues);
8:     }
9:  }
```

Fig.12. Basic programming model of an actuator aspect.

Aspect Weaver. An aspect weaver is a software tool to weave FA aspects, sensor aspects, and actuator aspects into the target software. In our framework, the aspect weaver is implemented with Aspect C++ Weaver.

AOP-Based SFSA Loop. An AOP-based SFSA loop is produced when an FA aspect with a sensor aspect and an actuator aspect is woven into the target software. If the target software has multiple self-adaptation goals, several SFSA loops may exist in the same target software. These SFSA loops may be driven and run individually in different threads of the target software. Experience from the control community tells us that a few of SFSA loops in the same software can work reliably if self-adaptation goals are not against each other. Furthermore, an SFSA loop in the target software can stay not only within a component but also across two or three components if called for (Fig.9). For example, an SFSA loop crosses several components when a sensor aspect in component *a* collects states of component *a* for an FA aspect in component *b*, and the decisions made by the FA aspect are put into effect on component *c* through an actuator aspect in component *c*.

In the SFSA implementation architecture, the FSAL editor and tester tool, the aspect generator tool, and the aspect weaver tool are included in the SFSA toolkit, which provides an integrated solution to automate the development of fuzzy self-adaptation of MCS and even general software. However, MCS in some applications may need to be real time, while our proposed implementation approach do not consider the real-time constraints yet. We will explore this issue in further research.

7 Evaluation

This part of the paper describes a series of experiments that evaluate the effectiveness and efficiency of the SFSA approach by applying it to Lon893OPCServer introduced as a running example in the first part of this paper. First, we give a detailed description of the implementation of the SFSA loop for Lon893OPCServer in Subsection 7.1. To verify the effectiveness of our approach, we test how well the SFSA-enabled version of Lon893OPCServer adapts to changes in its contexts in Subsection 7.2. Moreover, to explore the efficiency of our SFSA approach, we measure its overheads for the SFSA-enabled Lon893OPCServer in Subsection 7.2.

7.1 Implementation of SFSA Loop for the Running Example

7.1.1 Implementation

As mentioned earlier, one self-adaptation objective of Lon893OPCServer in the running example is *automatically adjusting its CPU utilization to maintain a running environment with proper CPU utilization* (see Example 5). Thus, to achieve this objective, we establish a fuzzy self-adaptation loop for the MCS under the guidance of our SFSA conceptual framework. This loop is displayed in Fig.13, which consists of a fuzzy adaptor, a CPU utilization sensor and a *ScanPeriod* actuator.

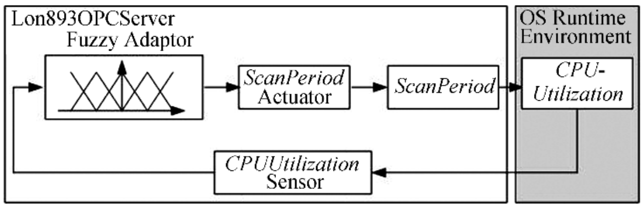


Fig.13. Fuzzy self-adaptation loop of the running example.

We realize the fuzzy self-adaptation loop using the proposed AOP-based implementation technology and with the help of the SFSA toolkit. Three classes, namely, *OCPUSensor*, *OFuzzyAdaptor* and *OScanPeriodActuator*, are produced with the toolkit. Every class is encapsulated in an aspect and woven into the source code of Lon893OPCServer.

In the OCPUSensor class, a method called *GetCpuValue* is defined to collect current CPU utilization from the OS runtime environment. Similarly, in the *OScanPeriodActuator* class, we define a method called *Output* to support modifying the *ScanPeriod* property of the global data structure *DeviceRTValue* in the Lon893OPCServer. The *OFuzzyAdaptor* class provides three main operations: fuzzification operation for input data from the sensor class, fuzzy inference for the decision, and defuzzification operation for the output data to the actuator class. Moreover, the class *ORTDatabase* is produced as a global real-time database for data exchange among *OFuzzyAdaptor*, *OCPUSensor* and *OScanPeriodActuator*.

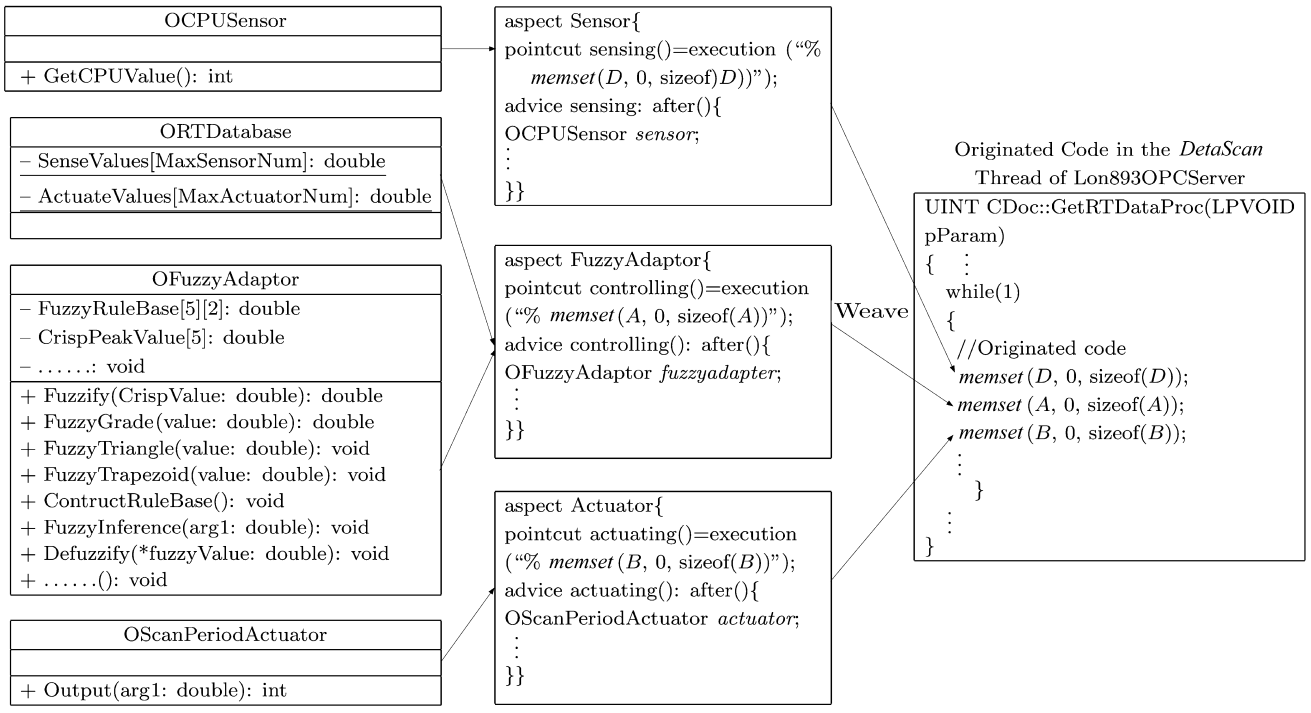
Corresponding to the above classes, three aspects, namely, the *FuzzyAdaptor* aspect, the *Sensor* aspect and the *Actuator* aspect, are generated with the toolkit to perform the task of weaving the SFSA loop into Lon893OPCServer. We identify three sequent program statements, which from the *DataScan* thread of Lon893OPCServer, as jointpoints defined in the pointcuts of these aspects. The AOP-based implementation process for achieving the self-adaptation ability of the running example is illustrated in Fig.14.

As seen in Fig.14, the fuzzy self-adaptation loop, including the three aspects attached with the four classes are woven into the *DataScan* thread of Lon893OPCServer. In the thread, advice from the *Sensor* aspect is executed after the execution of the statement $memset(D, 0, \text{sizeof}(D))$. Advice from the *FuzzyAdaptor* aspect is executed after the execution of the statement $memset(A, 0, \text{sizeof}(A))$. Advice from the *Actuator* aspect is executed after the execution of the statement $memset(B, 0, \text{sizeof}(B))$. Additionally, there is a while loop to ensure that the fuzzy self-adaptation loop runs ceaselessly.

Therefore, with the help of our AOP-based implementation framework, a sense-decide-act loop is established in Lon893OPCServer without any modification of its original source codes.

7.1.2 Time Complexity Analysis

Compared with the old version, the SFSA-enabled Lon893OPCServer has an additional fuzzy self-adaptation loop. In the loop, the main performance cost may come from five operations in three software classes: *GetCPUValue* operation in the *CSensor* class; the operations of fuzzification, fuzzy inference, and defuzzification in the *CFuzzyAdaptor* class; and the *Output* operation in the *CActuator* class. In the program codes which implement the two operations *GetCPUValue* and *Output*, there are only several API invoking statements, so their time complexity is $O(1)$. In the fuzzification operation, where the membership function is implemented, the time complexity is also $O(1)$. The implementations of the two operations, fuzzy inference and defuzzification, involve a *for-loop* individually, so each of their time complexity is $O(n)$. In sum, the total time complexity of the fuzzy self-adaptation in our Lon893OPCServer is $3O(1) + 2O(n)$. Generally, $O(1)$ is a constant that may be very large or very small. This depends much on the real computation task. As a result, the execution time of APIs used in sensors or actuators should be rationally considered when implementing an SFSA loop according to its application



```
OCPUSensor
+ GetCPUValue(): int

ORTDatabase
– SenseValues[MaxSensorNum]: double
– ActuateValues[MaxActuatorNum]: double

OFuzzyAdaptor
– FuzzyRuleBase[5][2]: double
– CrispPeakValue[5]: double
– ......: void
+ Fuzzify(CrispValue: double): double
+ FuzzyGrade(value: double): double
+ FuzzyTriangle(value: double): void
+ FuzzyTrapezoid(value: double): void
+ ContructRuleBase(): void
+ FuzzyInference(arg1: double): void
+ Defuzzify(*fuzzyValue: double): void
+ ......(): void

OScanPeriodActuator
+ Output(arg1: double): int
```

```
aspect Sensor{
pointcut sensing()=execution ("%
   memset(D, 0, sizeof)D))");
advice sensing: after(){
OCPUSensor sensor;
 ⋮
}}

aspect FuzzyAdaptor{
pointcut controlling()=execution
("% memset(A, 0, sizeof(A))");
advice controlling(): after(){
OFuzzyAdaptor fuzzyadapter;
 ⋮
}}

aspect Actuator{
pointcut actuating()=execution
("% memset(B, 0, sizeof(B))");
advice actuating(): after(){
OScanPeriodActuator actuator;
 ⋮
}}
```

```
UINT CDoc::GetRTDataProc(LPVOID
pParam)
{  ⋮
  while(1)
  {
   //Originated code
   memset(D, 0, sizeof(D));
   memset (A, 0, sizeof(A));
   memset(B, 0, sizeof(B));
   ⋮
     }
  ⋮
}
```

Fig.14. AOP-based implementation process for the self-adaptation of the running example.

domain. In the Lon893OPCServer, the real execution time of the operations (i.e., the *GetCPUUtilization* API plus the statements used to modify the *ScanPeriod* variable) related to $O(1)$ is about a few dozens of milliseconds. Therefore, $O(1)$ is very small, and can meet the requirements of embedded computing scenarios.

Therefore, we can conclude that the additional fuzzy self-adaptation loop is lightweight and with little additional performance cost for Lon893OPCServer.

7.2 Experiments and Analysis

We will then look at how the SFSA-enabled version of Lon893OPCServer adapts to changes in its contexts and estimate the overheads incurred by the SFSA framework proposed.

We experiment with MCS using a LonWorks fieldbus system. The architecture of the experiment system is illustrated in Fig.15. This experiment system consists

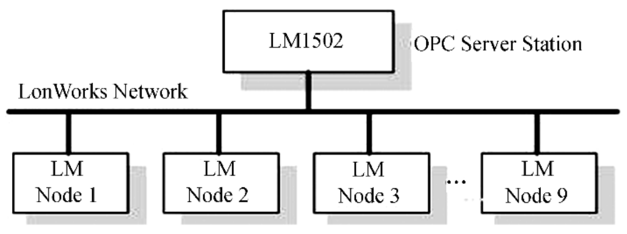


Fig.15. Architecture of the experiment system.

of one OPC server station (LM1502) and nine LonWorks I/O nodes (three nodes of LM1101, three nodes of LM1102, and three nodes of LM1204). As the running platform of Lon893OPCServer, the OPC server station is an embedded computer, which is configured with PCM5823 mainboard, Intel 1 G MHz CPU, and 512 MB memory. The OS of the station is Windows XP Embedded.

The first experiment compares the CPU utilization of OPC server stations of the old version of Lon893OPCServer and the SFSA-enabled one (also called FuzzyLon893OPCServer) under the same disturbance from contexts. In this experiment, we perform two executions on the same experiment platform. One is for Lon893OPCServer, and the other is for SFSA-enabled Lon893OPCServer. Both of the execution processes are accompanied by the same disturbance from the running of LMManager software (a debug software tool for the LonWorks network, which always consumes much CPU resources when accessing the fieldbus network). The 10-minute-range measurements for the two executions are illustrated in Fig.16.

During the running process of Lon893OPCServer with no self-adaptation, at time $t = 1$, the LMManager tool starts to run, causing a sudden increase in the CPU utilization of the OPC server station from approximately 20% to nearly 60%. CPU utilization drops and finally stabilizes at more or less 50%

(Fig.16). Meanwhile, for the SFSA-enabled version of Lon893OPCServer, while facing the same disturbance from the LMManager tool, the CPU utilization of the OPC server station stabilizes at around 30% after a brief increase. This means that the SFSA-enabled Lon893OPCServer can reduce effectively the CPU load by as much as 20%.

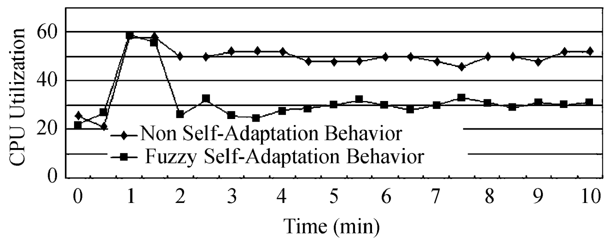


Fig.16. Lon893OPCServer without self-adaptation and with fuzzy self-adaptation.

This experiment shows that compared with the old version, under the same intervention, the SFSA-enabled Lon893OPCServer exhibits better self-adaptation ability when facing sudden changes in its contexts.

In the second experiment, to further estimate the effectiveness of the fuzzy control based self-adaptation method. We compare it with the classical PID (proportional, integral, and differential) control based approach because PID control is a very popular method of dealing with linear and determinate control problems and has been frequently used by several researchers[11-13] to settle software self-adaptation issues. A detailed explanation of PID control can be found in [36]. We implement the PID-based self-adaptation loop with the AOP-based framework. However, the PID control policy has to be coded manually because our programming framework only supports fuzzy control.

In the experiment, the basic "text-book" PID controller is chosen. Its difference equation is

$$\begin{aligned}u(k) = & K_P e(k) + K_I \sum_{j=0}^{k} e(j) + \\ & K_D[e(k) - e(k-1)],\end{aligned} \tag{12}$$

where k denotes the number of discrete sampled time sequence, $u(k)$ the control output (i.e., *ScanPeriod*) at the time k, $e(k)$ the error between the expected plant output and the real plant output at the time k, $e(k-1)$ the error at the time $(k-1)$, and K_P is the proportional gain, K_I the integral gain and K_D the derivative gain. Here, the control output is the scan period of Lon893OPCServer, and the plant output is the CPU utilization of the OPC server station. In this experiment, we set the value of the expected plant output at 30%, which implies that CPU utilization of the station is expected to stabilize at about 30% when disturbances occur.

We have to specify the PID parameters in the self-adaptation loop intuitively and experimentally because we cannot determine the accurate mathematical model to describe the relationship between the CPU utilization of OS and the *ScanPeriod* parameter of Lon893OPCServer. Here, we specify $P = 1.5$, $I = 0.8$, and $D = 1.2$. We run the PID-enabled Lon893OPCServer in the same experiment platform and the same disturbance with the SFSA-enabled version. Fig.17 shows the measurement of the execution process.

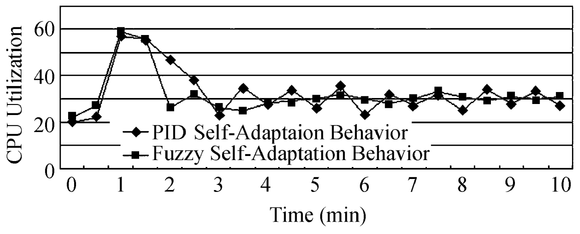


Fig.17. Comparison between PID self-adaptation and fuzzy self-adaptation behaviors of Lon893OPCServer.

As seen in Fig.17, although the PID-enabled Lon893OPCServer can reduce the CPU utilization to about 30% after a short increase, there are still two deficiencies when compared with the SFSA-enabled Lon893OPCServer. First, the time it takes for CPU utilization to drop to a steady value (about 30%) in the PID-enabled version is longer than in the SFSA-enabled one. Second, the CPU utilization adapted by the PID-enabled version tends to be oscillatory due to the intuitively specified PID parameters. This experiment shows that the SFSA-enabled Lon893OPCServer enjoys better self-adaptation capability than the PID-enabled one. It further shows that the fuzzy control based self-adaptation policy is more robust than the PID control based one under uncertain environment. Although one might possibly find a PID controller that performs better than the fuzzy controller, real-world practices tell us that this finding process often spends too much effort and time. However, our proposed approach is very intuitive and understandable, and can achieve good-enough self-adaptation goals with less effort and time. Besides, experience with fuzzy controllers from the control community has shown that they are often more robust and stable than PID controllers[37].

In the third experiment, we estimate the overheads incurred by the SFSA loop in Lon893OPCServer. The experiment assesses overheads from time and space perspectives respectively. To explore the time overhead, we measure the total execution time of the SFSA loop in Lon893OPCServer through the time-measuring API function (i.e., *GetTickCount*). We sample 10 groups of the execution time data at an interval of 30 seconds. We find that each execution time in the 10 groups is 0 ms. Of course, the actual execution time of the SFSA loop is impossible to be 0. Here, 0 means the actual execution time is much shorter than 0.5 ms. Therefore, the time overhead of the SFSA loop is rather brief and can even be neglected, which is an evidence for the theoretical analysis of time complexity in Subsection 7.1.2.

To explore the space overload of the SFSA loop, we compare the memory cost of the SFSA-enabled Lon893OPCServer with that of the old one. The cost is measured with the TaskManager tool in the Windows OS. The SFSA-enabled Lon893OPCServer and the old one are started with one, three and five process instances separately. For each round of running, the memory cost of every instance of the Lon893OPCServer is measured and the average memory cost of all the running instances is computed. Fig.18 shows the comparison of the average memory cost, which varies in accordance with the number of process instances, for the old Lon893OPCServer (No SFSA-Loop) and the SFSA-enabled one.

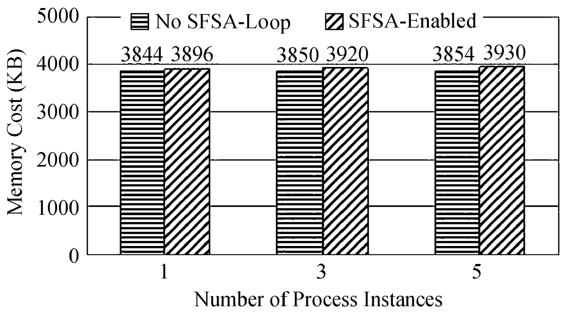


Fig.18. Impact of SFSA-loop to the memory cost of Lon893OPCServer.

Fig.18 shows that little memory cost is incurred in the SFSA loop. When a single process instance runs, the memory cost increases by 1.4% (from 3 844 KB to 3 896 KB). When three process instances run, the memory cost increases by 1.8% (from 3 850 KB to 3 920 KB). When the number of process instances is up to five, the memory cost increases by 1.9% (from 3 854KB to 3 930 KB). The third experiment shows that the SFSA framework brings about only a negligible amount of overheads.

In sum, the above experiments have corroborated that the SFSA framework proposed is both effective and economical.

7.3 Discussion

Looking back upon the whole experiment process, we realize that our study still suffers from some limitations. First, the fuzzy control-based framework belongs to the set of internal approaches that needs to intertwine application logic with self-adaptation logic. Therefore, there is a precondition that we should obtain in the first place the source codes of the target software. Second, due to restrictions from the current version of Aspect C++, our approach has to lend a static support to the weaving of the self-adaptation loop into the target software. Finally, more complicated self-adaptation scenarios of MCS need to be explored to demonstrate the competence of our approach.

8 Related Work

The field of software self-adaptation has been explored by many researchers. We recommend a reading of [3-4, 22] for a comprehensive understanding of the current research status of software self-adaptation. This paper is a follow-up study of our previous studies[27-28]. Compared with them, this paper has obtained significant improvements. First, it proposed an optimized conceptual framework for software fuzzy self-adaptation (SFSA), and established a formal model of SFSA. Second, the SFSA programming model and its supporting tool — SFSA toolkit were developed to automate the realization process of SFSA. Last but not least, we demonstrated our approach through the development of an adaptive MCS application in process control systems, and did more validation experiments to check whether the proposed fuzzy-control-based approach is effective and low in overheads.

In this section, we review studies that are of great relevance to our SFSA approach.

We first look at the existing feedback control based approach for software self-adaptation. Some studies in the research of uncertainty in self-adaptive software are surveyed. Furthermore, some recent research efforts on the implementation techniques of software self-adaptation are presented.

8.1 Control Theory Based Approach

Generally, the control theory is concerned with physical systems that use a sense-evaluate-act loop to interact with the world repeatedly. Since it shares the same feedback ideas with self-adaptive software, the control theory has been employed by a few of researchers as an important paradigm to address issues in software self-adaptation.

Current control theory based approaches for self-adaptation mostly pay attention to the side of control, not to the side of software. For instance, Kokar *et al.*[10] regarded the software system as a plant to be controlled based on modeling the plant and the environment as a dynamic system, and proposed such three self-adaptation models as feedback loop, adaptation loop and reconfiguration loop in terms of control models, which considers few features of software. Shen *et al.*[11] presented three software adaptive models derived from open-loop control, feedback control and adaptive control, and provided an application server framework to support the implementations of these three models.

Moreover, existing control-based paradigms often need to establish a model of the software plant. For example, Litoiu *et al.*[38] proposed a hierarchical model based control architecture for software systems to solve problems of scope and timescale differences in the adaptations. Diao *et al.*[12] used a PID controller to handle the dramatic performance degradation of an IBM database management product. On the other hand, considering the discrete nature of software, the supervisory control theory was employed to model software self-adaptation[14,39-40]. Some work started to combine the concepts of control and software. An example is ControlWare[41], a middleware QoS-control architecture based on the control theory, to realize performance assurance.

Our fuzzy control based approach is different from the foregoing approaches. First, the above approaches fail to consider technologies dealing with uncertainty, which is becoming a challenging problem for software systems. Second, most of the above approaches are based on the classical control theory, and they entail much professional knowledge of the control theory (e.g., PID). An inadequate understanding of the classical control theory always leads to unfavorable self-adaptation effects. In contrast, our approach is based on fuzzy logic, which is easy to be grasped and even familiar to some software researchers. Moreover, most of the above approaches simply treat software systems as controlled objects and do not consider the internal structure of software. Our approach provides programming models and the SFSA toolkit for bridging the gap between software engineering and control engineering, so that it debases the development difficulty of software self-adaptation.

Additionally, our fuzzy rule based approach differs from existing rule-based approaches[42], which lack formal logical or mathematical foundations and are subject to more rules than the one in our work when dealing with the same problems.

8.2 Approaches to Handling Uncertainty

The uncertainty problem in software self-adaptation has also been noticed by several researchers[43-46]. The discussions in this subsection are organized with the uncertainty taxonomy proposed.

On the aspect of internal uncertainty, Cheng *et al.*[42] described three specific sources of uncertainty, namely, problem-state identification, strategy selection and strategy success or failure, in their architecture-based self-adaptation framework called Rainbow[21]. They employ the probabilistic distribution technique to handle the three sources of uncertainty. Different from their approach, we use the fuzzy theory to deal with uncertainty in the self-adaptation loop. Our proposed approach offers a natural-language style of presenting self-adaptation logic and making self-adaptation decisions, making it easy to use and understand. Moreover, Esfahani *et al.*[46] described a POISED approach for tackling the challenge posed by uncertainty in making adaptation decisions. POISED relies on the possibility theory in assessing both the positive and negative consequences of uncertainty. Our proposed SFSA approach aims to engineer the software self-adaptation loop under uncertainty through the fuzzy control theory. Furthermore, we provide a set of engineering methods (SFSA conceptual framework, formal model and programming model) and automation tools (SFSA toolkit) to handle uncertainty in the software self-adaptation loop.

On the aspect of requirement uncertainty, Cheng *et al.*[44] introduced a goal-based modeling approach to develop the requirements for a dynamically adaptive systems and proposed the RELAX specification language to incorporate uncertainty into the specification of self-adaptive systems. Our work focuses on settling uncertainty, especially the fuzziness existing in the self-adaptation loop. Moreover, our approach pays much attention to the design and implementation phase of self-adaptive software.

On the aspect of environmental uncertainty, Gmach *et al.*[45] used a fuzzy controller for adaptive enterprise service management to remedy exceptional situations by automatically initiating service migration and replication. The work finds itself short of systematic methods to deal with the self-adaptation issue. Compared with our work, it simply applies the fuzzy control theory to manage and balance the resources of enterprise service; no supporting software methodology or tools are found.

8.3 AOP-Based Implementation Techniques

AOP-based implementation techniques for weaving self-adaptation facilities into target software have been explored by a few researchers. Based on the style of weaving the self-adaptation loop, current weaving approaches can be divided into two kinds: partially or fully weaving.

Following partially weaving approaches, Wu *et al.*[32] conducted a case study on implementing self-adaptive software architecture by dynamic AOP in an industrial web-based system. This approach only allows weaving self-adaptation policies into target component-based systems. Chan *et al.*[47] introduced an AOP-based approach to build generic monitoring systems for legacy applications through weaving sensors into target systems.

Under fully weaving approaches, Yang *et al.*[33] presented an AOP-based approach for dynamic adaptation, including a systematic process for defining where, when and how adaptation logic is fully incorporated into target software. One of the limitations of this approach lies in the lack of a definition of generic aspects for the self-adaptation loop, which makes self-adaptation logic inexplicit. Janik *et al.*[34] presented an approach to implementing an adaptability loop in autonomic computing (AC) systems on the basis of adaptable aspects. The application is instrumented with aspects selected from a set of all available aspects (sensors, effectors, and goal aspects) in such a way that the system can monitor and manage the application.

There are differences between our proposed approach and the aforementioned AOP-based implementation techniques. First, most of the above implementation techniques are Java-based, so that they can hardly be used in MCS, which is often realized with C/C++ language. Second, our approach provides complete supporting tools for automating the implementation process of software self-adaptation. Finally, our approach combines AOP with OOP technology to construct the self-adaptation loop, making the loop fairly explicit and visible in the target software.

9 Conclusions and Future Work

Inspired by mission-critical applications under uncertainty, we have proposed a fuzzy control based approach to achieving software self-adaptation. We have presented the SFSA conceptual framework and discussed the three stages (sensing stage, deciding stage and acting stage) and three software entities (software sensors, fuzzy adaptors and software actuators) in the framework. To show the strictness of our approach, we have established the formal model of SFSA, which offers a mathematical support for the application of our proposed approach in MCS. A novel SFSA implementation technology involving programming models for SFSA loops based on AOP has been proposed for realizing and engineering the SFSA conceptual framework into real-world software systems. We also provided the SFSA toolkit for generating aspects automatically in SFSA loops and automating the development of software self-adaptation. The results of our evaluation experiments with the running example show that our proposed fuzzy control based approach is an effective approach with low overheads for achieving self-adaptation of MCS.

The applicability of our fuzzy control based approach is not limited to MCS. It can be generalized to most C/C++-based software systems.

We envision an analysis of the stability of the self-adaptation process as a promising direction in the future since it might direct and optimize the design of fuzzy self-adaptation logic.

Acknowledgement The authors are very grateful to the anonymous reviewers for their insightful remarks and helpful comments on an earlier version. The authors also would like to give credit to Juan Rada-Vilela for his work in developing Fuzzy Lite, the free and open-source fuzzy logic inference system. Thanks a lot to Hong-Wei Xie in Nanjing University for helping us develop the FSAL Editor and Tester tool.

References

[1] Fowler K, Mission-critical and safety-critical development. *IEEE Instrumentation & Measurement*, 2004, 7(4): 52-59.

[2] Caslelli V, Harper R E, Heidelberger P *et al.* Proactive management of software aging. *IBM Journal of Research and Development*, 2001, 45(2): 311-332.

[3] Cheng B H C, Lemos R D, Giese H *et al.* Software engineering for self-adaptive systems: a research roadmap. In *Software Engineering for Self-Adaptive Systems*, B H Cheng *et al.* (eds.), Springer-Verlag, 2009, pp.1-26.

[4] Kramer J, Magee J. Self-managed systems: An architectural challenge. *In Proc. the 2007 International Conference on Software Engineering*, May 2007, pp.259-268.

[5] Li D Y, Liu C Y, Du Y, Han X. Artificial intelligence with uncertainty. *Journal of Software*, 2004, 15(11): 1583-1594. (In Chinese)

[6] Zadeh L A. Fuzzy sets. *Information and Control*, 1965, 8(3): 338-353.

[7] Zadeh L A. Fuzzy logic. *IEEE Computer*, 1988, 21(4): 83-93.

[8] Zadeh L A. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1999, 100(Supplement 1): 9-34.

[9] Cai K Y, Cangussu J W, DeCarlo R A, Mathur A P, An overview of software cybernetics. In *Proc. the 11th Annual International Workshop on Software Technology and Engineering Practice*, Sept. 2003, pp.77-86.

[10] Kokar M M, Baclawski K, Eracar Y A. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 1999, 14(3): 37-45.

[11] Shen J, Wang Q, Mei H. Self-adaptive software: Cybernetic perspective and an application server supported framework. In *Proc. the 28th Annual International Computer Software and Applications Conference*, Sept. 2004, pp.92-95.

[12] Diao Y, Hellerstein J L, Parekh S *et al.* A Control theory foundation for self-managing computing systems. *IEEE Journal on Selected Areas in Communications*, 2005, 23(12): 2213-2222.

[13] Peng X, Chen B, Yu Y, Zhao W. Self-tuning of software systems through goal-based feedback loop control. In *Proc. the 18th International Conference on Requirements Engineering*, Sept. 27-Oct. 1, 2010, pp.104-107.

[14] Phoha V V, Nadgar A U, Ray A, Phoha S. Supervisory control of software systems. *IEEE Transactions on Computers*, 2004, 53(9): 1187-1199.

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

[16] Wang L X. A Course in Fuzzy Systems and Control. Prentice Hall, 1996.

[17] Passino K M, Yurkovich S. Fuzzy Control. Addison-Wesley Longman, 1997.

[18] Lakin L I. A fuzzy controller for aircraft control systems. In *Industrial Applications of Fuzzy Control*, Sugeno M (eds.), Northholland, Amsterdam, 1985, pp.87-104.

[19] Liu B, Tang W. Modern Control Theory (3rd edition). Beijing: China Machine Press, 2006. (In Chinese)

[20] Yang Q, Xing J, Wang P. Design and implementation of the OPC server oriented to one LonWorks network. *Computer Engineering*, 2007, 33(2): 228-230. (In Chinese)

[21] Liu J, Lim K W, Ho W *et al.* Using the OPC standard for real-time process monitoring and control. *IEEE Software*, 2005, 22(6): 54-59.

[22] Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 2009, 4(2): Article No.4.

[23] Cheng S W. Rainbow: Cost-effective software architecture-based self-adaptation [Ph.D. Thesis]. Carnegie Mellon University, USA, 2008.

[24] IBM. An architectural blueprint for autonomic computing (4th Edition). Technical Report. June 2006.

[25] Oreizy P, Gorlick M M, Taylor R N *et al.* An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 1999, 14(3): 54-62.

[26] Floch J, Hallsteinsen S, Stav E *et al.* Using architecture models for runtime adaptability. *IEEE Software*, 2006, 23(2): 62-70.

[27] Yang Q L, Lv J, Li J L, Ma X X *et al.* Towards a fuzzy-control-based approach to design of self-adaptive software. In *Proc. the 2nd Asia-Pacific Symposium on Internetware*, Nov. 2010, Article No.15.

[28] Yang Q L, Lv̈ J, Xing J C *et al.* Fuzzy control-based software self-adaptation: A case study in mission critical systems. In *Proc. the 35th Annual IEEE Conference on Computer Software and Applications*, July 2011, pp.13-18.

[29] Subramanian N. Adaptable software architecture generation using the NFR approach [Ph.D. Thesis]. University of Texas at Dallas, USA, May 2003.

[30] Garlan D, Cheng S W, Huang A C *et al.* Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 2004, 37(10): 46-54.

[31] Manoel E, Nielsen M J, Salahshour A *et al.* Problem determination using self-Managing autonomic technology. *IBM redbooks*, June 2005.

[32] Wu Y, Wu Y, Peng X *et al.* Implementing self-adaptive software architecture by reflective component model and dynamic AOP: A case study. In *Proc. the 10th International Conference on Quality Software*, July 2010, pp.288-293.

[33] Yang Z, Cheng B H C, Stirewalt R E K *et al.* An aspect-oriented approach to dynamic adaptation. In *Proc. the 1st Workshop of Self-Healing Software*, Nov. 2002, pp.85-90.

[34] Janik A, Zielinski K. Adaptability mechanisms for autonomic system implementation with AAOP. *Software Practice and Experience*, 2010, 40(3): 209-223.

[35] Kiczales G, Lamping J, Mendhekar A, Maeda C *et al.* Aspect-oriented programming. In *Proc. the 11th ECOOP*, June 1997, pp.220-242.

[36] Ang K H, Chong G, Li Y. PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 2005, 13(4): 559-576.

[37] McNeill F M, Thro E. Fuzzy Logic: A Practical Approach, Morgan Kaufmann Pub, 1994.

[38] Litoiu M, Woodside M, Zheng T. Hierarchical model-based autonomic control of software systems. In *Proc. Workshop on Design and Evolution of Autonomic Applications Software*, May 2005, pp.1-7.

[39] Tziallas G, Theodoulidis B. A controller synthesis algorithm for building self-adaptive software. *Information and Software Technology*, 2004, 46(11): 719-727.

[40] Karsai G, Ledeczi A, Sztipanovits J *et al.* An approach to self-adaptive software based on supervisory control. In *Proc. the 2nd International Workshop on Self-Adaptive Software*, May 2001, pp.24-38.

[41] Zhang R, Lu C, Abdelazher T F *et al.* ControlWare: A middleware architecture for feedback control of software performance. In *Proc. the 22nd International Conference on Distributed Computing Systems*, July 2002, pp.301-310.

[42] Wang Q X. Towards a rule model for self-adaptive software. *ACM SIGSOFT Software Engineering Notes*, 2005, 30(1): 1-5.

[43] Cheng S W, Garlan D. Handling uncertainty in autonomic Systems. *In Proc. 2007 International Workshop on Living with Uncertainties*, November 2007.

[44] Cheng B H, Sawyer P, Bencomo N. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proc. the 12th International Conference on Model Driven Engineering Languages and Systems*, Oct. 2009, pp.468-483.

[45] Gmach D, Krompass S, Scholz A *et al.* Adaptive quality of service management for enterprise services. *ACM Transactions on the Web*, 2008, 2(1): Article No.8.

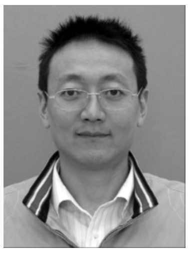
[46] Esfahani N, Kouroshfar E, Malek S. Taming uncertainty in self-adaptive software. In *Proc. the 19th ACM SIGSOFT Internal Symposium on the Foundations of Software Engineering*, Sept. 2011, pp.234-244.

[47] Chan H, Chieu T C. An approach to monitor application states for self-managing (autonomic) system. In *Proc. the 18th ACM Sigplan Conference on Object-Oriented Programming, Systems, Languages, and Applications.* Oct. 2003, pp.312-313.

Qi-Liang Yang is currently a Ph.D. candidate in the Department of Computer Science at Nanjing University. He is also an associate professor in PLA University of Science and Technology, Nanjing. His research interests include self-adaptive software systems, mission-critical system and software, and pervasive computing. He is a member of CCF and IEEE.

Jian Lv received the B.Sc., M.Sc., and Ph.D. degrees in computer science from Nanjing University in 1982, 1984, and 1988, respectively. He is currently a professor in the Department of Computer Science at Nanjing University. He is also the director of the State Key Laboratory for Novel Software Technology and the vice director of the Institute of Software Technology at Nanjing University. His research interests include programming methodology, pervasive computing, software agent, and middleware. He is a member of ACM and a fellow of CCF.

Xian-Ping Tao received the M.Sc. and Ph.D. degrees in computer science from Nanjing University in 1994 and 2001, respectively. He is currently a professor in the Department of Computer Science at Nanjing University. His research interests include software agents, middleware systems, Internetware methodology, and pervasive computing. He is a member of CCF and IEEE.



Xiao-Xing Ma received the Ph.D. degree in computer science from Nanjing University in 2003. He is a professor at Nanjing University. His research interests include middleware systems, software architecture and self-adaptive software systems.

Jian-Chun Xing received his B.Sc. and M.Sc. degrees in electric system and automation from Engineering Institute of Engineering Corps, China, in 1984 and 1987, respectively, and the Ph.D. degree in information system engineering from PLA University of Science and Technology (PLA UST), Nanjing, in 2006. He is currently a professor in PLA UST. His research interests include intelligent control, artificial intelligence and information processing. He is a member of CCF and IEEE.

Wei Song received the Ph.D. degree in computer software and theory from Nanjing University in 2010. He is now a lecture and master supervisor at Nanjing University of Science and Technology. His research interests include services computing, software engineering, and program analysis. He is a member of CCF, ACM and IEEE.