

# Training Fuzzy Logic Based Software Components for Reuse

Junda Chen and David C. Rine  
School of Information Technology and Engineering  
George Mason University, Fairfax, Virginia 22030  
chenj@wangfed.com drine@cs.gmu.edu

## Abstract

*A training framework of an effective method for off-line training of a class of control software components (for example, for first-order nonlinear feedback control systems) using combinations of three kinds of adaptation algorithms is presented. Each control software component is represented at the abstract level by means of a set of adaptive fuzzy logic (FL) rules and at the concrete level by means of fuzzy membership functions (MBFs). At the concrete representation level adaptation algorithms specified for use in adapting MBFs are: genetic algorithms, neural net algorithms, and Monte Carlo algorithms. We specify effective combinations of these three existing adaptation algorithms to train an erroneous FL rule-based software component in the tracker problem. In the framework, training consists of two phases: testing and adapting. In this paper, only the adapting phase is addressed. For each fault scenario adaptation algorithms and their combinations are used to modify the MBFs of the component. Effectiveness of the adapting phase is determined in terms of flexibility, adaptability, and stability. We perform experiments using a genetic algorithm. Simulation results are discussed.*

## 1. Introduction

A reusable software component with a training approach as an alternative to software component reuse without training is presented. Our research relates to the following perspectives: software component *reusability*, *trainability*, and *stability*.

A *reusable* software component is defined as a software component which can be used in other software systems being developed in the same domain or application environment. It is observed that most software systems are not as “soft” or flexible as we expect. Instead, they are “hard” and brittle. Hence,

development and maintenance are both expensive. For example, maintenance alone may take up to 65% of the total development cost (Lientz and Swanson [5]). An additional difficulty in building a software controller is that the controller is used to control a system under an evolving or unreachable environment. To date, a control system, for example, a smart vehicle, is an integration of many electronically control components. Development of applications includes the cost of finding an intelligent control model, testing and adapting individual software control components, and integrating them to an application environment. It is commonly believed that the above costs can be reduced by decreasing development and maintenance effort while increasing product quality. Software component reuse is in general considered as an effective approach for decreasing development and maintenance effort and increasing product quality. Traditionally, code component reuse can be approached by using three kinds of methods. They are transformational methods, template reuse methods, and building block composition methods. In these methods, a component domain model or domain repository needs to be maintained. In many cases, “it is likely that when software developers are able to locate a useful component, it will not integrate into their design without major modification.” (Sonnemann [10]). Hence, to perform software component reuse, it is necessary to either modify the selected domain-specific component (DSC) or modify the component repository structure. Studies show that application software perfective maintenance is the most expensive task within the software maintenance phase (Lientz and Swanson [5]). It can be imagined that manual modification is expensive, especially in an unreachable circumstance and in a continually evolving environment. Hence, in our research we have selected an automated approach DSC modification.

*Software training* is a way to achieve automatic modifications, originated and motivated by Rine [8]. This was expanded in [2]. A trainable (software) component (TC) is specific for a particular application domain.

Training is the process of changing internal structures with external behavior enhancement by using training examples. There are two training modes: supervised and unsupervised. In a supervised mode, the trainer provides training examples for every training cycle. In an unsupervised mode, there is no trainer involved in each training cycle. The trainer needs only to specify a fitness function. A related architectural framework can be found in [9].

A component training is said to be *stable* when its output change is proportional to the input change. In a software development or application environment, input change is the difference between the new and old requirements. The output is the portion of internal structure changes. Stability directly affects the maintenance effort of the software component. One reason for unstable component changes would be conflicts between the changing application environment and the requirement. Relaxing precision in the requirement specification and postponing the precise binding between the application environment and the requirement specification can enhance stability.

In this paper, we present a fuzzy logic based software training method combined with adaptation based algorithms. With this method, we hope that in-house perfective maintenance effort on controller software code components can be reduced.

## 2. Fuzzy logic and adaptation algorithms

A fuzzy logic controller (FLC) is a controller based on fuzzy set theory. A typical fuzzy logic (FL) control system is shown in (Figure 2.1), where  $e$  is the error feedback,  $\Delta e$  is the change of  $e$ ,  $d$  is a derivation which transfers  $e$  to  $\Delta e$ ,  $u$  is the input to the controlled process  $x$ ,  $\Delta u$  is the control output from the defuzzifier,  $s$  is an integration which transfers  $\Delta u$  to  $u$ ,  $y$  is the actual performance of the controlled process,  $y_d$  is the desired performance,  $r$  is user commands,  $v$  is system uncertainty, and  $w$  is measurement uncertainty. In a tracker control problem  $y_d$  is changeable. A FLC or fuzzy logic system has four kinds of modules, a FL component, fuzzifier(s), a fuzzy inference engine, and defuzzifier(s). A *FL component* consists of two parts, the rules (or fuzzy associative memory (FAM)) and the membership functions (MBFs) Figure 5.1). The rules encode human knowledge, whereas the MBFs interpret the rules into numeric values.  $E$ ,  $\Delta E$ , and  $\Delta U$  (denoted as Noun Names:  $e$ ,  $de$ , and  $du$  in Figure 5.1 respectively) are fuzzy variables of  $e$ ,  $\Delta e$ , and  $\Delta u$ . Their domain values include: Negative, Small Negative, Zero, Small Positive, and Positive. Each triangle is called a fuzzy generic

element (FGE). For isosceles triangular FGEs, let  $z_{ij}$  and  $l_{ij}$  be the peak position and the width of the  $j$ th FGE in the  $i$ th MBF respectively.

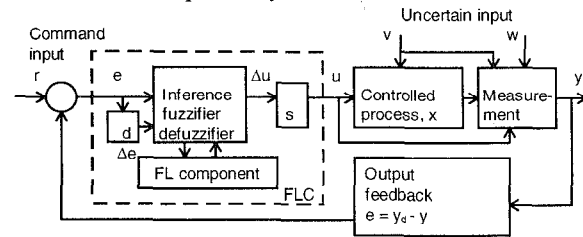


Figure 2.1 A fuzzy logic feedback control system.

*Genetic algorithms (GAs)* are used to maintain a best solution set. Each candidate solution is represented as a linear structure (chromosome)

$$Y = \{y_0, y_1, y_2, \dots, y_{N-1}\} \quad (2.1)$$

where  $y_i$  is a gene and  $N$  is the size of a chromosome. In fact, a solution can be represented in any structure. Linear structure is the simplest representation. Let  $pop_{sz}$  be the population size of chromosomes. A population of candidates,  $Y_0, \dots, Y_{pop_{sz}-1}$ , is manipulated by GA operations of mutation, crossover, and selection. Let  $G$  represent a combination of these operations. In each generation, a new set of candidate solutions  $Y_0', \dots, Y_{pop_{sz}-1}'$  is

$$Y_0', \dots, Y_{pop_{sz}-1}' = G(Y_0, \dots, Y_{pop_{sz}-1}) \quad (2.2)$$

A selection operation, which determines a new set of chromosomes, is based on the evaluation of a fitness function,  $f(Y) = \Sigma e^2$ , where  $e$  denotes the difference between the desired performance and the actual performance of  $Y$ .

*Neural networks and their training algorithms (NNs)* are used to fine tune, generalize, and store candidate solutions from a large set of numeric inputs by adjusting weights of network edges. A multi-layer feedforward neural network can be trained by an *error back-propagation algorithm (BP)*. Let  $X = \{x_0, x_1, \dots, x_{N_i-1}\}$  represent an input with  $N_i$  attributes (input nodes). Let  $Y_d = \{y_0, y_1, \dots, y_{N_o-1}\}$  represent a desired solution with  $N_o$  attributes (output nodes). Each training cycle includes two phases: a feedforward phase and an error back-propagation phase. In a feedforward phase, a candidate solution  $Y$  is generated. In an error back-propagation phase, error from the output layer is back propagated and edge weights are updated. After being trained, the network is expected to come up with a reasonable  $Y'$  efficiently for any new  $X'$ . It is also expected that if  $X'$  is close to a given  $X$  in the training examples, then  $Y'$  is also close to the  $Y_d$  of  $X$ .

*Monte Carlo algorithms (MCs)* have been used for solving complex function optimization problems. A MC is an extension of a neighborhood searching method (for

example, decent method). One random factor of the search strategy is

$$\delta = g(X_{next}) - g(X_{now}) > 0, \quad (2.3)$$

where  $X_{next}$  is a random move generated from current state  $X_{now}$ , and  $X_{next}$  is still considered to be the next move with a decreasing probability. For example, this probability can be

$$\exp(-\delta/t) > a \quad (2.4)$$

$$\text{New } t = \alpha(t), \quad (2.5)$$

where  $a \in (0, 1)$  is the random generated threshold, and  $t$  is the “temperature” which is cooling down with a procedure  $\alpha(.)$  as the search goes on. As the “temperature” is cooling down, the randomness reduces, and the search “homes in” to a solution  $x^*$ . Hopefully,  $g(x^*)$  is a better result than one found from a traditional decent method that may end up in a local optimum. One crucial factor in using MCs is the choice of the cool down procedure. In fact, a MC is a special case of a GA where the population reduces to 1 and the mutation rate increases.

### 3. Research problem

*Principles of reusable modules are: information hiding, abstraction, encapsulation, portability, and modularity. Information hiding* is a principle by which a module’s details that do not contribute to its essential functional and informational characteristics are hidden from the user. *Abstraction* is an essential characteristic of a module that distinguishes its end-user services from all other kinds of end-user module services and thus provides crisp-defined conceptual boundaries relative to the perspective of the end-user. *Encapsulation* is an essential characteristic of a module that groups state and behavior within a well-defined boundary. *Portability* is an essential characteristic of a module that ensures that the module end-user services are independent of the supporting concrete software and hardware. *Modularity* is an essential characteristic of a module that ensures that the module has high cohesion, limited complexity, high understandability, and limited coupling with other modules. An *abstract data type* (ADT) is a reusable module because it satisfies the above principles. An ADT has an abstract specification (Spec) layer interfacing with the ADT user and a concrete body layer containing current internal representation details. For example, specification of a cruise controller software module at the abstract ADT level (component) may contain high level rules, such as

*“if actual speed is less than desired speed and the vehicle is accelerating, then press the accelerator a little.”*

In the usual algorithmic concrete body level, specification is implemented in terms of a crisp code body. To reuse this component for different engine models and road conditions, people usually modify the algorithms of the body *by hand* with programming support tools.

We present a training framework using FL components and adaptation algorithms. The justification for this training framework is as follows. First, a FL software component (module) satisfies the same aspects of the principles of reusable modules. That is, it must be reusable [10]. Second, a FL component is represented with two parts: a FAM and a set of MBFs. We may make an analogy between a traditional reusable module and a FL module. A FAM is abstract and is like the specification of a traditional reusable module. For example, in our control domain an abstract form of fuzzy rules have terms of error, change of error, and change of control. MBFs contain the representation details and are like the body of a traditional reusable module. A human user can only see a FAM that hides implementation details from the user. A behavior boundary is defined by the FAM and MBFs. A FL component can be ported to different environments by tuning MBFs. Hence, a FL rule-based ADT follows the principles of reusable modules. In addition, FL has been proved very effective in handling imprecision in human reasoning. It is believed (Mamdani [7]) that using FL to develop and maintain a rule-based controller is easier and less expensive than using conventional approaches. It is especially useful when rigid and precise mathematical modeling is impossible or unnecessary. Thirdly, studies show that FLCs can be tuned by adaptation algorithms (Chang [1], Karr [4], Lin and Lee [6]). Hence, a FL component can be an adaptive ADT and is defined as a *trainable software component*. The FAM and MBFs replace the usual specification layer (abstract) code and body layer algorithm’s (concrete) code, respectively. With this new technique, a fuzzy logic cruise controller software module (component) is specified by a set of abstract rules, such as

*“if Error is Positive and dError is Negative, then dControl is Small Positive.”*

This rule is most likely unchanged for component reuse. The body with MBFs is modified by adaptation algorithms, in which  $z_{ij}$  and  $l_{ij}$  change accordingly. Hence, the component is reused by automatic tuning without much affect on the component’s user. Hence, a FL-based representation of a software component supports both ease of understanding *by a trainer* and ease of *automatic* modification.

There does not exist an effective method for off-line training of a class of control software components (for

example, in first-order nonlinear feedback control systems for the tracker problem) using combinations of three kinds of adaptation algorithms. These adaptation algorithms are genetic algorithms, neural network training algorithms, and Monte Carlo algorithms. Given a domain of control and a trainable software code component of the domain, we have the following research questions. What are the combinations of adaptation algorithms to make the training effective and efficient? What are the trainability metrics? How can this training method decrease software development and maintenance effort and increase the software component quality? Using GA or NN separately to tune FLC is nothing new. However, applying FL and various combinations of adaptation algorithms to software component engineering in terms of automatic program (code component) modification is new. It is useful to have a component reuse framework for these control software components.

#### 4. Research method

The research is carried out in the following steps: start with domain engineering of reusable component at the abstract level, design the trainable software code components (TC), define a training method, define a trainability metric taxonomy, design and implement a training environment, and perform experiments.

Let us assume we have domain engineered abstract-level reusable software code components [10]. A FL based TC has multiple inputs ( $e, \Delta e, \Delta^2 e, \dots, \Delta^{n-1} e$ ) and one output ( $\Delta u$ ). Each MBF<sub>*i*</sub> in a TC has the same numbers of FGEs. This input/output representation is abstract, which means it can apply to any other feedback control domain for a tracker problem without significantly changing the fuzzy rules of the component. A training method is defined in two phases: testing and adapting. The fault scenario driver is the major component in the testing phase, which adaptively searches for effective fault scenarios (*fs*'s). The adapter in the adapting phase modifies the component. Adaptation algorithms apply to both phases. Hence, a *solution* in the testing phase is the most effective fault scenario, while in the adapting phase a *solution* is the component with minimum error function value. A GA-BP testing method for finding fault scenarios is described in (Chen and Rine [3]). The goal of the testing phase is to generate *fs*'s and locate faulty fuzzy elements (FFE)s. An erroneous fuzzy rule causes a higher *e* after its execution than the previous *e* as input to the FLC. FFEs are FGEs in all erroneous fuzzy rules. In this paper, only the adapting phase is addressed. Assume one is given a fault scenario,

a controlled process simulation, and an erroneous TC. Adapting the TC includes the following steps.

1. Specify performance evaluation criteria (fitness function).
2. Specify *modifiable fuzzy elements (MFEs)* of the component induced from the test phase.
3. Generate new values for the *MFEs* by using adaptation algorithms (for example, GA).
4. Evaluate performance of the component by executing the fault scenario.

Repeat step 3 and 4 until performance is satisfied.

The effectiveness (*trainability*) of the three adaptation algorithms and their combinations in the adapting phase can be addressed by three metric categories: *flexibility*, *adaptability*, and *stability*. A fuzzy logic representation is *flexible*, because changes in MBFs do not affect the semantic of the fuzzy rules. *Adaptability* is defined as the ability to improve a component's performance during training. That is "How easily can the reconfigured component reach an acceptable performance (in terms of trials)?" *Stability* is defined in Section 1. Obviously, changing MBFs are less expensive than changing fuzzy rules.

The training environment consists of the following components: a min-max compositional fuzzy inference engine, a fault (test) scenario driver, a component adapter, a GUI, a TC, and a controlled process simulation. We use FLDE (Fuzzy Logic Development Environment, from Syndesis Ltd., Greece) as our GUI in which a C code generator is also available. All other components are written in C++. The integrated program executes on a Pentium PC.

#### 5. Experiments and discussions

The controlled process is simulated according to a vehicle linearized mathematical model,  $f(x, t) = dx/dt = Ax + Bu + Rv$ ,  $y = x$ , where  $A = -2ky_d/m$ ,  $B = 1/m$ ,  $R = \mu g \sin v_0 - g \cos v_0$ ,  $v$  is grade of slope,  $y_d$  is desired speed,  $\mu$  is friction between the road and the tire,  $m$  is the mass of the vehicle,  $v_0$  is the starting  $v$ ,  $g (= 9.8)$  is gravity,  $k = 0.01$ , and  $h = 0.01$ . The actual speed  $y$  can be obtained by solving this differential equation. In the experiment, we show how the framework works. Let us use FLDE to define a TC with two inputs, one output, 3 MBFs, and 5 FGEs per MBF (simply 2INPUT-5FGE) (Figure 5.1). We run the test phase where a *fs* is generated. A fault scenario *fs* is a sequence with two environment states,  $st_0$  and  $st_1$ . An environment state is an instance of an environmental condition. A state is defined as  $(v, y_d, \mu, m)$ . Let us have a two state *fs*,

{0.523705 1.986804 0.043548 5004.000000;

0.567537 2.998534 0.020166 3081.935484}.

The erroneous rule set found in the testing phase is {E=SN&dE=PO->dU=SN, ZE&PO->SP, PO&PO->PO}. Thus,

FFE<sub>s</sub> = {SN, ZE, PO (FFE for MBF0); PO (for MBF1); SN, SP, PO (for MBF2)}.

Note that a FFE may not be a MFE, and vice versa. The GA parameters are as follows. Crossover rate (*cr*) is 0.6. Mutation rate (*mr*) is 0.001. Population size (*popsz*) is 10. Maximum number of trials (*Mnt*) is 500. Structure (bit string) length (*ls*) is 160 bits (20 bits per variable or gene in the MFEs). We then specify all FGEs modifiable and run the adapting phase. A set of new MBFs are found by GA as

MBF0 = (-43.17, 12.67; -21.78, 14.89; 9.48, 21.04; 11.75, 23.01; 47.68, 12.43),

MBF1 = (-47.57, 25.76; -20.93, 11.75; -4.92, 27.99; 18.16, 17.73; 42.38, 29.47), and

MBF2 = (-4839.05, 2595.67; -2285.58, 1789.00; 334.08, 1249.24; 2848.02, 2363.13; 4295.39, 2440.62), where each FGE is represented by a pair of (*z*, *l*).

The performance of adaptation is shown in Figure 5.2 with ending SOE2=1104. Finally, the FL component is transformed to C code by FLDE. A code fragment is shown as follows.

```

unsigned char e_SP(unsigned char Crisp_Value)
{
    if (Crisp_Value <= 128)
        return 0;
    if (Crisp_Value > 128 && Crisp_Value <= 227)
        return ((unsigned int)255 * (unsigned
int)Crisp_Value - 32640) / 99;
}

```

```

if (Crisp_Value > 227)
    return (65025 - (unsigned int)255 *
(unsigned int)Crisp_Value) / 28;
return 0;
}
.....
m0 = e_SP(e);
m1 = de_NE(de);
duResults[3].m = AND0(m0, m1);
duResults[3].AdjIndex = 3;
if (maxm[3] < duResults[3].m)
    maxm[3] = duResults[3].m;

```

Experiments show that the adaptation process is sensitive to above parameters. In addition, the critical factor is to select a set of MFEs. We found that the best MFEs in this case is

MFE\* = {SN, ZE, SP, ZE, SP, PO; ZE, PO}

with ending SOE2=777 (Figure 5.3). Interestingly, either tuning a FGE between two FFEs or tuning a FGE between ZE and a FFE (underlined in MFE\*) can achieve better results than tuning FFEs alone or tuning all FGEs. (How can we induce MFE\* from FFEs?) To adapt the component by tuning MFE\* with various GA parameters, we observe the following. When *cr* = {0.4, 0.8}, the adaptation performance traps in local optima with ending SOE2 = {1085, 934}; SOE2 = {947, 1062} when *ls* = {160, 640}; and SOE2 = {900, 1100} when *popsz* = {5, 20} (Figure 5.4). Hence, we obtain the best adaptation performance by tuning MFE\* with GA parameters: *cr*(0.6), *mr*(0.001), *ls*(320, 20bits/gene), and

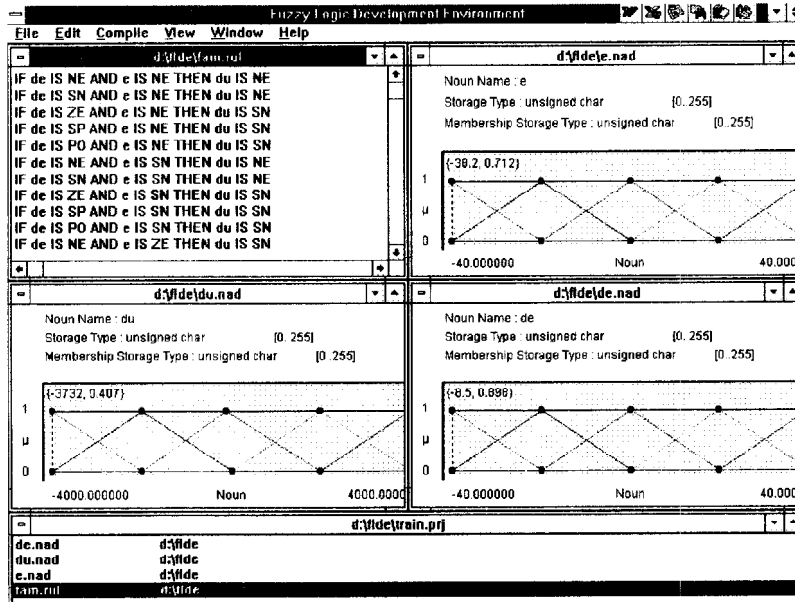


Figure 5.1 Defining a TC component using FLDE, where e.nad for MBF0, de.nad for MBF1, and du.nad for MBF2.

*popsz(10)*. Experiments of the training process show that, to some extent, this training framework enhances quality and reduces effort in controller software component development for reuse.

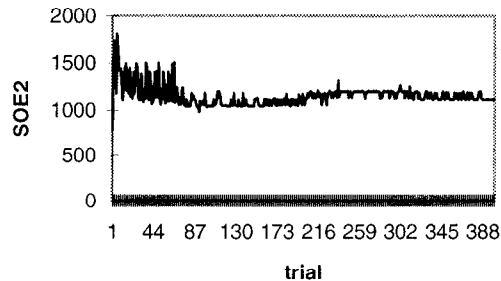


Figure 5.2 Adapting by modifying all FGEs.

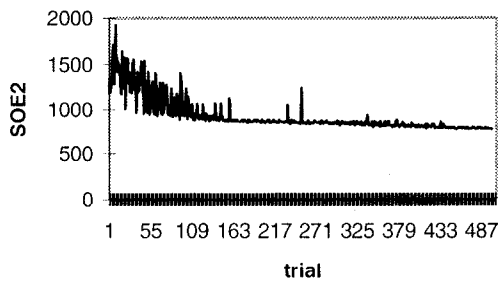


Figure 5.3 Adapting by modifying MFE\*

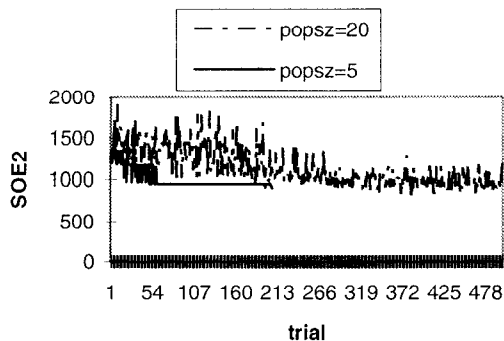


Figure 5.4. Population size affects on adaptation processes.

## 6. Conclusion and future works

A software component training framework is described. In the framework, we model a software

component by using fuzzy logic, in which adaptation is achieved by tuning MBFs. Trainability metric categories of flexibility, adaptability, and stability are discussed. Future work includes using NN/BP and MC for the adapting phase, using BP to generate fuzzy rules, training under different application environment, and training with another controller simulation.

## References

1. Chang, C.H.. (1993). Designing and tuning of a fuzzy logic controller via genetic input/output mapping factors. Ph.D. dissertation, University of Oklahoma.
2. Chen, J., and Rine, D., (1995). A training framework to develop reusable software components by combining adaptation algorithms, *Proceedings of Artificial Neural Networks In Engineering (ANNIE'95) Conference*, (C.H. Dagli, et al. Eds.), Nov. 12-15, Sponsored by the Office of Naval Research, St. Louis, MO.
3. Chen, J., and Rine, D., (1996). Testing trainable software components by combining genetic algorithms and backpropagation algorithms, *Proceedings of Artificial Neural Networks In Engineering (ANNIE'96) Conference*, (C.H. Dagli, et al. Eds.), Nov. 10-13, Sponsored by the Office of Naval Research, St. Louis, MO.
4. Karr, C.L., (1993). Fuzzy control of pH using genetic algorithms, *IEEE Trans. on Fuzzy Systems*, vol. 1, no.1, 46-53.
5. Lientz, B.P., and Swanson, E.B., (1981). Problems in application software maintenance, *Commun. ACM*, vol. 24, no 11, 763-769.
6. Lin, C.-T., and Lee, C.S.G., (1994). Supervised and unsupervised learning with fuzzy similarity for neural network-based fuzzy logic control systems, in *Fuzzy Sets, Neural Networks, and Soft Computing* (R.R. Yager, and L.A. Zadeh, Eds.), Van Nostrand Reinhold, 85-125.
7. Mamdani, E.H., (1994). Twenty years of fuzzy control: experiences gained and lessons learnt, in *Fuzzy Logic Technology and Applications* (R. J. Marks II, Ed.), IEEE.
8. Rine, D., (1993). Software perfective maintenance: including retrainable software in software reuse, *Information Sciences: International Journal*, vol. 75, Dec.
9. Rine, D., Ahmed, M., and Chen, J., (1996). A reusable software adaptive fuzzy controller architecture, *Proceedings of the ACM International Conference on Software Applications*, Feb. 15-17.
10. Sonnemann, R. M., (1995). Exploratory study of software reuse success factors, Ph.D. dissertation, George Mason University.