Fuzzy Subtractive Clustering Based Indexing Approach for Software Components Classification

Sathit Nakkrasae[1,2,3]
[1]*Department of Computer Technology, Faculty of Science, Ramkhamhaeng University, Bangkok, 10240, Thailand.*
Khunsathit@hotmail.com

Peraphon Sophatsathit[2]
[2]*Advanced Virtual and Intelligent Computing (AVIC) Center, Faculty of Science, Chulalongkorn University, Bangkok, 10330, Thailand.*
Peraphon.S@Chula.ac.th

William R. Edwards, Jr.[3]
[3]*Center for Advanced Computer Studies (CACS), The University of Louisiana at Lafayette, Lafayette, LA 70504, U.S.A.*
wre@cacs.louisiana.edu

Abstract

Software Engineering is not only a technical discipline of its own, but also a problem domain where technologies coming from other disciplines are relevant and can play important roles. One important example is knowledge engineering [1], a term that is used in a board sense to encompass artificial intelligence, computational intelligence, knowledge bases, data mining, and machine learning. Many typical software development issues can benefit from these disciplines. For this reason, this paper will employ computational intelligence approach to classify software component repository into similarity component cluster groups with the help of Fuzzy Subtractive Clustering algorithm. The center of each cluster will be used to construct a coarse grain classification indexing structure. Subsequent retrieval requirements of software component are compared with all the indexed cluster centers. Any software components belonging to the cluster partition whose center is closest to the required software component will be retrieved for subsequent participation in component selection at fine grain level. This approach not only is suitable for multidimensional data, but also automatically decides the correct model classification.

Keyword: software component classification, knowledge engineering, neural networks, and Fuzzy Subtractive Clustering.

1. Introduction

Reuse is a popular design methodology common to engineering discipline. It has two primary aspects: (a) cost reduction resulting from not design a new solution; and (b) increased confidence in the solution because of its successful reusing. For reuse to be an effective problem solving methodology, the designer must be able to reuse appropriate solutions, adapt a solution to fit the new problem, and evaluate the resulting solution. In software engineering, reuse is popularly applied in design domain. Owing to creativity and complexity of design paradigms, approaches, and the process itself, design reuse must, in many cases, be tailored to suit specific requirements. Moreover, automated software reuse support has been slow to emerge due to the difficulty in providing a useful design representation for software components. This design representation must be able to efficiently support component storage, retrieval, adaptation, and verification.

This paper presents software component matrix representation based on a well-defined software component specification [2] and an alternative software component classification and retrieval approach, utilizing computational intelligence on neural networks [3].

The remaining of this paper is organized as follows. Section 2 discusses related papers on formal specification, and classification of software components. Other related topics are also incorporated. Matrix representation of software component based on structural, functional, and behavioral properties is presented in Section 3. Models and methodologies of software component classification in both coarse and fine grain are described in Section 4. Section 5 discusses the experiment and the results of component classification. Our final thought is discussed in Section 6.

2. Related Work

A popular method for describing repositories of reusable software components is a faceted classification scheme [4]. Using this methodology, components are classified by a set of attribute-value pairs, or features. A domain expert, who is required to analyze the repository of software components and classify them according to predefined terms, performs the classification. The knowledge of domain expert is implicit in the classification. To provide a basis for similarity calculations, the terms that represent the set of possible values for a feature are often related by a conceptual distance graph [4]. The informality and imprecision of these classification schemes complicates the automation of the overall reuse process. Automation of the classification process requires reverse engineering from source code. The imprecision of the classification scheme does not support formal component verification since reasoning about identically classified components requires source code analysis.

The use of formal specifications to augment software reuse has been proposed to solve problems [2, 5, 6, 7, 8, 9, 10, 11]. There are many benefits to applying formal methods to software reuse. First, formal specifications provide an explicit representation of structure, function, and behavior of a software component free from many implementation details. This is valuable because structure, function, and behavior are the primary point of interest when determining reusability. Next, the expressiveness of formal specification languages allows precision beyond that of faceted classification. Equivalent specifications perform equivalent properties of software component (structural, functional, and behavioral). Finally, formal specifications and their associated formal system provide a basis for automated reasoning. A formal specification defines the structure, function, and behavior within a domain model, which is a collection of axioms that define the data types and operation used in the system. Formal reasoning based on the domain model can be used logically to verify the reusability of a software component.

This paper proposes a systematic approach for classifying software components in the form of machine learning through computational intelligence such as neural networks, fuzzy logic, and artificial intelligence. Some suggest Self-Organizing Map (SOM) [1, 12] for software component classification. However, there are limitations on this method:

- Its determination is not based on optimizing any model of process or data;
- Prototype parameters may be severely affected by noise from data points and outliners. This is due to the fact that learning rates in SOM are computed as a function of the number of input presentations and node positions in the grid, while they are independent of the actual distance separating the input pattern from the cluster template;
- The size of the output lattice, the step size, and the size of the resonance neighborhood must vary empirically from one data set to another to achieve useful results; and
- It should not be employed in topology-preserving mapping when the dimension of the input space is larger than three [13].

Other popular classifying techniques are also taken into account, such as Fuzzy C-Means clustering technique, which is a simple and straightforward approach but requires two predefined clusters where every data point membership depends on membership grade. It is clear from existing approaches that clustering technique is the fundamental building block of data classification. As such, we proposed Fuzzy Subtractive Clustering (FSC) technique [14] which is a fast one-pass algorithm for estimating the number of clusters and cluster centers in a set of data [15] to pre-process the software components. Once the software component groups are formed, classification process can proceed.

3. Software Component Representation

The proposed approach employs a formal specification [2] describing three properties of software component, namely, structure, functional, and behavioral properties, free from most implementation details. These specifications are denoted in matrix form to support classification in the component repository. Subsequent retrieval of the desired component will utilize the same matrix to find the appropriate matching. As such, we will present a formulation of the classification matrix below.

Define software component X to be

$$X = (S, F, B)$$

where S denotes structural properties, F denotes functional properties, and B denotes behavioral properties. Each property is a list of the form

$$\begin{aligned} S &= \{ S_1 , S_2 , S_3 ,\dots, S_m \}, \\ F &= \{ F_1 , F_2 , F_3 ,\dots, F_n \}, \text{ and} \\ B &= \{ B_1 ,B_2 , B_3 ,\dots, B_p \}, \text{ respectively.} \end{aligned}$$

Each member of the list S, F, and B is also a list of the form

$$\begin{aligned} S_i &= \{ S_{i,1} , S_{i,2} , S_{i,3} , \dots , S_{i,ui} \}, 1 \le i \le m \text{ and } S_{i,j} \in D(S_i) \\ F_i &= \{ F_{i,1} , F_{i,2} , F_{i,3} , \dots , F_{i,vi} \}, 1 \le i \le n \text{ and } F_{i,j} \in D(F_i) \\ B_i &= \{ B_{i,1} , B_{i,2} , B_{i,3} , \dots , B_{i,wi} \}, 1 \le i \le p \text{ and } B_{i,j} \in D(B_i) \end{aligned}$$

and u_i, v_i, and w_i denote the number of members within S_i , F_i, and B_i, respectively. Each member is ordered from left to right, followed by the software component specification [2]. $D(S_i)$, $D(F_i)$, and $D(B_i)$ define separate equivalent classes (*EC*). For example, a system designer may wish to define a family of data objects to be stack-like, all belong to equivalent class of *LIFO*. This formulation entails a set of equivalent classes to be predefined within a component repository system by designer or developer.

Two preamble assumptions of our component repository stipulate that the number of elements in the set of equivalent classes for a given software component be finite, and that the number of each equivalent class in each property of the components be known. Denote the number of each structural, functional, and behavioral equivalent class properties by T_{Si}, T_{Fj}, and T_{Bk}, where $1 \le i \le m$, $1 \le j \le n$, and $1 \le k \le p$, respectively, we define property matrix representation as follows:

$$\begin{aligned} &Col\_s = \text{Max}(T_{Si} ,1 \le i \le m), && Row\_s = m \\ &Col\_f = T_{F1}, && Row\_f = 1 + \Sigma^{n}_{i=2} T_{Fi} \\ &Col\_b = T_{B1}, && Row\_b = 1 + \Sigma^{p}_{i=2} T_{Bi} \end{aligned}$$

```
First: S Row_s×Col_s
    For i = 1 to m
        For j = 1 to $T_{Si}$
            S (i,j) = l ($S_i$ has l terms in equivalent class j)
Second: F Row_f×Col_f
    For i = 1 to Fnum
    Begin
        j = Eq_class_number(function(i))
        F(1,j) = 1
        For k = 2 to Row_f
            F(k,j) = m ($F_j$ has m terms in equivalent class k)
     End for i
Third: B Row_b×Col_b
    For i = 1 to Bnum
    Begin
        j = Eq_class_number(Behavior(i))
        B(1,j) = 1
        For k = 2 to Row_b
            B(k,j) = p ($B_j$ has p terms in equivalent class k)
    End for I
```

Figure 1. Matrix calculation algorithm

Based on this representation, a software component matrix *X* can be written as follows:

$$C = (S_{Row\_s \times Col\_s}\ ,\ F_{Row\_f \times Col\_f}\ ,\ B_{Row\_b \times Col\_b})$$

The algorithm for component matrix formulation proceeds as Figure 1, where *Fnum* denotes the number of functions in a component and *Bnum* denotes the number of behavior in the component. The final matrix becomes

$$X = (S, F, B)$$

This matrix will be transformed for use by subsequent proposed neural network computations.

4. Software Component Classification

Our approach for component classification is based on how software component is reused through the reuse model in order to establish a classification framework over the applicable component domain. We employed formal notations presented in [2] to represent component class, along with an example to demonstrate the applicability of the proposed framework. Various components are then grouped by coarse grain criteria (structure, function, and behavior). We measured classification correctness by means of recall and precision techniques. If the result is satisfactory, we proceed to fine grain classification to ensure proper reuse indicator for the designated components being retrieved.

4.1 Software Reuse Model

The software reuse model encompasses a repository which stores formal specifications of software components and retrieval mechanisms to facilitate component check-in/check-out during the development process. The underlying principle of the proposed classification scheme relies on component similarity comparison that is derived from a user-defined classification function. This offers a quantitative technique to enumerate the component suitability in coarse grain level. Assessment begins by representing software components in matrix form.

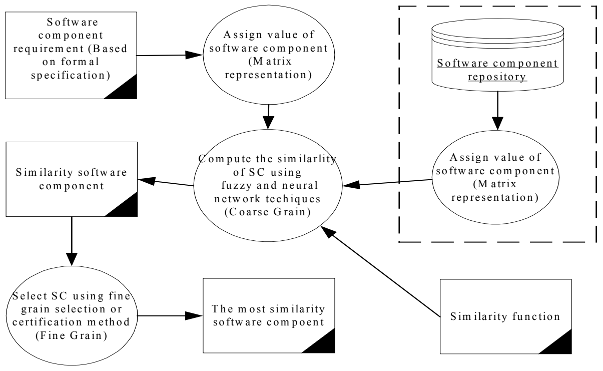
Software component requirement (Based on formal specification)
Assign value of software component (Matrix representation)
Software component repository
Similarity software component
Compute the similarlity of SC using fuzzy and neural network techiques (Coarse Grain)
Assign value of software component (Matrix representation)
Select SC using fine grain selection or certification method (Fine Grain)
The most similarity software compoent
Similarity function

Figure 2. Software reuse model

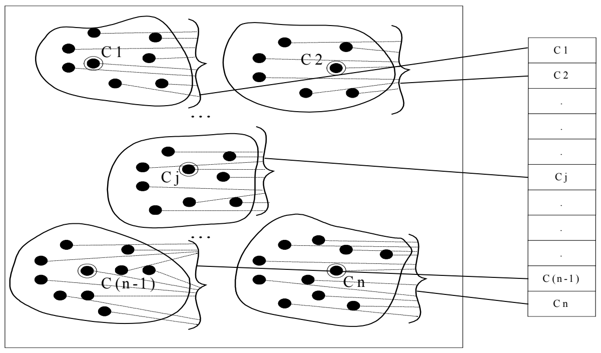

Figure 3. The n cluster partitions generated by FSC algorithm. The black dots and double line black dots denote software components and cluster centers, respectively

This permits quantitative evaluation of the requirements specification of the designated component and the components stored in the repository, based on the requirement specification of the cluster component. Subsequent classification process will sort out the closest matched component for reuse purpose. Detail on how classification and retrieval are carried out is described in the next section. Evaluation is performed using FSC algorithm to arrive at a similarity value. This process is depicted in Figure 2.

4.2 Coarse Grain Level Software Component Classification Techniques

The three properties used to classify software components are component structure, function, and behavior. The component structure is made up of a component name, a subcomponent name, a class name, a signature, and an interaction name. The component function consists of a function name, input parameters, local variables, output parameters, and pre/post-expressions. The component behavior is composed of a behavior name, a state name, and an action name. These properties are represented in matrix form described in Section 3.

The classification process starts by dividing software components into groups using FSC algorithm. These clusters are used to construct the index structure as shown in Figure 3. Searching for the closest match between the specified requirement and those in the index yields optimal software component retrieval. When a match is found, all components belonging to that cluster are retrieved. In general, more than one match may result. A fine grain certification step is required to ensure the best candidate being retrieved. Details on how certification proceeds will be postponed till Section 4.3.

Two measures of software component retrieval performance used in this paper are recall and precision [16]. Recall is the ratio of the number of relevant items retrieved to the total number of relevant items in the repository. High recall indicates that relatively few relevant software components were overlooked. Precision is the ratio of the number relevant items retrieved to the total number of items retrieved. High precision means that relatively few irrelevant software components were retrieved. In general, there is tradeoff between precision and retrieval. The goal is to find a practical balance between the two. The relevance condition is fundamental to the evaluation of a retrieval system.

It was also informative to observe the number of software components retrieved by the system. This number helped estimate the load that would be placed on the designer to interpret the results of a query in an interactive system, or similarly, the search space that would be faced by an adaptation system when considering software component compositions.

Given a set of a priori clusters $C = \{c\}_1^n$ and the calculated FSC clusters, $C' = \{c'\}_1^m$, the performance measures of FSC is defined as follows [17]:

Recall = Number of target software components retrieved / Number of target software components

$$= \sum_{c_i \in C \wedge c_j' \in C'} \frac{c_i \cap c_j'}{\# c_i} \qquad Eq(1)$$

Precision = Number of target software components retrieved / Number of software components retrieved

$$= \sum_{c_i \in C \wedge c_j' \in C'} \frac{c_i \cap c_j'}{\# c_j'} \qquad Eq(2)$$

where $\#c_i$ denoted the number of elements on cluster c_i and $0 \leq recall, \; precision \leq 1$. Based on the above definitions, recall expresses the ratio of the target repository objects being actually retrieved out of all the expected target repository objects, whereas precision indicates the ratio of target repository objects to the retrieved set. For example, there are 10 repository objects and 4 of them are pre-specified as target repository objects. Given a query retrieving 5 objects and 3 out of those five objects are target objects. In this case, recall is 0.75 and precision is 0.6. The higher the recall and precision get, the more accurate the method for retrieval becomes. We can calculate the accuracy for each FSC clusters based on the information pertaining to their natural clusters. The response time of the system was measured to determine the practicality of the method. For each measured quality, the minimum and median were calculated from every scenario in the experiment, which will be discussed in Section 5.

4.3 Fine Grain Level Software Component Selection Technique

In this level, we will try to find the most suitable software component for reuse. The degree of significance defined by user will be used as the selection criteria. The following notations will be given:

- $\O_s$, $\O_f$, and $\O_h$ are the degree of significance of structural, functional, and behavioral properties, respectively, satisfying $0 \leq \O_s , \O_f , \O_h \leq 1$ and $\O_s + \O_f + \O_h = 1$. The degree of significance depends on system environment under which developers can define in accordance with the underlying system;
- N_r is the number of retrieved software components from the cluster whose center is closest to the required software component;
- X_i is the i^{th} retrieved software component in the component matrix described in Section 3, i.e., $X_i = (S, F, B)$ where $1 \leq i \leq N_r$;
- X_r is the component requirements; and
- SC is the most suitable software component which can be determined as follows:

$$SC = X_{reuse}$$

where the value of reuse can be computed from

$$reuse = \arg \min_{1 \leq i \leq N_r} \Big(\sum_{p=S,F,B} \phi_p \; \| \; Xp_r - Xp_i \; \| \Big) \qquad Eq(3)$$

5. Experiment

5.1 Data Collection

One hundred software component specification data were generated by uniform distribution generator. The data were arranged in matrix form suitable for the proposed algorithm described in Section 3. Components were classified according to their structural, functional, and behavioral properties which, in turn, were grouped into appropriate equivalent classes. In so doing, each component data vector encompassed 1320 dimensions.

The data set was divided into two groups, namely, 50 training set and 50 test set. Each data vector was normalized within [0,1] according to

$$v_{new} = \frac{v_{old} - v_{\min}}{v_{\max} - v_{\min}} \qquad Eq(4)$$

where v_{new} is the new value of the designated variable for that data point, v_{old} is the old value of the data point, v_{min} is the minimum value of the variable from all data points, and v_{max} is the maximum value of the variable from all data points.

5.2 Cluster Center Detection

We selected FSC approach to determine cluster centers using parameter $r_a = 14$. This value is the maximum distance between any two points within the same cluster, yet less than the distance between any two points from different clusters where each point belongs. The multiplier *Sqsh* = 1.25 is the default squash factor value of MATLAB 5.3.

The criteria for cluster center consideration are based on acceptance and rejection ratios. Acceptance ratio can be determined by fractions of the potential first cluster center, above which another data point will be accepted. Rejection ratio is the condition to reject a data point to be a cluster center, which is obtained from fractions of the potential first cluster center, below which a data point will be rejected as a cluster center. We chose 0.5 as the acceptance ratio (which is the default value from MATLAB version 5.3) for the first cluster center. We chose the rejection ratio (η) between 0.15-0.5 to derive other cluster centers. The resulting rejection ratios from various cluster centers were used to compare and evaluate the component classification. The procedure for grouping 50 data point clusters $\{ X_1, X_2, X_3, \ldots, X_{n=50} \}$ in the training set is described below.

1. Compute the initial potential value for each data point (x_i)

$$P_i = \sum_{j=1}^{n} e^{-\alpha \| x_i - x_j \|^2} \qquad Eq\ (5)$$

where $\alpha = 4/r_a^2$

$\| . \|$ is the Euclidean distance

r_a is a positive constant representing a normalized neighborhood data radius

Any point falls outside this encircling region will have little influence to the potential point. The point with the highest potential value is selected as the first cluster center. This tentatively define the first cluster center.

2. A point is qualified as the first center if its potential value ($P^{(1)}$) is equal to the maximum of initial potential value ($P^{(1)*}$)

$$P^{(1)*} = \max_i (P^{(1)}(x_i)) \qquad Eq\,(6)$$

3. Define a threshold δ as the decision to continue or stop the cluster center search. This process will continue if the current maximum potential remains greater than δ.

δ = *(reject ratio)* $\times$ *(potential value of the first cluster center)*

where the rejection ratio (η) used in this work is 0.15-0.5, and $P^{(1)*}$ is the potential value of the first cluster center.

4. Remove the previous cluster center from further consideration.

5. Revise the potential value of the remaining points according to the equation

$$P_i = P_i - P_k^* e^{-\beta \| x_i - x_k^* \|^2} \qquad Eq\,(7)$$

where x_k^* is the point of the k^{th} cluster center, P_k^* is its potential value, and $\beta = 4/1.25$ ($sqsh * r_a$).

6. For the point having the maximum potential value, calculate the acceptance ratio. If this value is greater than the predefined constant (0.5), the point is accepted to be the next cluster center. Otherwise, compute the rejection ratio. If the rejection ratio is greater than the predefined threshold (η = 0.15-0.5), this point is accepted.

This procedure is repeated to generate the cluster centers until the maximum potential value in the current iteration is equal to or less than the threshold δ. After applying subtractive clustering, we get different cluster center numbers from 50 training patterns depending on different rejection ratios. We used these different cluster center numbers to compare and evaluate software classification.

5.3 Evaluation

From the 100 vector data participated in the experiment, we regulated the rejection ratio in the range of 0.15-0.5 to avoid high rejection rate, whereby yielding too many unclassified or misclassification of the above data.

We assessed the accuracy of FSC algorithm by measuring recall and precision performance. The derived centers were anticipated to correctly classify software components in repository into each group of its predefined equivalent classes. From 50 training data set with predefined 10 equivalent classes, we conducted 5 trials using the remaining 50 test data with different rejection ratio (η) groups (0.15-0.20, 0.25, 0.30, and 0.35-0.50) and used the derived cluster centers from each trial to calculate its recall and precision performance. The results can be interpreted as follows. Based on 0.15-0.20, 0.25, 0.30, and 0.35-0.50 rejection ratio (η) groups, the values of cluster centers so derived are 18, 15, 11, and 10, respectively.

Table 1 shows the comparative results of all 4 rejection groups obtaining from FSC classification of software component being quite satisfactory. Note that recall performance suffers a slight drop due to the decreasing of rejection ratio (η). There was, however, no fault classification in each equivalent class since the centers were closely located to their corresponding component groups. As such, retrieval was accomplished with relatively few attempts. The high number of centers selected from the 10 equivalent classes having 0.15-0.20, 0.25, and 0.30 rejection ratio (η), implied that there were more than one center in each equivalent class, whereby yielding 98-100% accuracy.

5.4 Example of Fine Grain Component Selection

We employed cluster centers obtained from the experiment to create an index structure for use as an example to select the most suitable software component. Suppose *X* denotes a software component requirement matrix, which matches cluster center *SC*19, this cluster encompasses *SC*16, *SC*17, *SC*18, *SC*19, and *SC*20. Table 2 shows the results of most suitable software component selection from Equation 3 having different degrees of significance on structural, functional, and behavioral properties.

Table 1. Recall and precision performance comparison

Rejection Ratio	Number of Center Selected	Recall	Precision
0.15-0.20	18	0.56	1.00
0.25	15	0.73	1.00
0.30	11	0.89	0.98
0.35-0.50	10	0.98	0.98

Table 2. Software Component Selection with Different Degree of Significance

Degree of Significance			The most Suitable Software Component for Reuse
Structural	Functional	Behavioral	
0.8	0.1	0.1	SC18
0.1	0.8	0.1	SC19
0.1	0.1	0.8	SC18
0.3	0.3	0.3	SC18

6. Conclusion

We have proposed two computational intelligent approaches to classify software components for effective archival and retrieval purposes, namely, fuzzy subtractive clustering algorithm and neural network technique. Component specifications are represented in matrix form to quantitatively organize these software artifacts for subsequent applications. Components were indexed based on the cluster centers so obtained. As such, subsequent reference and retrieval could be carried out efficiently through this indexing mechanism. We also conducted an experiment to assess the validity of the proposed approach, which turned out to be quite satisfactory.

We envision in our future work concerning software certification process to benefit from this rigorous formulation that will eventually be incorporated as part of the machine learning research endeavor. As a consequence, pervasive use of software components in the same manner as their hardware counterparts, as well as the ultimate COTS application, can be realized.

7. References

[1] W. Pedrycz, "Computational Intelligence as an Emerging Paradigm of Software Engineering", in *Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering*, 2002, pp.7-14.

[2] S. Nakkrasae and P. Sophatsathit, "Formal Approach for Specification and Classification of Software Components", in *Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering*, 2002, pp.773-780.

[3] S. Haykin, *Neural Network*, Prentice Hall, pp.256-312, 1999.

[4] Eduardo Ostertag, James Hendler, Ruben Prieto Diaz, and Christine Braun, "Computing similarity in a reuse library system: An AI Base Approach", *ACM Transactions on Software Engineering and Methodology*, pp. 205-228, 1992.

[5] A. M. Zaremski and J. M. Wing, "Specification matching software components", in *the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[6] D. E. Perry and S. S. Popovitch, "In quire: Predicate-Based Use and Reuse", in *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, 1993, pp. 144-151.

[7] S. A. Ehikioya, "A formal model for the reuse of software specifications", in *IEEE Canadian Conference on Electrical and Computer Engineering*, Volume: 1, 1999, pp. 283-288.

[8] J. J. Jeng and B. H. C. Cheng, "Using formal methods to construct a software library", in *Proceedings of 4th European software Engineering Conference, Lecture Notes in Computer Science*, 1993, pp.397-417.

[9] J. J. Jeng and B. H. C. Cheng, "A formal approach to using more general components", in *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, 1994, pp. 90-97.

[10] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "LaSSIE: A knowledge-base software assistant", *Communications of the ACM*, pp. 34-49, 1991.

[11] R. S. Pressman, *Software Engineering, A Practitioner's Approach*, 4th Editon, New York:McGraw-Hill, 1997.

[12] S. M. Charters, C. Knight, N. Thomas and M. Munro, "Visualization for informed decision making; Form code to components", in *Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering: SEKE'02*, 2002, pp.765-772.

[13] A. Baraldi and P. Blonda, "A survey of fuzzy clustering algorithms for pattern recognition Part 2", *IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics,* Vol. 29 No.6, 1999.

[14] S. Chiu, "Method and Software for Extracting Fuzzy Classification Rules by Subtractive Clustering", in *Fuzzy Information Proceeding Society, Biennial Conference of the North American*, 1996, pp. 461-465.

[15] P. Eklund, L. Kallin and T. Riissanen, *Fuzzy Systems*, Lecture notes prepared for courses at Department of computing Science at Aumea University, Sweden, February, 2000.

[16] H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions", *IEEE Transactions on Software Engineering*, pp. 528-562, 1995.

[17] I. King and T. K. Lau, "Performance analysis of clustering algorithm for information retrieval in image databases", *International Joint Conference on IEEE World Congress on Computational Intelligence*, 1998, pp.932-937.