

การระบุ การกำหนดลักษณะจำเพาะและการจำแนกซอฟต์แวร์คอมพิวเตอร์โดยวิธีรูปนัย  
และโครงข่ายประสาทเทียม



นายสาธิษฐ์ นากกระแสร์

สถาบันวิทยบริการ

จุฬาลงกรณ์มหาวิทยาลัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรดุษฎีบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์ ภาควิชาคณิตศาสตร์

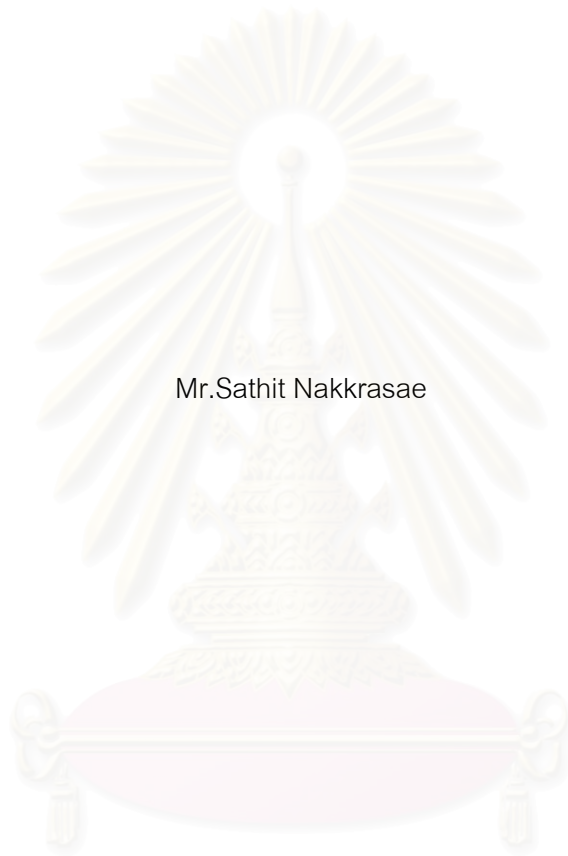
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2547

ISBN 974-17-5898-7

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

SOFTWARE COMPONENT IDENTIFICATION SPECIFICATION AND CLASSIFICATION USING  
FORMAL METHOD AND ARTIFICIAL NEURAL NETWORKS



Mr.Sathit Nakkrasae

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

A Dissertation Submitted in Partial Fulfillment of the Requirements  
for the Degree of Doctor of Philosophy in Computer Science

Department of Mathematics

Faculty of Science

Chulalongkorn University

Academic year 2004

ISBN 974-17-5898-7



นายสาธิษฐี นากกระแสร : การระบุ การกำหนดลักษณะจำเพาะและการจำแนกซอฟต์แวร์คอมโพเนนต์โดยวิธีรูปนัยและโครงข่ายประสาทเทียม. (SOFTWARE COMPONENT IDENTIFICATION SPECIFICATION AND CLASSIFICATION USING FORMAL METHOD AND ARTIFICIAL NEURAL NETWORKS) อ. ที่ปรึกษา : ผู้ช่วยศาสตราจารย์ ดร. พีระพนธ์ โสพัศสถิตย์, 102 หน้า. ISBN 974-17-5898-7.

วิทยานิพนธ์นี้ได้นำเสนอการระบุ การกำหนดลักษณะจำเพาะของซอฟต์แวร์คอมโพเนนต์โดยวิธีรูปนัยและนำเอาวิธีโครงข่ายประสาทเทียมมาจำแนกซอฟต์แวร์คอมโพเนนต์ในฐานข้อมูลตามภาวะคล้าย มีการนำเสนอวิธีต่างๆ 3 วิธีประกอบด้วย อาร์พีซีแอล (RPCL) เอฟเอสซี (FSC) และ เอสโอม (SOM) เมื่อแยกกลุ่มได้แล้ว จะนำเอาตัวกลางของแต่ละกลุ่มไปสร้างเป็นดัชนีเข้าถึงข้อมูล โดยอาจจะเป็นดัชนีแบบไม่มีลำดับชั้น (non-hierarchy) หรือ ดัชนีแบบมีลำดับชั้น (hierarchy) เรียกขั้นตอนนี้ว่าการจำแนกชั้นหยาบ (coarse grain classifications)

ในการศึกษาครั้งนี้ มีการวิเคราะห์โดยตัววัดประสิทธิภาพเพื่อเปรียบเทียบวิธีที่นำเสนอ ซึ่งประกอบด้วย ค่าการเรียกกลับคืน (recall), ค่าความถูกต้องเที่ยงตรง (precision), และเวลาการฝึกฝนของโครงข่ายประสาทเทียม (training time) สรุปได้ว่า วิธีอาร์พีซีแอล (RPCL) เป็นวิธีที่เหมาะสมต่อการนำมาใช้จำแนกซอฟต์แวร์คอมโพเนนต์ ขั้นตอนถัดมาคือการค้นคืน โดยกลุ่มของซอฟต์แวร์คอมโพเนนต์จะถูกเลือกขึ้นมาเมื่อตัวกลางของกลุ่มมีภาวะคล้ายกับตัวที่ต้องการมากที่สุด และกลุ่มดังกล่าว จะถูกนำมาคัดเลือกหาตัวที่เหมาะสมมากที่สุดในการจำแนกชั้นละเอียด (fine grain classifications) วิธีการและขั้นตอนที่นำเสนอนี้เหมาะสำหรับข้อมูลที่มีมิติจำนวนมาก และถือเป็นแนวทางพื้นฐานสำหรับการทำงานแบบอัตโนมัติต่อไป

## สถาบันวิทยบริการ จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา	<b>คณิตศาสตร์</b>	ลายมือชื่อนิสิต.....
สาขาวิชา	<b>วิทยาการคอมพิวเตอร์</b>	ลายมือชื่ออาจารย์ที่ปรึกษา.....
ปีการศึกษา	2547	

# # 4373844823 : MAJOR COMPUTER SCIENCE

KEY WORD: Software Component, Identification, Specification, Classification, Formal Method,  
Knowledge Engineering, Neural Networks

SATHIT NAKKRASAE: SOFTWARE COMPONENT IDENTIFICATION  
SPECIFICATION AND CLASSIFICATION USING FORMAL METHOD AND  
ARTIFICIAL NEURAL NETWORKS. THESIS ADVISOR: AISS. PROF. PERAPHON  
SOPHATSATHIT, Ph.D., 102 pp. ISBN 974-17-5898-7.

This dissertation presents a formal approach for identification and specification based on the well-established object-oriented paradigm then employs artificial neural network (ANN) to classify software component repository into similar component cluster. This clustered component repository is subsequently indexed using non-hierarchical and hierarchical indexing based on three unsupervised neural network techniques, namely, Rival Penalized Competitive Learning (RPCL), Fuzzy Subtractive Clustering (FSC), and Kohonen's Self-Organizing feature Map (SOM). This step is referred to as coarse grain classification.

In this study, analysis of the proposed the approach has been conducted to measure their efficiency in terms of precision, recall, and training time. The results confirmed that Rival Penalized Competitive Learning (RPCL) was the superior technique. Subsequent retrieval of software component belonging to the cluster partition whose center is closest to the requirements can thus be retrieved and participated in selecting the most suitable software component at the fine grain level. Consequently, this approach not only is suitable for multidimensional data, but also furnishes a basis for machine learning applications.

Department **Mathematics**

Student's signature.....

Field of study **Computer Science**

Advisor's signature.....

Academic year **2004**

## Acknowledgements

First and foremost, I would like to thank the Thai Government who sponsors the research scholarships. I would like to express my eternal gratitude to my supervisor, Asst. Prof. Dr. Peraphon Sophatsathit for his academic guidance, emotional support, encouragement, and patience on the work of this dissertation. I would also like to sincerely thank my examining committee, Prof. Dr. Chidchanok Lursinsap, Assoc. Prof. Dr. Somchai Prasitjutrakul, Asst. Prof. Dr. Somjai Boonsiri, and Dr. Suraphan Meknawin, for their comments and useful suggestions on this thesis.

My special appreciation goes to Assoc. Prof. Dr. William R. Edwards, Jr. of CACS at University of Louisiana at Lafayette U.S.A., Dr. Khamron Sunat, Dr. Wiphada Wettayaprasit, and Ms. Atchara Mahaweerawat of AVIC at Chulalongkorn University for their insight and constructive suggestions of my dissertation.

Finally, I am deeply grateful to my parents, grandparents, brothers, and my wife for their love, support, and patience during the four years period.

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## Table of Contents

Thai Abstract . . . . .	iv
English Abstract . . . . .	v
Acknowledgements . . . . .	vi
List of Tables . . . . .	x
List of Figures . . . . .	xi
1 Introduction . . . . .	1
1.1 Background and Motivation . . . . .	1
1.2 Contributions . . . . .	3
1.3 Dissertation Organization . . . . .	4
2 Related Works . . . . .	5
3 Software Component Configuration . . . . .	8
3.1 Component Identification . . . . .	9
3.2 Component Modeling Technique (CMT) . . . . .	14
3.2.1 Structural Model . . . . .	15
3.2.2 Functional Model . . . . .	15
3.2.3 Behavioral Model . . . . .	16
3.3 Component Specifications . . . . .	17
4 Software Component Formulation . . . . .	27
5 Software Component Classification Model . . . . .	32
5.1 Software Reuse Model . . . . .	32
5.2 Cluster Center Establishment . . . . .	34
5.2.1 Conventional Statistical Approach . . . . .	34
5.2.2 Self-Organizing Map . . . . .	35
5.2.3 Fuzzy Subtractive Clustering . . . . .	37

5.2.4	Rival Penalized Competitive Learning . . . . .	39
6	Experiment . . . . .	42
6.1	Data Collection . . . . .	42
6.2	Method Learning and Clustering . . . . .	43
6.3	Evaluation . . . . .	44
7	Component Classification and Retrieval . . . . .	51
7.1	Non-hierarchical Indexing Classification of Software Components . . . . .	51
7.2	Hierarchical Indexing Classification of Software Components . . . . .	52
7.3	Software Component Selection Technique (Fine Grain Level) . . . . .	55
7.4	Application of Neural Network . . . . .	56
7.4.1	Coarse Grain RPCL Selection . . . . .	56
7.4.2	Coarse Grain FSC Selection . . . . .	61
7.4.3	Fine Grain RPCL Selection . . . . .	61
7.4.4	Fine Grain FSC Selection . . . . .	66
8	Conclusion . . . . .	67
8.1	Concluding results . . . . .	67
8.2	Future work . . . . .	68
	References . . . . .	70
	APPENDICES . . . . .	76
	Appendix A . . . . .	77
A-1	Abstract Data Type Repository (ADTR) . . . . .	77
A-1.1	Structural Property Equivalence . . . . .	77
A-1.2	Functional Property Equivalence . . . . .	79
A-1.3	Behavioral Property Equivalence . . . . .	81
A-2	Component Requirement Example . . . . .	83
	Appendix B . . . . .	86



B-1 Z language and Notations . . . . .	86
B-1.1 Syntax and Notations . . . . .	86
B-1.2 Schema . . . . .	88
VITA . . . . .	90



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## List of Tables

6.1	Comparison of precision performance . . . . .	45
6.2	Comparison of recall performance . . . . .	45
6.3	Comparison of average computation time . . . . .	46
6.4	Comparison of derived cluster number . . . . .	46
6.5	Comparison of the four methods of clustering performance . . . . .	46
6.6	Unsupervised Methods Comparison of precision performance . . . . .	48
6.7	Unsupervised Methods Comparison of recall performance . . . . .	48
6.8	Unsupervised Methods Comparison of average computation time . . . . .	49
6.9	Comparison of the three methods of clustering performance . . . . .	49
7.1	RPCL recall and precision performance of 10 random initialized centers.	57
7.2	RPCL recall and precision performance of 20 random initialized centers.	58
7.3	RPCL recall and precision performance of 30 random initialized centers.	58
7.4	RPCL recall and precision performance of 40 random initialized centers.	59
7.5	RPCL recall and precision performance of 50 random initialized centers.	60
7.6	RPCL recall and precision performance comparison. . . . .	61
7.7	FSC recall and precision performance of 0.15-0.20 rejection ratio ( $\eta$ ) value	62
7.8	FSC recall and precision performance of 0.25 rejection ratio ( $\eta$ ) value . .	63
7.9	FSC recall and precision performance of 0.30 rejection ratio ( $\eta$ ) value . .	64
7.10	FSC recall and precision performance of 0.35-0.50 rejection ratio ( $\eta$ ) value	64
7.11	FSC recall and precision performance comparison . . . . .	65
7.12	RPCL software component selection with different degree of significance.	65
7.13	FSC software component selection with different degree of significance .	66

## List of Figures

3.1	Component Model. . . . .	9
3.2	Component Relationship. . . . .	13
3.3	Component Model Example. . . . .	16
3.4	Component Interaction. . . . .	17
3.5	Software Component Specification. . . . .	22
3.6	Signature Specification. . . . .	23
3.7	Interaction Specification. . . . .	23
3.8	Behavior Specification. . . . .	24
3.9	Example of Structural Specification. . . . .	24
3.10	Example of Signature Specification. . . . .	25
3.11	Example of Interaction Specification. . . . .	25
3.12	Example of Behavior Specification. . . . .	26
4.1	Software Component Matrix Representation. . . . .	30
4.2	Matrix Calculation Algorithm. . . . .	31
5.1	Software Reuse Model. . . . .	33
6.1	Box Plot of precision, recall, average computation time, and number of derived clusters. . . . .	47
6.2	Box Plot of precision, recall, and average computation time. . . . .	50
7.1	The n cluster partitions generated by ANN technique. The dots are the repository software component whereas the crosses are the centers. The center index is used for indexing . . . . .	51

7.2	$C_3$ and $C_4$ are the clusters inside $C_1$ while $C_5$ and $C_6$ are clusters inside $C_2$ . The black dots represent repository software components (property vector), the crosses represent the center of each cluster . . . . .	54
7.3	Indexing structure for hierarchical clustering. $C_0$ is the root node which contains all software components in repository. $Dist(RC, C_i)$ is the Euclidean Norm Distance between the nearest-neighbor designated component $RC$ and the center $C_i$ of cluster $C_i$ . . . . .	55
B-1	Format Schema Type. . . . .	88
B-2	Example of Signature Specification. . . . .	89



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

# CHAPTER I

## Introduction

### 1.1 Background and Motivation

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 an important role in the development process. One important example is knowledge engineering [33], a term that is used in broad sense to encompass artificial intelligence, computational intelligence, knowledge base, data mining, and machine learning. Many typical software development benefits from these disciplines. Two popular methodologies of high potential impact on software development productivity and reliability are formal specification and reuse. Reuse has a larger potential impact at early stages in the development process while formal specification allows increase in reuse automation process. Adoption of these technologies requires significant investment in building libraries, educating designers, and constructing domain models. Therefore, to make this development investment economically attractive, there is a need for methodologies and automated tools to support development activities, especially design reuse, at the specification level to attain a usable software product.

From software engineering standpoint, 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 from scratch; and (b) increased confidence in the solution because of its successful reuse history. 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 practice, reuse is popularly applied in the 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. Thus, this design representation must be able to efficiently support component retrieval, adaptation, and verification. For this reason, software components are commonly stored in repository with the help of knowledge engineering, neural networks, and fuzzy logic techniques.

Classification is automated by using formal descriptions to define features describing a component. These definitions control the classification scheme in place of a human domain expert. A feature is assigned to a component if a corresponding predicate is logically implied by the component specification. This method of assigning features ensures that components more likely to match reuse criteria will have similar feature sets. Full scale specification matching, using automated theorem proving, can be applied to the retrieved components to precisely evaluate their reusability.

Automatic software component classification is one of the prime objectives in component-based development. Existing classification approaches focus on creating a component structure index to aid in the deposit and retrieval processes. The underlying principle rests on intelligent clustering mechanism where neural network technique seems to be the norm of choices.

Such automated component classification support is conducive toward the application of software reuse. Developers can make use of available classification support to acquire the desired components for the task at hand. As a consequence, software development will become more productive at higher quality and less laborious than existing state-of-

the-practice.

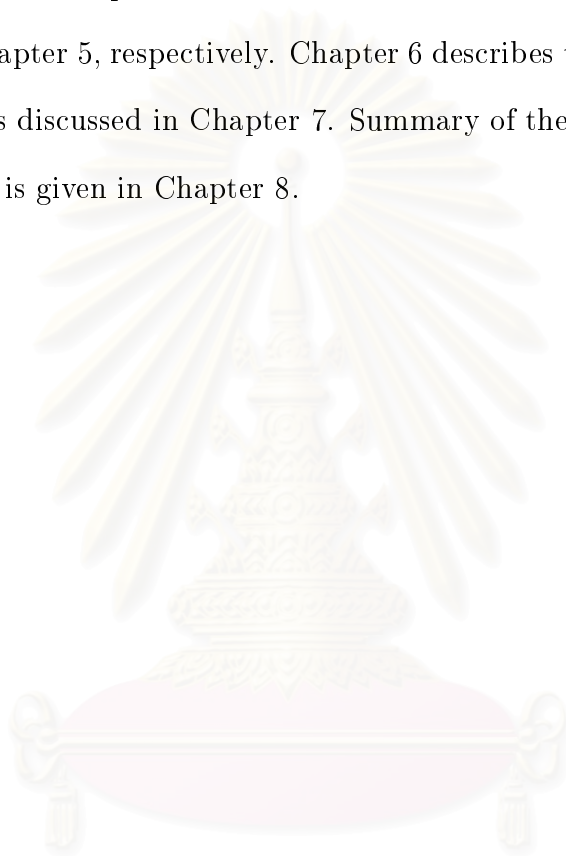
## 1.2 Contributions

This dissertation proposes a formal method for software component specification based on three identification properties, namely, structural, functional, and behavioral properties. These properties are used to classify software components to be stored in a repository. The proposed approach focuses on establishing a rigorous framework to arrive at a formal method for software component specification and employs neural network techniques to classify for software components. Thus, the main contributions of this work are:

- (1) Formally define the "three views of the system" of software component specifications using Z language based on object-oriented paradigm;
- (2) Present a software component matrix representation based on the above well-defined software component specification;
- (3) Gather and classify sample software components in coarse and fine grains based on formal definitions so established for archival purpose according to their attributes. The classification process calls for intelligent knowledge engineering techniques to sort out the components under investigation, for instance, Self Organizing Map (SOM) Network, Fuzzy Subtractive Clustering (FSC), or Rival Penalized Competitive Learning (RPCL) technique;
- (4) Conduct component retrieval with the help of an indexing structure constructed according to component clusters in two different fashions, namely, 1) Non-hierarchical indexing and 2) Hierarchical indexing; and
- (5) Analyze classified components and establish a retrieval process based on their reuse component specifications, classification techniques, and retrieval procedures.

### 1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Some related work is presented in Chapter 2. Chapter 3 describes software component configuration. The principal and fundamental software component formulation and classification process are elucidated in Chapter 4 and Chapter 5, respectively. Chapter 6 describes the experiment. Component retrieval process is discussed in Chapter 7. Summary of the proposed method, together with future work, is given in Chapter 8.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย



## CHAPTER II

### Related Works

Software component library construction, as presented in [28] using information retrieval techniques to automatically assemble various components, can be divided into two steps. First, component attributes are extracted from natural language documentation through an indexing algorithm. Second, a hierarchical indexing structure is generated using a clustering technique based on analysis of natural language documentation obtained from manual pages or comments. Despite being a rich source of conceptual information, natural language is not rigorous for specifying the behavioral specification of software components. As such, a formal specification language can serve as precise definitions and a means for communication among software clients, specifiers, and implementers [40].

The MAPS system [31] applies formal specifications termed case-like expressions to specify software modules. MAPS exploits the unification capability to search through reusable modules in the library. Jeng, et. al, [19] utilizes formal methods to specify software component in a hierarchically organized library. However, both approaches still have granularity limitations.

Hong and Kim [18] proposed three classification methods to systematically organize the components according to their formal specification so developed, namely, the enumerative method, the facet method, and the information retrieval method using clustering technique. Some of the related works [1], [5], [6], [7], [18], [20], [37], [41], [44] also

investigate on such issues as component identification, component reuse, formal method for component reuse, and classification of software components.

A popular method for describing repositories of reusable software components is a faceted classification scheme [32]. 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 classifies the components. 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 [32]. The informality and imprecision of these classification schemes complicate the automation of the overall classification, as well as the reuse process. Automation of the classification process requires reverse engineering from source code. The imprecision of this 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 various software design problems [11], [14], [19], [20], [34], [36], [45], [46]. There are many benefits to applying formal methods to software reuse. First, formal specifications provide an explicit representation of structure, function, and behavior of the software component, independent of many implementation details. This is valuable because the structure, function, and behavior of the designed software components are the primary concern in reusability. Additional benefits stem from the expressiveness of formal specification languages that allows precision beyond that of faceted classification, e.g., equivalent specification can be derived to yield equivalent properties of software components. 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 operations used in the system. As such, formal reasoning based on the domain model can be used logically to verify the reusability of a software component, which can serve as a foundation of machine learning in automated software classification and reuse.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## CHAPTER III

### Software Component Configuration

The fundamentals of software component usage primarily rests on object-oriented concepts, but aim at much larger specification than that of a single object. Despite the fact that the concepts encompass various reusability provisions, component classification and archival principles are still difficult to administer for reuse. In a software repository, software component retrieval is usually accomplished through some classification schemes [6], [7], [18], [28]. The users must supply as much relevant descriptions as possible for the closest match of the desired component. The underlying implementation details of retrieval process are often transparent to the users. As such, it is imperative for every archived component that the components be correctly identified, classified, and stored to effectively support subsequent retrieval.

To achieve such goals, a well-defined component specification and classification framework based on conventional object-oriented paradigm, formal approach, and computational intelligence is proposed for systematic component storage and retrieval.

The process requires software component identification and specification to be rigorously defined which is based on three aspects of software component, namely, structural, functional, and behavioral properties. A component modeling technique (CMT) is employed to specify and construct a visual component model using Unified Modeling Language (UML) [29]. The essence of this approach is to capture as much information pertaining to the fundamental properties of the component as possible. Such information

is then described through formal specification by means of Z language [35], [41].

### 3.1 Component Identification

The basic premise of software component rests on the notions of reuse building blocks. Every component is made up of zero or more subcomponent in the form of concrete classes. Thus, a component can be regarded as a self-contained complex entity. Figure 3.1 illustrates the composition of a component having a subcomponent, a class, or a family of subcomponents or classes, common data, and common methods. Each part is linked to its structurally related subcomponents or classes defined by some interactions, which are eventually connected to the outside world via the interface. Thus, any component can be linked to other components via different interactions. The behavioral interactions among these subcomponents or classes are described in the form of transactions. This procedure is recursively applied to create and identify component interrelationships.

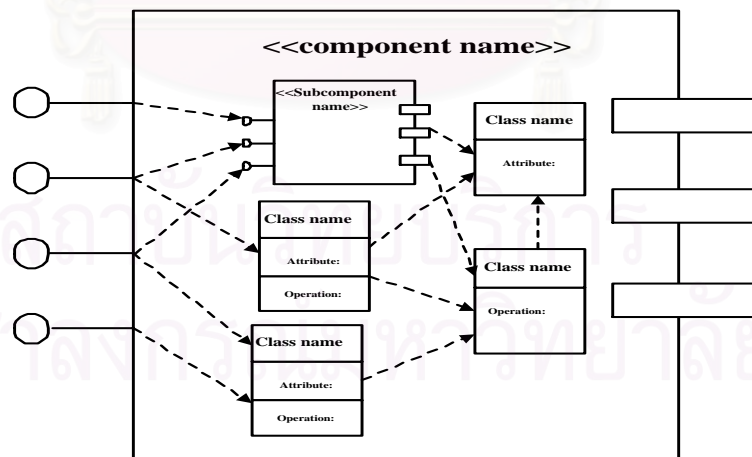


Figure 3.1: Component Model.

The following definitions defined in [4], [38], [36] are reiterated here for use in subse-

quent library analysis and composition.

*Class*: is a general specification or description for a set of objects that share a common behavior. It defines the properties (attributes), methods (operations), messages (requests for operations), relationship, and semantics of a similar group of objects. Classes may themselves be objects (entities). Classes usually are templates from which individual objects can be created.

*Abstraction*: is a design technique that focuses on the essential aspects of an entity and ignores or conceals less important or non-essential aspects. It is an important tool concerning with both the “attributes” and “behavior (operation)” for simplifying a complex situation to a level where analysis, experimentation, or understanding can take place.

*Object*: is a concept, an abstraction, and a thing with sharp boundaries and meaning for an application, all of which represent an instance of a class. An object has identity, state, and behavior.

*Common Class*: is a class that can be requested from other classes, consisting of

- Common Data Class: is the class that collects common data of others classes in the component and can be accessed by default operations (read, update, delete, and insert); and
- Common Method Class: is the class that collects the common operations of other classes in the component from which the operations can be requested.

*Relationship*: is an association among one or more objects or components. It is either a fact or a reference. Occasionally, a relationship is nested (or objectified) so that the relationship itself can participate in other relationships. There are class relationship and component relationship. The former composes of dependency, association, generalization, and realization [26], [29], whereas the latter associates components that may either

be abstract (specifications) or concrete (implementations). An *abstract component* describes functional behavior—*what* services a subsystem provides. A *concrete component* describes an implementation—*how* a subsystem’s services are provided. Such relationships entail data abstraction, information hiding, multiple implementation, and self-contained descriptions of component behavior. This work focuses on component relationship that is further characterized as implements, uses, and extends relationships.

The first fundamental component relationship upon which all others rely for meaning and utility is *implements*. Implements describe the key relationship between an implementation (a concrete component) and a specification (an abstract component). The implements relationship may be defined informally as follows:

- A concrete component  $X$  *implements* abstract component  $Y$  if and only if  $X$  exhibits the behavior specified by  $Y$  [12].

This is a fairly intuitive relationship between a specification and an implementation. However, a fully formal and general definition of the implements relationship is very intricate and has been the subject of much research. Implements express a dependency between two components in the following sense. If component  $X$  implements component  $Y$ , then  $X$  depends on  $Y$  to provide an abstract, client-level description of its behavior – a “cover story” hiding all implementation details.

Justifying a claim that  $X$  implements  $Y$  may require significant effort, especially if done formally, whereby considerable leverage can be gained from doing so. If two different concrete components both implement the same abstract component, then either of them may be used in a context requiring the functional behavior described by the abstract component. In this case, the two implementations are *behaviorally substitutable* with respect to the specification they both implement. The two implementations may differ in non-functional characteristics such as execution time, space requirements, cost,

warranty, legal use restrictions, level of trust in correctness, and so forth.

While *implements* relationship describes a dependency between an implementation and a specification, *uses* relationship describes a dependency that exists between two different abstractions. The relation name *uses* actually applies to three different yet closely related component relationships. *Uses* may describe a dependency between two implementations, between two specifications, or between an implementation and specification. The last of these three relationships is defined as follows:

- A concrete component  $X$  *uses* abstract component  $Y$  if and only if  $X$  depends on the behavior specified by  $Y$  [12].

This form of the *uses* relationship expresses a *polymorphic* relationship between implementations. Any component that implements abstract component  $Y$  may serve as the actual concrete component used by instances of component  $X$ . Thus, this form of *uses* reduces unnecessary dependencies between components. Note that none of the three *uses* relationships is equivalent to the *is-a-part-of* relationship. If implementation  $X$  uses implementation  $Y$ ,  $Y$  may or may not be a part of the data representation of  $X$ . The client wishing to use component  $X$  does not need to know the specific role of  $Y$  in relation to  $X$  in order to use component  $X$ .

The third component relationship is *extends*. The name *extends* applies to two different, yet closely related component relationships. One *extends* expresses a dependency between two abstract components. The other expresses a dependency between two concrete components. Both *extends* relationships may be defined informally as follows:

- A component  $X$  *extends* component  $Y$  if and only if all of the interface and behavior described by  $Y$  is included in the interface and behavior described by  $X$  [12].



This definition conveys the intuitive meaning of extends, that is, component  $X$  extends the interface and behavior of component  $Y$ . It implies the essential property of behavioral substitutability. If abstract component  $X$  extends abstract component  $Y$ , concrete component  $X1$  implements  $X$ , and concrete component  $Y1$  implements  $Y$ , then  $X1$  is behaviorally substitutable for  $Y1$  with respect to  $Y$ . Note that  $Y1$  is *not* behaviorally substitutable for  $X1$  with respect to  $X$  in this case. Thus behavioral substitutability is a ternary relationship, not a binary equivalence relation. In some cases, the extends relationship may sound very much like an inheritance relationship. It is important to understand that *extends* is *not* an inheritance relationship. Extends describes a *behavioral* relationship between two components, whereas *inherits – from* does not. Furthermore, while inheritance may be a convenient programming language mechanism for expressing structural aspects of the extends relationship, extends may be encoded in programming languages without any use of inheritance. Figure 3.2 shows the component relationship.

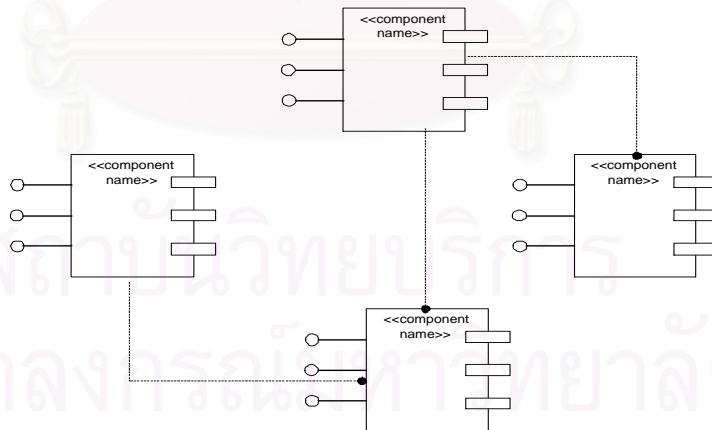


Figure 3.2: Component Relationship.

*Interface*: defines the access points to components that are a collection of operations used to specifying service of a component. In an interface, each operation's semantic

is specified. This specification provides implementation of the interface and the clients who use the interface. The interface of a component is important for the composition and customization of component by the users since it allows the users to find suitable components and to understand their purpose, functionality, usage, and restrictions.

The description of a component interface consists of a signature and a behavior part. The signature defines operations or methods, parameters (type and size), and return values. The behavior part, on the other hand, describes how the component acts or reacts to requests (messages) from other components.

Components can implement (export) one or more interfaces and can also use (import) interfaces from other components. Therefore, export interfaces correspond to the services a component provides and import interfaces correspond to the services a component needs.

Interface allows the heterogeneous components to interoperate. At a minimum, the components' interface should provide a contract that states both the services a component offers and the services the component requires in fulfilling its commitment.

### **3.2 Component Modeling Technique (CMT)**

Component Modeling Technique adheres to the “three views of a system” paradigm, namely, structure, behavior, and function. The static model is termed the structural model; the representation of system dynamic is termed the behavioral model; and the representation of component process is termed the functional model. Each model is used to build a set of component views defined on a domain. As stated in Section 3.1, it is possible to recursively represent embedded component view based on the covering domain. Configuration detail of each model will be discussed in the sections that follow to establish a formal approach which will denote the component specifications.

### 3.2.1 Structural Model

The structural model in CMT consists of:

- Component box with small rectangles representing methods on one side and little lollipops connected by solid line representing the interface on the other side;
- Internal box representing classes with a mandatory class name, optional attribute name or declarations, and optional operation names and declarations;
- Dashed line representing dependency relationship;
- Solid line representing association relationship;
- Solid line with open arrow representing generalization relationship;
- Dash line with close arrow representing realization relationship;
- Solid line with close arrow representing uses relationship;
- Dash line with open diamond representing implements relationship;
- Dash line with close diamond representing extends relationship; and
- Bracketed textual items denoting constrains.

Figure 3.3 shows the structural model of an example component `Array_Stack_Data_Structure` consisting of two subcomponents (`Array` and `Stack`), common classes (`Method` class and `Attribute` class), and their interactions.

### 3.2.2 Functional Model

The functional model in CMT specifies how operations derive output values from input values without regarding to the order of computations. The original illustration utilizes

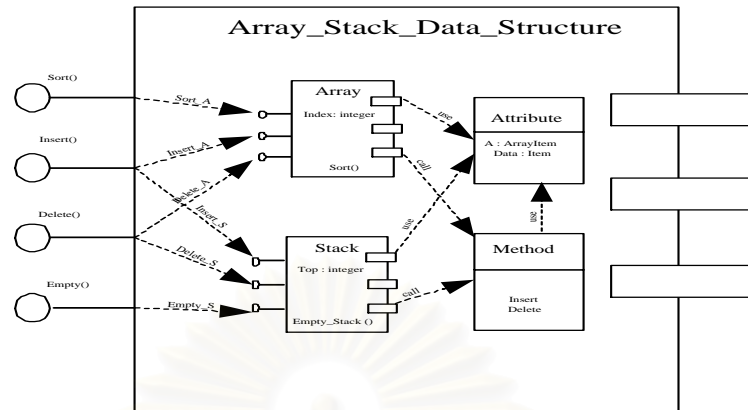


Figure 3.3: Component Model Example.

data flow diagrams with directed named edges linking process nodes to process nodes or data nodes. The directed named edges denote data flowing in and out of each process. Throughout this document, UML diagrams are employed in place of classical DFD for the discussion.

### 3.2.3 Behavioral Model

The behavior model in CMT refines the activity model, proposed capabilities, and component capability requirements. Some capabilities may be fulfilled by a combination of multiple behaviors. The behavior model, as depicted in Figure 3.3, is divided into two parts:

- Component interaction part that shows the messages (behavior name and/or message argument) sent between components (subcomponents, classes).
- State transition part that presents the state transitions of each component (subcomponent, class) or the interaction between the components (or subcomponent, class).

A state can be viewed as an equivalent class or subcomponent of attribute values and association values of an object class or subcomponent. This equivalent class or subcomponent is formed under a relation of “same behavior” with respect to the real world situation or requirements being modeled. The state thus defines a “transitory subtype” of the object class or subcomponent to which it belongs [30].

State description can be incorporated in the graph, based on David Harel, et. al [26], that consists of a name, optional internal actions (instantaneous operations), activities (operations that take time to complete), optional entry (trigger) and exit actions. Edges represent transitions between states. Figure 3.4 depicts a behavioral model of a component interaction.

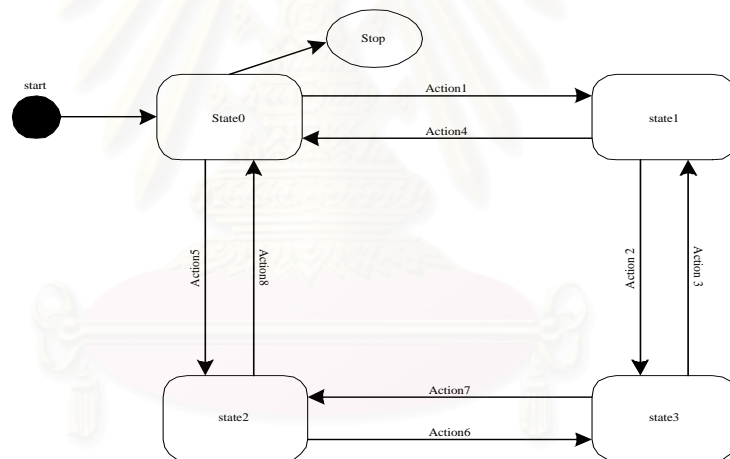


Figure 3.4: Component Interaction.

### 3.3 Component Specifications

In order to establish formal component specifications, the syntax and semantics must be analyzed and represented. A mathematical representation by means of Z specification language [35], [41] is given to denote formal component specification. A software

component (hereafter SC) is a 5-tuple element having the form:

$$SC := (n, SSC, CL, S, In)$$

where

$n$  : a unique name of the software component

$SSC$  : (sub-software component) a set of software component

$CL$  : member class ( $M_{CL}$ ) and common class ( $C_{CL}$ ) (both are sets of classes) encompassing:

$$CL = M_{CL} \cup C_{CL}$$

$C_{CL}$  : method class ( $Met_C$ ) and attribute class ( $Atb_C$ ) (both are sets of classes)

$$C_{CL} = Met_C \cup Atb_C$$

$S$  : a set of 6-tuple signature describing the operation having

$$S := \{ (n_O, In_O, Local_O, Out_O, Pre_O, Post_O) \}$$

where

$n_O$  : name of operation

$In_O$  : set of input parameters

$Local_O$  : set of local variables

$Out_O$  : set of output parameters

$Pre_O$  : pre-condition expression

$Post_O$  : post-condition expression

$In$  : a set of 3-tuple interaction describing the transaction such that

$$In := \{ (S_I, B_H, D_I) \}$$

where

$S_I$  : interaction source

$B_H$  : a set of 4-tuple behavioral part describing all behavior between  $S_I$  and  $D_I$   
having

$$B_H := \{ (nb, Start, St\_Ac\_St, Stop) \}$$

where

$nb$  : name of behavior

$Start$  : set of states

$St\_Ac\_St$  : set of cartesian product between state, action, and state

$Stop$  : set of states

$D_I$  : interaction destination

Based on the above definitions, standard Z notations (See Appendix B for details) for software component are developed.

Step 1 : Denoting basic type sets which are declared and enclosed in square brackets.

[operation name] [input parameter]

[output parameter] [local variable]

[expression] [behavioral name]

[component name] [class name]

[request] [action]

[state] [class]

[string]

Step2 : Denoting composite/aggregate types in the form of set.

member class : F class

method class : F class

attribute class : F class

common class : method class  $\cup$  attribute class

interaction source : component name  $\cup$  class name  $\cup$  operation name

interaction destination : component name  $\cup$  class name  $\cup$  operation name

ProperComponent\_name : set of component name

ProperOperational\_name : set of operation name

ProperBehavior\_name : set of behavior name

Step3 : Denoting software component.

Software component ( $\lambda$ ) ==  $\{SC : (n, SSC, CL, S, In) \mid$

$\forall n$  : component name

$SSC$  : P (set of software component)

$CL$  : P class

$S$  : P  $O_P$

$In$  : P  $T_r \bullet$

$(n, SSC, CL, S, In) \in SC \Rightarrow$

$n \in \text{ProperComponent\_name} \wedge$

$SSC \in F$  (set of software component )  $\wedge$

$CL \in \text{common class} \cup \text{member class} \wedge$

$S \in F O_P \wedge In \in F T_r \}$

Step4 : denoting the signature.

Signature ( $\sigma$ ) ==  $\{ O_P : \text{set of } (n_O, In_O, Local_O, Out_O, Pre_O, Post_O) \mid$

$\forall n_O$  : operation name

$In_O$  : P input parameter

$Local_O$  : P local variable

$Out_O$  : P output parameter

$Pre_O$  : expression

$Post_O$  : expression  $\bullet$

$(n_O, In_O, Local_O, Out_O, Pre_O, Post_O) \in O_P \Rightarrow$

$n_O \in \text{ProperOperational\_name} \wedge$



$In_O \in F$  input parameter  $\wedge$   
 $Local_O \in F$  local variable  $\wedge$   
 $Out_O \in F$  output parameter  $\wedge$   
 $Pre_O \in$  expression  $\wedge$   
 $Post_O \in$  expression }

Step5 : denoting the interaction.

Interaction ( $\iota$ ) == {  $Tr$  : set of  $(S_I, B_H, D_I)$  |  
 $\forall S_I$  : interaction source  
 $B_H$  :  $\beta$   
 $D_I$  : interaction destination •  
 $(S_I, B_H, D_I) \in Tr \Rightarrow$   
 $S_I \in$  interaction source  $\wedge B_H \in \beta \wedge$   
 $D_I \in$  interaction destination }

Step6 : denoting the behavior.

Behavior ( $\beta$ ) == {  $B$  : set of  $(nb, Start, St\_Ac\_St, Stop)$  |  
 $\forall n_{Bh}$  : behavior name  
 $Start$  : state  
 $St\_Ac\_St$  : P (state  $\times$  action  $\times$  state)  
 $Stop$  : state •  
 $(n_{Bh}, Start, St\_Ac\_St, Stop) \in B \Rightarrow$   
 $n_{Bh} \in$  ProperBehavior\_name  $\wedge Start \in$  state  $\wedge$   
 $St\_Ac\_St \in F$  (state  $\times$  action  $\times$  state)  $\wedge Stop \in$  state }

Step7 : Putting all notations (step3 - step6) into Z schema as shown in Figure 3.5-3.8

SpecifySoftwareComponent	
$n?$	: component name
$SSC?$	: P (set of software component)
$CL?$	: P class
$S?$	: P $O_p$
$In?$	: P $Tr$
$SC!$	: set of $\lambda$

---

$n? \in F \text{ string} \wedge SSC? \in F (\text{set of software component}) \wedge$   
 $CL? \in \text{common class} \cup \text{member class} \wedge$   
 $S? \in F O_p \wedge In? \in F Tr \wedge SC! \in \text{set of } \lambda$

Figure 3.5: Software Component Specification.

Other definitions such as constants in declaration part and conditions in predicate part can also be included. Figure 3.9-3.12 demonstrate a step by step component specification formulation of the Array\_Stack\_Data\_Structure example of Figure 3.3.

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

SpecifySignature	
$n_o?$	: operation name
$In_o?$	: P input parameter
$Local_o?$	: P local variable
$Out_o?$	: P output parameter
$Pre_o?$	: expression
$Post_o?$	: expression
$O_p!$	: $\mathcal{O}$
<hr/>	
$n_o?$	$\in$ F string $\wedge$
$In_o?$	$\in$ F input parameter $\wedge$
$Local_o?$	$\in$ F local variable $\wedge$
$Out_o?$	$\in$ F output parameter $\wedge$
$Pre_o?$	$\in$ expression $\wedge$
$Post_o?$	$\in$ expression $\wedge O_p! \in \mathcal{O}$

Figure 3.6: Signature Specification.

SpecifyInteraction	
$S_C?$	: interaction source
$B_H?$	: $B$
$D_C?$	: interaction destination
$Tr!$	: $\mathcal{L}$
<hr/>	
$S_C?$	$\in$ F string $\wedge$
$B_H?$	$\in B \wedge$
$D_C?$	$\in$ F string $\wedge$
$Tr!$	$\in \mathcal{L}$

Figure 3.7: Interaction Specification.

SpecifyBehavior	
$n_{Bh}?$	: behavior name
$Start$	: state
$St\_Ac\_St?$	: $P(\text{state} \times \text{action} \times \text{state})$
$Stop$	: state
$B!$	: $\beta$
<hr/>	
$n_{Bh}?$	$\in F \text{ string} \wedge Start \in \text{state} \wedge$
$St\_Ac\_St?$	$\in F (\text{state} \times \text{action} \times \text{state}) \wedge$
$Stop$	$\in \text{state} \wedge$
$B!$	$\in \beta$

Figure 3.8: Behavior Specification.

Specify Static_Sequential_Data_Structure	
$Array\_Stack\_Data\_Structure?$	: component name
$\{Array?, Stack?\}$	: $P(\text{set of software component})$
$\{Method?, Attribute?\}$	: $P \text{ class}$
$\{Sort()?, Insert()?, Delete()?, Empty()?\}$	: $P O_p$
$\{Sort\_A?, Insert\_A?, Delete\_A?...\}$	: $P Tr$
$Array\_Stack\_Data\_Structure \text{ Component!}$	: set of $\lambda$
<hr/>	
$Array\_Stack\_Data\_Structure?$	$\in F \text{ string} \wedge$
$\{Array?, Stack?\}$	$\in F (\text{set of software component}) \wedge$
$\{Method?, Attribute?\}$	$\in \text{common class} \cup \text{member class} \wedge$
$\{Sort()?, Insert()?, Delete()?, Empty()?\}$	$\in F O_p \wedge$
$\{Sort\_A?, Insert\_A?, Delete\_A?\}$	$\in F Tr \wedge$
$Array\_Stack\_Data\_Structure \text{ Component!}$	$\in \text{set of } \lambda$

Figure 3.9: Example of Structural Specification.

SpecifyInsert	
<i>Max</i> :	<i>N</i>
<i>Insert?</i>	: operation name
<i>{Data?}</i> :	<i>integer</i> : P input parameter
<i>{Index_A?}</i> :	<i>integer, Top_S?: integer, A?:ArrayType</i> : P local variable
<i>{A?}</i> :	<i>ArrayType</i> : P output parameter
<i>Pre?</i> :	<i>len(A) &lt; Max</i> : expression
<i>Post?</i> :	<i>len(A') = len(A) + 1 <math>\wedge</math> A'((len(A)) = Data <math>\wedge</math></i>
<i>(Index_A' = Index_A + 1 <math>\vee</math> Top_S' = Top_S + 1)</i>	: expression
<i>Insert Signature!</i>	: $\sigma$
<hr/>	
<i>Max</i> $\in$	<i>100 <math>\wedge</math></i>
<i>Insert?</i>	$\in$ F strings $\wedge$
<i>{Data?}</i> :	<i>integer</i> $\in$ F input parameter $\wedge$
<i>{Index_A?}</i> :	<i>integer, Top_S?: integer, A?:ArrayType</i> $\in$ F local variable $\wedge$
<i>{A?}</i> :	<i>ArrayType</i> $\in$ F output parameter $\wedge$
<i>Pre?</i> :	<i>len(A) &lt; Max</i> $\in$ expression $\wedge$
<i>Post?</i> :	<i>len(A') = len(A) + 1 <math>\wedge</math> A'((len(A)) = Data <math>\wedge</math></i>
<i>(Index_A' = Index_A + 1 <math>\vee</math> Top_S' = Top_S + 1)</i>	$\in$ expression $\wedge$
<i>Insert Signature!</i>	$\in$ $\sigma$

Figure 3.10: Example of Signature Specification.

SpecifyArray_Sort	
<i>Sort()</i> ?	: interaction source
<i>Sort_A?</i>	: <i>B</i>
<i>Array?</i>	: interaction destination
<i>Array_Sort Interaction!</i>	: $\zeta$
<hr/>	
<i>Sort()</i> ?	$\in$ F string $\wedge$
<i>Sort_A?</i>	$\in$ <i>B</i> $\wedge$
<i>Array?</i>	$\in$ F string $\wedge$
<i>Array_Sort Interaction!</i>	$\in$ $\zeta$

Figure 3.11: Example of Interaction Specification.

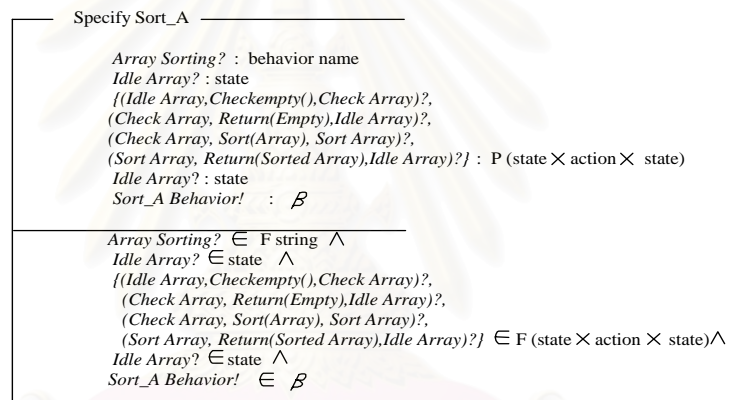


Figure 3.12: Example of Behavior Specification.

## CHAPTER IV

### Software Component Formulation

In order to represent software component for subsequent classification process, formal specifications are required to denote structural, functional, and behavioral properties, whereby accurate, succinct, and implementation independent details can be captured. The formulation is represented in a matrix form. As a consequence, a step-by-step process describing the transformation is given.

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

$$S = \{S_1, S_2, S_3, \dots, S_m\},$$

$$F = \{F_1, F_2, F_3, \dots, F_n\}, \text{ and}$$

$$B = \{B_1, B_2, B_3, \dots, B_p\}, \text{ respectively.}$$

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

$$S_i = \{S_{i,1}, S_{i,2}, S_{i,3}, \dots, S_{i,u_i}\}; \quad 1 \leq i \leq m \text{ and } S_{i,j} \in D(S_i)$$

$$F_i = \{F_{i,1}, F_{i,2}, F_{i,3}, \dots, F_{i,v_i}\}; \quad 1 \leq i \leq n \text{ and } F_{i,j} \in D(F_i)$$

$$B_i = \{B_{i,1}, B_{i,2}, B_{i,3}, \dots, B_{i,w_i}\}; \quad 1 \leq i \leq p \text{ and } B_{i,j} \in D(B_i)$$

and  $u_i$ ,  $v_i$ , and  $w_i$  denote the number of members in  $S_i$ ,  $F_i$ , and  $B_i$ , respectively. Each members is ordered from left to right, followed by the software component specification [30]. Thus,  $D(S_i)$ ,  $D(F_i)$ , 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 belonging to the class *LIFO*. This formulation entails a set of equivalent classes to be predefined within a component repository system by the designer or the developer.

Two preamble assumptions of this component repository stipulate that the number of elements in a set of equivalent classes for a given software component be finite, and that the number of each equivalent property classes of the components be known. Denote the number of each structural, functional, and behavioral equivalent class property by  $T_{S_i}$ ,  $T_{F_j}$ , and  $T_{B_k}$ , where  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ , and  $1 \leq k \leq p$ , respectively. Define the property matrix representation as follows:

$$Col_s = \max(T_{S_i}, 1 \leq i \leq m), \quad Row_s = m$$

$$Col_f = T_{F_1}, \quad Row_f = 1 + \sum_{i=2}^n T_{F_i}$$

$$Col_b = T_{B_1}, \quad Row_b = 1 + \sum_{i=2}^p T_{B_i}$$

The software component matrix X can thus be written as follows:

$$C = (S_{Row_s \times Col_s}, F_{Row_f \times Col_f}, B_{Row_b \times Col_b})$$

For example, suppose  $S = \{S_1\}$ ;  $T_{S_1} = 5$  ( $S_1$  represents the component name of structural property of the software component),  $F = \{F_1, F_2, F_3\}$ ;  $T_{F_1} = 5$ ;  $T_{F_2} = 10$ ;  $T_{F_3} = 10$  ( $F_1, F_2, F_3$  represent the functional name, input, and output in functional property of the software component),  $B = \{B_1, B_2\}$ ;  $T_{B_1} = 5$ ;  $T_{B_2} = 10$  ( $B_1, B_2$  represent the behavioral name and action in behavioral property of the software component).  $C$  is further assumed to be made up of 3 functions and 4 behaviors, that is,

$$C_{S_1} = \{S_{1,1}\}$$

is the component structure name  $S_1$  of  $C$  that contains  $S_{1,1}$ . The function name  $F_1$ , input  $F_2$ , and output  $F_3$  of  $C$  that represent the functional properties are positionally arranged in matrix form. Thus, function 1 becomes

$$C_{F_1} = \{F_{1,1}\}$$

$$C_{F_2} = \{F_{2,1}, F_{2,2}, F_{2,3}\}$$



$$C_{F_3} = \{F_{3,2}, F_{3,2}\}$$

Similarly, function 2 becomes

$$C_{F_1} = \{F_{1,2}\}$$

$$C_{F_2} = \{F_{2,3}, F_{2,10}\}$$

$$C_{F_3} = \{F_{3,2}, F_{3,3}\}$$

and function 3 becomes

$$C_{F_1} = \{F_{1,5}\}$$

$$C_{F_2} = \{F_{2,1}, F_{2,7}\}$$

$$C_{F_3} = \{F_{3,9}, F_{3,10}\}$$

By the same token, the behavioral properties of behavior 1 are denoted by

$$C_{B_1} = \{B_{1,1}\}$$

$$C_{B_2} = \{B_{2,2}, B_{2,3}, B_{2,5}\}$$

Similarly, behavior 2 becomes

$$C_{B_1} = \{B_{1,3}\}$$

$$C_{B_2} = \{B_{2,3}, B_{2,3}, B_{2,6}\}$$

and behavior 3 becomes

$$C_{B_1} = \{B_{1,4}\}$$

$$C_{B_2} = \{B_{2,5}, B_{2,5}, B_{2,8}, B_{2,9}\}$$

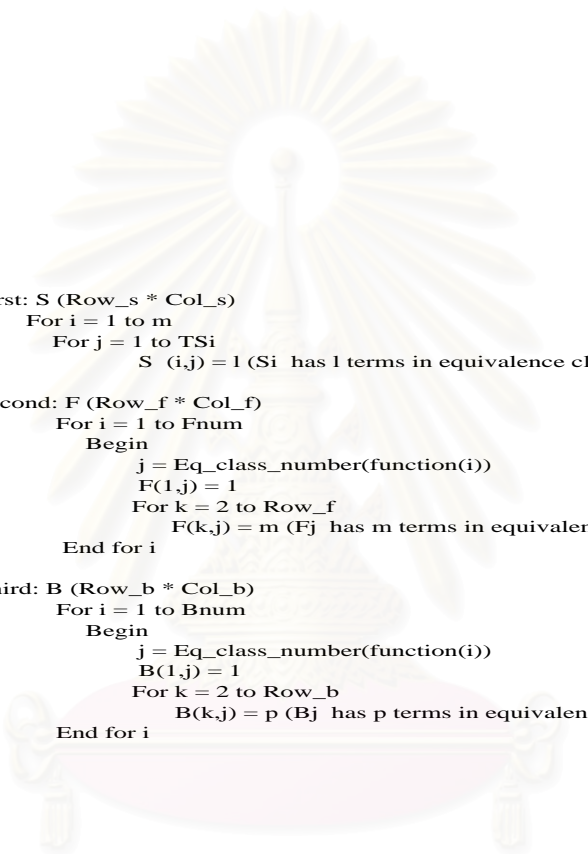
and behavior 4 becomes

$$C_{B_1} = \{B_{1,5}\}$$

$$C_{B_2} = \{B_{2,2}, B_{2,3}, B_{2,7}, B_{2,10}\}$$

The software component matrix is formed by concatenating individual  $i^{th}$  component property vertically. Thus,  $S = \{C_{S1,n}\}$  where  $n$  denotes the column,  $F = \{C_{F1,n}$  concat  $C_{F2,n}$  concat  $C_{F3,n}\}$  and  $B = \{C_{B1,n}$  concat  $C_{B2,n}$  concat  $C_{B3,n}$  concat  $C_{B4,n}\}$ . Any missing property columns are denoted by zero. The resulting matrices S, F, and B are depicted in Figure 4.1.





```

First: S (Row_s * Col_s)
  For i = 1 to m
    For j = 1 to TSi
      S (i,j) = 1 (Si has 1 terms in equivalence 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 (Fj has m terms in equivalence class k)
      End for i

Third: B (Row_b * Col_b)
  For i = 1 to Bnum
    Begin
      j = Eq_class_number(function(i))
      B(1,j) = 1
      For k = 2 to Row_b
        B(k,j) = p (Bj has p terms in equivalence class k)
      End for i

```

Figure 4.2: Matrix Calculation Algorithm.

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## CHAPTER V

### Software Component Classification Model

The preceding standard software component notations identified and represented are employed in this chapter as a basis for component classification. The first step is to examine how software component is reused in order to establish a classification framework over applicable component domains. A formal classification approach will then be presented, along with a simple example to demonstrate the applicability of the proposed framework.

#### 5.1 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 principles of the proposed classification scheme rely 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. The proposed approach experiments with three neural network techniques in an effort to classify the software components as accurate as it can be, namely, FSC, SOM, and RPCL. Preliminary finding concluded that RPCL was the preferred approach. In so doing, a set of clusters and their corresponding centers are obtained, whereby an indexing structure holding cluster centers is set up to enhance the search and retrieve operations. Software components are denoted in matrix form as

discussed in Chapter 4. Evaluation process determines a similarity value of the designed component and the components stored in the repository using rival penalized competitive learning (RPCL) clustering algorithm. However, searching for similar components can be time-consuming as the repository grows. An indexing structure is constructed not only to speed up the process, but also to classify component indexing structure. All software components belonging to the cluster partition whose center is closest to the designated software component will be retrieved for subsequent selection process, i.e., the fine grain level that supports certification methods for the most suitable software component. This process is depicted in Figure 5.1.

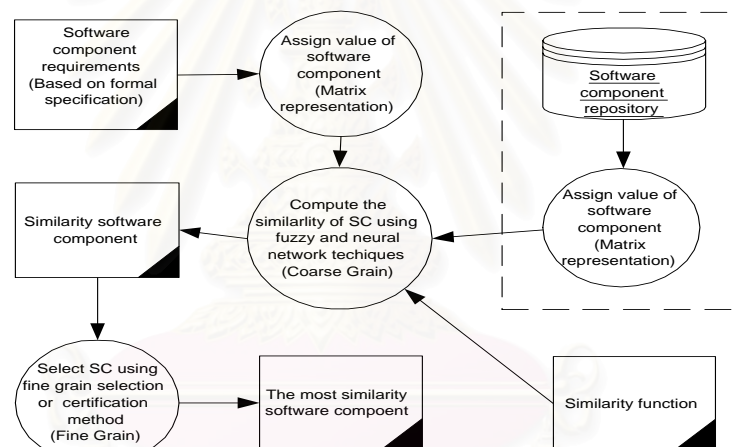


Figure 5.1: Software Reuse Model.

The following sections will describe how the clusters are established using conventional statistical approach in comparison with the neural network techniques for better performance.

## 5.2 Cluster Center Establishment

Based on the above Software Reuse Model, components that possess similar predefined attributes or characteristics are grouped together to form a cluster. Each cluster has a center which denotes the designated properties of that cluster. The process of grouping software components to forming similar cluster calls for intensive computations required by the underlying clustering approach.

Each computational approach is carried out with respect to its training and testing strategy, along with experimental results. This study ran MATLAB 5.3 software on Toshiba Genuine Intel Pentium III Processor with 312 MB RAM.

### 5.2.1 Conventional Statistical Approach

A distribution-free approach known as the Minimum (Mean) Distance Classifier (MDC) is used as a baseline measure of the experiment with which subsequent classification approaches can be compared. The technique is described below.

Let  $x$  be a prototype (or mean) vector of a training class:

$$m_j = \frac{1}{N_j} \sum_{x \in w_j} x \quad \text{for } j = 1, 2, \dots, M \quad (5.1)$$

where  $N_j$  is the number of training pattern vectors of class  $j$ ,  $M$  is the number of classes, and  $w_j$  is the all vectors in individual class. An arbitrary pattern vector  $x$  in the class whose prototype vector is the closest in term of the Euclidean distance is given by

$$D_j(x) = \|x - m_j\| \quad \text{for } j = 1, 2, \dots, M \quad (5.2)$$

This is equivalent to computing and assigning  $x$  to class  $w_j$  if  $d_j(x)$  yields the largest value, i.e.,

$$d_j(x) = m_j x^T - \frac{1}{2}(m_j m_j^T) \quad \text{for } j = 1, 2, \dots, M \quad (5.3)$$

Finally, the decision boundary, which separates class  $w_i$  from  $w_j$  can be obtained from

$$d_i(x) - d_j(x) = 0 \quad (5.4)$$

In general, MDC works well when there are sufficient data to enhance the weight spread (or randomness) converging toward the mean, i.e., the cluster center of each class.

### 5.2.2 Self-Organizing Map

The Self-Organizing Map (SOM), as proposed in [23], [24] and described thoroughly in [25], is one of the most well-known unsupervised artificial neural network models. This model consists of a layer of input units, each of which is fully connected to a grid of output units. These output units are arranged in some topology represented by a two-dimensional grid. The network is first initialized, followed by three essential processes involving the formation of the self-organizing map as summarized below:

1. *Competition.* For each input pattern, the neurons in the network compute their respective values of a discriminant function. This discriminant function provides a basis for competition among the neurons. The particular neuron with the largest value of discriminant function is declared the winner of the competition.

2. *Cooperation.* The winning neuron determines the spatial location of a topological neighborhood of excited neurons, thereby providing the basis for cooperation among neighboring neurons.

3. *Synaptic/Adaptation.* This last mechanism enables the excited neurons to increase their individual value of the discriminant function in relation to the input pattern through suitable adjustments applied to their synaptic weights. The adjustments made are such that the response of the winning neuron to the subsequent application of a similar input pattern is enhanced and subsequently adapted to be a number of the cluster. This process is decomposed into two phases, namely, an ordering or self-organizing phase and convergence phase as follows:

3.1. Self-organizing or ordering phase. This phase may take as many as 1000 iterations or higher. Careful considerations must be given to the choice of the learning rate parameter and neighborhood function.

3.2. Convergence phase. This phase is needed to fine tune the feature map and therefore provides an accurate statistical quantification of the input space. As a general rule, the number of iterations constituting the convergence phase must be at least 500 times the number of neurons in the network. Thus, the convergence phase may have to go on for thousands and possibly tens of thousands of iterations.

Given the above adaptation process, it is apparent that both phases constitute a combinational explosion of iterations. As a consequence, this approach is subsequently the least efficient.

The following steps summarize the overall learning algorithm [17]:

1. Choose small random values for the initial weight vectors that serve as the candidate center of cluster, that is, for neuron  $j$ , the random value is  $w_j(0)$ . The only restriction here is that the  $w_j(0)$  be different for  $j = 1, 2, \dots, l$ , where  $l$  is the number of neurons in the lattice.



2. Draw a pattern  $x$  from the input space with a certain probability [17]
3. Find the winning neuron  $i(x)$  at the  $n^{\text{th}}$  iteration based on minimum Euclidean criterion

$$i(x) = \arg \min_j \|x(n) - w_j\|, \quad \text{for } j = 1, 2, \dots, l \quad (5.5)$$

4. Adjust the weight vectors of all neurons according to

$$w_j(n+1) = w_j(n) + \eta(n)h_{j,i(x)}(n)(x(n) - w_j(n)) \quad (5.6)$$

where  $\eta(n)$  is the learning rate, and  $h_{j,i(x)}(n)$  is the neighborhood function centered at  $i(x)$ . Here,  $\eta(n)$  and  $h_{j,i(x)}(n)$  vary with time during learning as indicated.

5. Repeat step 2 until no noticeable changes in the feature map are observed. In so doing, the weighted vectors move toward the input vectors and often tend to follow the distribution of the input vectors.

### 5.2.3 Fuzzy Subtractive Clustering

Fuzzy Subtractive Clustering (FSC) [2], [3], [10], [13], [22] does not require a predefined number of clusters. Each data point is regarded as a potential cluster center. A measure of centering potential of each data point is determined from the density of the surrounding data points.

The FSC approach sets up the cluster centers using a parameter  $r_a$  which is the maximum distance between any two points in the same cluster, yet less than the distance between any two points from different clusters. 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 is the 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, obtained from the fraction of the potential first cluster center below which a data point will be rejected as a cluster center. The default value from MATLAB version 5.3 of 0.5 was chosen as the accepted ratio for the first cluster center. The rejection ratio ( $\eta$ ) was set 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  $\{X_1, X_2, X_3, \dots, X_{n=50}\}$  in the training set is described below.

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

$$P_i = \sum_{j=1}^n e^{-\alpha \|x_i - x_j\|^2} \quad (5.7)$$

where  $\alpha = \frac{4}{r_a^2}$

$\|\cdot\|$  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 on 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  $P^{(1)*}$

$$P^{(1)*} = \max_i (P^{(1)}(x_i)) \quad (5.8)$$

3. Define a threshold  $\delta$  to be the decision to continue or stop the cluster center search. This process will continue if the current maximum potential remains greater

than  $\delta$ .

$$\delta = (\text{rejection ratio}) \times (\text{potential value of the first cluster center})$$

where the rejection ratio ( $\eta$ ) 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 each remaining data point according to the equation

$$P_i = P_i - P_k^* e^{-\beta \|x_i - x_k^*\|^2} \quad (5.9)$$

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

6. For the point having the maximum potential value, calculate the acceptance value. If this value is greater than the predefined constant (0.5), the point is accepted to be the next cluster center. Otherwise, if the acceptance value is greater than the predefined threshold ( $\eta = 0.15-0.5$ ), and it achieves a good balance between having a reasonable potential and being far from all existing cluster centers, this data point can be accepted as the next cluster center.

This procedure is repeated to generate each cluster center until the maximum potential value in the current iteration is equal to or less than the threshold  $\delta$ .

### 5.2.4 Rival Penalized Competitive Learning

Rival Penalized Competitive Learning (RPCL) [9], [42], [43] is a new version of Competitive Learning by adding some mechanisms to Frequency Sensitive Competitive Learning (FSCL) method [42], [43]. The main idea is to accept the input weight vector  $w_c$  (which

subsequently becomes the winner) and to reject (or de-learned) the weight vector  $w_r$  of the rival vector by a given learning rate. The RPCL process reduces the total learning cost to a global minimum as a result of the convergence of all weight vectors to the winner in the cluster.

The RPCL approach establishes the cluster centers with learning ( $\alpha_c$ ) and de-learning ( $\alpha_r$ ) rates in the range  $0 \leq \alpha_c, \alpha_r \leq 1$  [42], where  $\alpha_r \ll \alpha_c$  for each step. In this experiment, the values  $\alpha_c = 0.5$  and  $\alpha_r = 0.004$  are chosen from a priori information. The number of initial centers is set to greater than or equal to the number of equivalent classes established earlier, in this case, 10, 20, 30, 40, and 50.

The procedures for grouping the training data proceed as follows:

Given  $D = \{d(k), x(k)\}$ , where  $d(k) = [d_1(k), \dots, d_{10}(k)]$  is the desired output set,  $x(k) = [x_1(k), \dots, x_{1320}(k)]$  denote the input vector,  $p = 10$  equivalent classes, and  $k = 1, \dots, 50$ .

Step 1: Initialization:

Randomly select  $n$  seeded centers  $c_j$ ;  $j = 1, \dots, n$  in the observation space containing the data set  $D$ . The value of  $n$  is determined either from a priori information on  $p$  or by simply setting  $n$  large enough. In this case,  $n = 10$ . Initialize the  $m_j = 1$ ,  $j = 1, \dots, n$  and approximate  $\gamma_j$  by

$$\gamma_j = m_j / \sum_i m_i \quad (5.10)$$

Step 2: Adaptive learning:

Sequentially take each  $x(k)$  from the data set  $D$ , determine the winner seed point  $C_{c(k)}$  and a rival seed point  $C_{r(k)}$  in relation to the proximity of each  $x(k)$  from  $C_{c(k)}$  and  $C_{r(k)}$ , that is,

$$c(k) = \arg \min_{1 \leq j \leq n} E_j(k) \quad (5.11)$$

$$r(k) = \arg \min_{j \neq c, 1 \leq j \leq n} E_j(k) \quad (5.12)$$

$$E_j(k) = \gamma_j \|x(k) - c_j\|^2 \quad (5.13)$$

The winner is moved closer to  $x(k)$ , whilst the rival is moved away from  $x(k)$ . That is,

$$c_j^{new} = c_j^{old} + \Delta c_j(k) \quad (5.14)$$

with

$$\Delta c_j = \begin{cases} \alpha_{c(k)}(x(k) - c_j), & \text{if } j = c(k) \\ -\alpha_{r(k)}(x(k) - c_j), & \text{if } j = r(k) \\ 0, & \text{otherwise} \end{cases} \quad (5.15)$$

The count of winner  $m_{c(k)}$  is incremented by 1 and  $\gamma_j$  is updated according to Eq.(6). Step 2 is repeated until the winner of seed points is unchanged for all  $x(k)$ . The winning seed point converges to a position surrounded by data points. This position is taken to be a preliminary cluster center. The points that diverge away from the center are excessive and consequently discarded.

The experiment was repeated with different initial centers, namely, 20, 30, 40, and 50 to study variation effects on cluster centers of the classification process. The results were analyzed and evaluated accordingly.

## CHAPTER VI

### Experiment

#### 6.1 Data Collection

One hundred software components were gathered according to formal component specification [30]. These component data were separated into their respective equivalent classes based on the three properties whose ranges were defined by a priori class properties. The data were arranged in matrix form as described in Chapter 4, where each element of the matrix denoted a specific property falling in the above range. For instance, in Figure 4.1, one software component data  $C_{F_3} = \{F_{3,2}, F_{3,3}\}$  of function 2 rendered the second column of the 2<sup>nd</sup> and the 3<sup>rd</sup> rows of the third partition in matrix  $F$  to be  $[ 2 \ 1 \ 0 \ 0 \ 0 ]$  and  $[ 0 \ 1 \ 0 \ 0 \ 0 ]$ , respectively. As such, each component data vector encompassed 1320 dimensions.

The data set was divided into two groups, namely, 50 training data and 50 test data to be used by our proposed algorithm. Each data point was normalized according to the following criteria:

$$v_{new} = \frac{v_{old} - v_{min}}{v_{max} - v_{min}} \quad (6.1)$$

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.

## 6.2 Method Learning and Clustering

Two measures of software component retrieval performance used in this study 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 recall. The goal is to find a practical balance between the two. The relevant conditions are 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 utilized by an adaptation system when considering software component compositions.

Given a set of predefined clusters  $C = c_1^n$  and the calculated RPCL clusters,  $C' = c_1^m$ , the performance measures of RPCL is defined as follows [15]:

*Recall = Number of target software component retrieved/Number of target software component*

$$= \sum_{c_j \in C \wedge c_j \in C'} \frac{c_i \cap c_j}{\#c_i} \quad (6.2)$$

*Precision = Number of target software component retrieved/Number of software component retrieved*

$$= \sum_{c_j \in C \wedge c'_j \in C'} \frac{c_i \cap c'_j}{\#c'_j} \quad (6.3)$$

where  $\#c_i$  denoted the number of elements on the cluster  $c_i$  and  $0 \leq recall$ ,  $precision \leq 1$ . Recall shows the ratio of the target repository objects are actually retrieved out of all the expected target repository objects, whereas precision indicates the ratio of target repository objects in 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, recall is 0.75 and precision is 0.6. The higher the recall and precision, the more accurate the method for retrieval. The accuracy for each RPCL clusters can be calculated based on the information pertaining to their natural clusters. The response time of the system is measured to determine the practicality of the method. For each measured quality, the minimum and median are computed from every scenario in the experiment, which will be discussed in the next section.

### 6.3 Evaluation

From the one hundred vector data participated in the experiment, the experiment regulated different classification approaches, namely, MDC, SOM, FSC, and RPCL to determine the recall, precision, training time, and the amount of derived cluster.

Table 6.1, 6.2, 6.3, and 6.4 show the precision, recall, average computation time, and derived cluster number results obtained from the MDC, SOM, FSC, and RPCL approaches, respectively. Figure 6.1 illustrates the comparative magnitude of the individual results so obtained.

Some comparative inferences as shown in Table 6.5 can be drawn from the above experimental outcomes as follows:



Table 6.1: Comparison of precision performance

Data Type	MDC	SOM	FSC	RPCL
Well-separated clusters	100.00%	97.02%	100.00%	100.00%
1 <sup>st</sup> set Overlapping clusters	87.50%	93.33%	91.67%	96.67%
2 <sup>nd</sup> set Overlapping clusters	92.50%	98.21%	96.38%	100.00%
3 <sup>rd</sup> set Overlapping clusters	92.50%	92.86%	95.00%	96.15%
4 <sup>th</sup> set Overlapping clusters	95.00%	94.64%	97.50%	98.21%
Overall	93.50%	95.21%	96.11%	98.21%

Table 6.2: Comparison of recall performance

Data Type	MDC	SOM	FSC	RPCL
Well-separated clusters	100.00%	68.57%	100.00%	100.00%
1 <sup>st</sup> set Overlapping clusters	88.00%	65.33%	45.00%	64.00%
2 <sup>nd</sup> set Overlapping clusters	92.00%	70.00%	41.74%	66.67%
3 <sup>rd</sup> set Overlapping clusters	92.00%	65.71%	47.00%	73.85%
4 <sup>th</sup> set Overlapping clusters	94.20%	67.14%	48.00%	70.00%
Overall	93.20%	67.35%	56.35%	74.90%

Table 6.3: Comparison of average computation time

Data Type	MDC	SOM	FSC	RPCL
Well-separated clusters	0.06	38008	5.20	219.37
1 <sup>st</sup> set Overlapping clusters	0.06	37890	5.27	270.45
2 <sup>nd</sup> set Overlapping clusters	0.05	36797	5.55	274.69
3 <sup>rd</sup> set Overlapping clusters	0.06	37890	5.22	262.49
4 <sup>th</sup> set Overlapping clusters	0.06	38182	5.22	255.68
Overall	$0 < T_{MDC} \leq 1$	$T_{SOM} > 10000$	$1 < T_{FSC} \leq 10$	$100 < T_{RPCL} \leq 300$

Table 6.4: Comparison of derived cluster number

Data Type	MDC	SOM	FSC	RPCL
Well-separated clusters	10	14	10	10
1 <sup>st</sup> set Overlapping clusters	10	15	20	15
2 <sup>nd</sup> set Overlapping clusters	10	14	23	15
3 <sup>rd</sup> set Overlapping clusters	10	14	20	13
4 <sup>th</sup> set Overlapping clusters	10	14	20	14
Overall	$NC_{MDC} = 10$	$14 \leq NC_{FSC} \leq 15$	$10 < NC_{FSC} \leq 23$	$10 < TR_{PCL} \leq 15$

Table 6.5: Comparison of the four methods of clustering performance

Data Type	MDC	SOM	FSC	RPCL
Precision	Lowest	Moderate	High	Highest
Recall	Highest	Moderate	Lowest	High
Preprocessing Speed	Fastest	Very slow	Fast	Moderate
Number of Clusters	Fewest	Moderate	High	Moderate

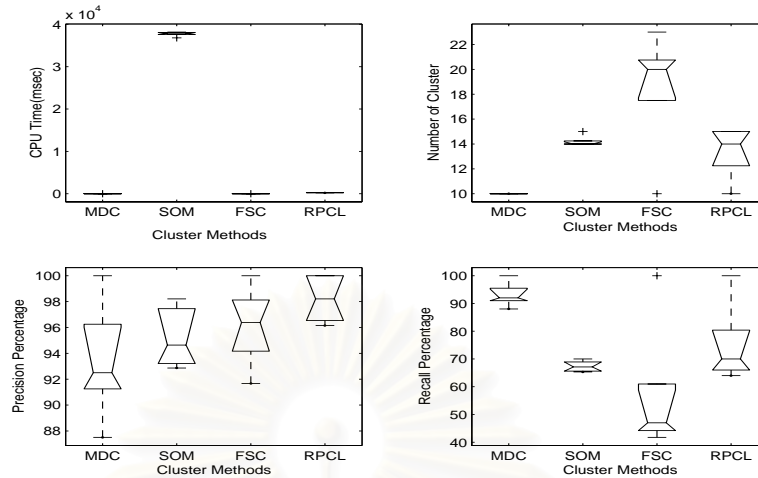


Figure 6.1: Box Plot of precision, recall, average computation time, and number of derived clusters.

1. Although MDC possesses the highest recall and speed, its precision is the lowest. The deficiency of this method is because it is a supervised classification. The number of equivalent classes must be known in advance.
2. The recall and precision of SOM are lower than RPCL. The time performance is also the poorest among all approaches. As such, the method is not considered to be a viable candidate.
3. Despite the second best time performance and acceptable precision result, FSC yields too many centers to consolidate the dispersion of classifying clusters.

An entirely unsupervised experiment (without MDC) was conducted with fixed number of cluster centers revealed the same outcomes. The results are shown in Table 6.6, 6.7, 6.8, and 6.9. It is apparent from Table 6.9 and Figure 6.2 that FSC method performs poorly as the number of cluster centers decreases (fixed in this case) since the method builds the clusters by utilizing the data points as cluster centers as oppose to SOM and RPCL methods that use the initial weights to be the cluster centers. Hence, some data points will be misclassified since they are forced into a cluster where they don't belong.

Table 6.6: Unsupervised Methods Comparison of precision performance

Data Type	SOM	FSC	RPCL
Well-separated clusters	97.02%	100.00%	100.00%
1 <sup>st</sup> set Overlapping clusters	93.33%	88.65%	97.22%
2 <sup>nd</sup> set Overlapping clusters	98.21%	83.67%	100.00%
3 <sup>rd</sup> set Overlapping clusters	92.86%	77.72%	97.02%
4 <sup>th</sup> set Overlapping clusters	94.64%	77.72%	98.21%
Overall	95.21%	85.55%	98.49%

Table 6.7: Unsupervised Methods Comparison of recall performance

Data Type	SOM	FSC	RPCL
Well-separated clusters	68.57%	71.43%	71.42%
1 <sup>st</sup> set Overlapping clusters	65.33%	57.33%	64.00%
2 <sup>nd</sup> set Overlapping clusters	70.00%	57.14%	71.73%
3 <sup>rd</sup> set Overlapping clusters	65.71%	54.29%	68.57%
4 <sup>th</sup> set Overlapping clusters	65.71%	54.29%	68.57%
Overall	67.35%	58.90%	69.14%

Table 6.8: Unsupervised Methods Comparison of average computation time

Data Type	SOM	FSC	RPCL
Well-separated clusters	38768.16	5.30	223.76
1 <sup>st</sup> set Overlapping clusters	38647.80	5.38	275.86
2 <sup>nd</sup> set Overlapping clusters	37532.94	5.66	280.18
3 <sup>rd</sup> set Overlapping clusters	38647.80	5.32	267.74
4 <sup>th</sup> set Overlapping clusters	38945.64	5.32	260.79
Overall	$T_{SOM} > 10000$	$1 < T_{FSC} \leq 10$	$100 < T_{RPCL} \leq 300$

Table 6.9: Comparison of the three methods of clustering performance

Data Type	SOM	FSC	RPCL
Precision	Moderate	lowest	Highest
Recall	Moderate	Lowest	Highest
Preprocessing Speed	Very slow	Fastest	Moderate

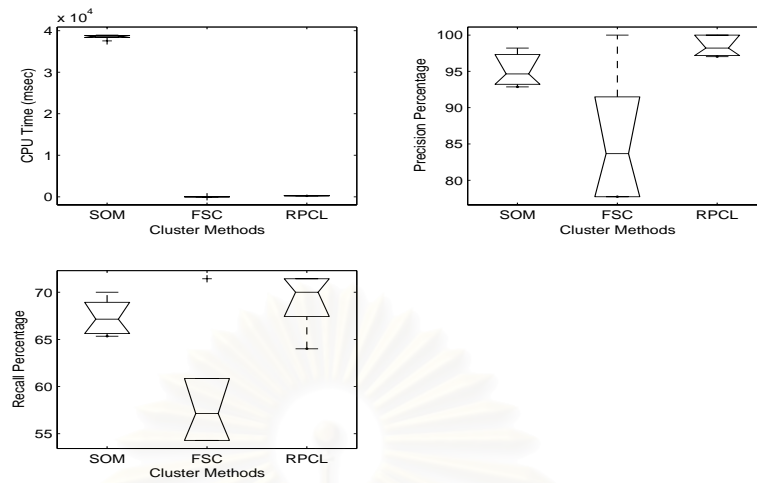


Figure 6.2: Box Plot of precision, recall, and average computation time.

Bearing the aforementioned shortcomings of all methods, RPCL exhibits near optimal performance measures in all categories summarized below.

1. RPCL is a fast clustering methods and moderately over SOM.
2. RPCL offers a good clustering approximation owing to its straightforward cluster center computations.
3. The actual number of clusters can usually be determined from the input data set.
4. RPCL consumes low storage utilization because only the information pertaining to cluster centers is maintained [15].
5. RPCL employs tree-indexing approach which lends itself to myriad of tree-related analyses, algorithms, tools, and implementations.

## CHAPTER VII

### Component Classification and Retrieval

#### 7.1 Non-hierarchical Indexing Classification of Software Components

Software component classification process divides software components into groups using rival penalized learning clustering. A flat indexing structure accompanying these clusters is constructed as shown in Figure 7.1, which serves as a retrieval mechanism supporting software component repository. This classification process is called non-hierarchical indexing.

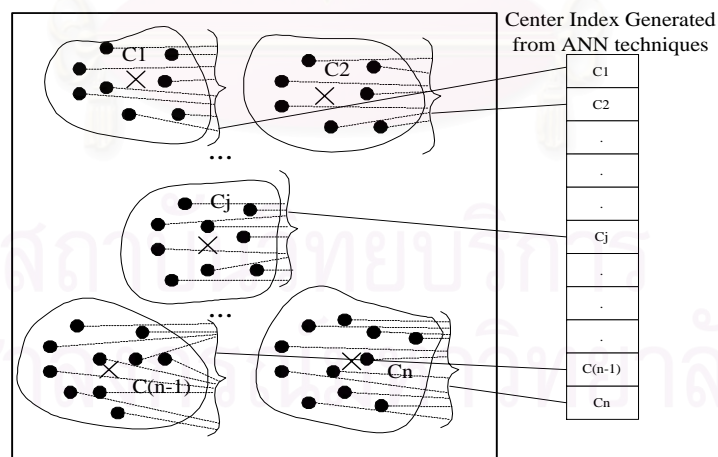


Figure 7.1: The  $n$  cluster partitions generated by ANN technique. The dots are the repository software component whereas the crosses are the centers. The center index is used for indexing

## 7.2 Hierarchical Indexing Classification of Software Components

There are two principal content-based indexing methods, namely, rectangle-based indexing and partition-based indexing. Various content-based indexing methods have been successfully applied in some cases. The short-falls of content-retrieval are caused by components whose contents (properties) lie on the partition boundary of the adjacent clusters. The querying algorithm merely searches for matching contents, thus failing to take property distribution into account. This property distribution, in some cases, may be incorrectly partitioned into different clusters, whereby yielding multiple clusters of different characteristics that encompass the same components with natural property. This phenomenon is known as the boundary search problem. As a consequent, the retrieval precision will decrease when a query falls near the boundary of a partition in the indexing structure, which is derived directly from the systematic yet unfavorable overlapping clusters.

For example, the rectangle-based indexing [21], [27] such as R-tree, R+-Tree, and R\*-tree are built on the input sequence of the data objects so that they are not affected by the distribution of the input data and the calculated natural cluster. The partition-based indexing method such as MDC and VP-tree partition the data object space according to the median distance between the data objects and the cluster points. Such approaches still cannot exactly determine the natural cluster of the component for retrieval. As a consequence, the performance of nearest-neighbor retrieval for these methods suffers from the boundary problem.

To solve the boundary problem for classification and retrieval, various unsupervised ANN techniques are employed to group the clusters and derive the corresponding cluster index structure, namely, RPCL, SOM, and FSC. The experiment described in Chapter



6 was carried out to compare each technique in terms of their precision, recall, time performance, and derived clusters.

The above ANN techniques utilize non-hierarchical approach to construct a property vector space that is derived from constituent components. Straightforward as the process may sound, the approach poses some inherent limitations [21], [27]. First, the flat non-hierarchical structure lacks the relationship between nodes that exist if created in hierarchical fashion. As such, component classification and retrieval follows a non-hierarchical indexing which renders inefficient insertion and deletion operations. The target node must be explicitly located for the ensuing deletion or insertion. Second, the non-hierarchical structure still possesses some overlapping classification-mix, which results in the perpetual boundary problem in nearest-neighbor query search.

In order to lessen the above problems, the RPCL technique is employed to build a hierarchical content-based indexing structure to facilitate subsequent classification and retrieval of software components in the repository. The index structure is based on nearest-neighbor search result. In nearest-neighbor search, a group of properties are retrieved as the result of a query. The proximity of each member of the group from the cluster's center is employed as the basis to arrange the indexing placement. As such, the relationship between nodes in different levels can be embedded into the index structure. The hierarchical approach recursively transforms a property vector space into a sequence of clusters. In so doing, all property vectors are progressively organized into nested clustered as shown in Figure 7.2. This configuration not only lends itself to easy index structure update, but also supports backtracking and elimination rules required by the branch-and-bound algorithm. Consequently, 100% nearest-neighbor results can be obtained.

The approach is formulated as follows [21], [27]. Let the property vector set  $X$  having  $n$  components be denoted by

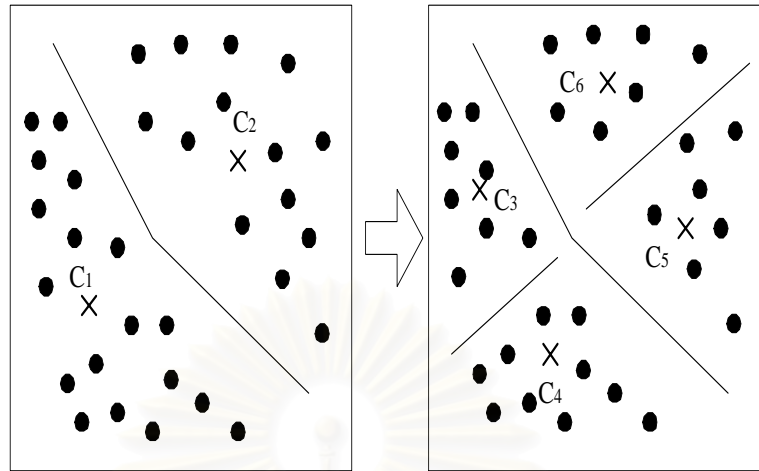


Figure 7.2:  $C_3$  and  $C_4$  are the clusters inside  $C_1$  while  $C_5$  and  $C_6$  are clusters inside  $C_2$ . The black dots represent repository software components (property vector), the crosses represent the center of each cluster

$$X = \{X_i\}_{i=1}^n \quad (7.1)$$

A cluster  $C$  of  $X$  breaks  $X$  into subsets  $C_1, C_2, \dots, C_m$  satisfying the following conditions

$$C_i \cap C_j = \phi, 1 \leq i, j \leq m, i \neq j, \text{ and } C_1 \cup C_2 \cup \dots \cup C_m = X$$

Cluster  $B$  is nested inside cluster  $A$  if every component of  $B$  is a proper subset of  $A$ . The dichotomy of cluster  $C$  entails a derivation of a binary mapping function that ties all the constituent components into the indexing structure so constructed. This is illustrated in Figure 7.3. From the root level  $C_0$ , there are  $2^i$  subsets (clusters) at depth  $i$  to be directly located by the index. At the top level, a nearest-neighbor query  $q$  (for a designated software component) is compared to the centers of the clusters of the immediate lower level. The cluster whose center is the closest to the query point  $q$  is selected. The elements in the selected cluster will be the result of the query if they

satisfy the criteria of the nearest-neighbor search. Otherwise, the search will proceed to the next lower levels. Further details of this method can be found in [21], [27].

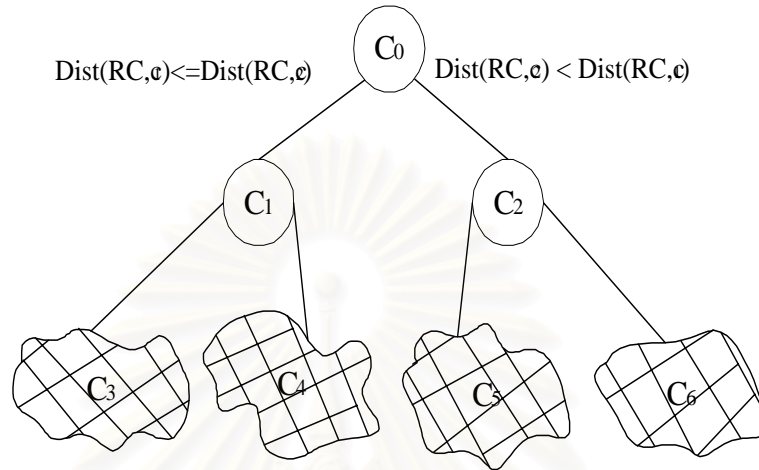


Figure 7.3: Indexing structure for hierarchical clustering.  $C_0$  is the root node which contains all software components in repository.  $\text{Dist}(RC, C_i)$  is the Euclidean Norm Distance between the nearest-neighbor designated component  $RC$  and the center  $C_i$  of cluster  $C_i$

### 7.3 Software Component Selection Technique (Fine Grain Level)

This level locates the most suitable software component for reuse. The degree of significance defined by the user will be used as the selection criteria. The following notations are used:

- $\phi_S$ ,  $\phi_F$ , and  $\phi_B$  are the degree of significance of structural, functional, and behavioral properties, respectively, satisfying  $0 \leq \phi_S, \phi_F, \phi_B \leq 1$  and  $\phi_S + \phi_F + \phi_B = 1$ . 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 Chapter 4, 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:

$$SC = X_{reuse} \quad (7.2)$$

where the value of reuse can be computed from

$$reuse = \arg \min_{1 \leq i \leq N_r} \left( \sum_{p=S,F,B} \phi_p \|Xp_r - Xp_i\| \right) \quad (7.3)$$

## 7.4 Application of Neural Network

In performance comparison depicted in Figure 6.1, the proposed methods, namely, FSC and RPCL, yield satisfactory performance, while SOM [33], [8] is the most time-consuming method. The supervised MDC method that is used as the benchmark method gives the lowest correction. The following section will compare the results of experiment with FSC and RPCL which was conducted on well-separated data clusters.

### 7.4.1 Coarse Grain RPCL Selection

From 50 training data sets of 10 equivalent classes, five trials with 10, 20, 30, 40, and 50 initial centers were utilized to derive coarse grain clusters based on RPCL algorithm. Each set consists of one hundred 1320-dimension random feature vectors. The other 50

Table 7.1: RPCL recall and precision performance of 10 random initialized centers.

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
4	SC1,SC2,SC3,SC4,SC5	SC1,SC2,SC3,SC4,SC5	1.00	1.00
1	SC6,SC7,SC8,SC9,SC10	SC6,SC7,SC8,SC9,SC10	1.00	1.00
3	SC11,SC12,SC13,SC14,SC15	SC11,SC12,SC13,SC14,SC15	1.00	1.00
8	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
9	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
10	SC26,SC27,SC28,SC29,SC30	SC26,SC27,SC28,SC29,SC30	1.00	1.00
6	SC31,SC32,SC33,SC34,SC35	SC31,SC32,SC33,SC34,SC35	1.00	1.00
7	SC36,SC37,SC38,SC39,SC40	SC36,SC37,SC38,SC39,SC40	1.00	1.00
5	SC41,SC42,SC43,SC44,SC45	SC41,SC42,SC43,SC44,SC45	1.00	1.00
2	SC6,SC47,SC48,SC49,SC50	SC46,SC47,SC48,SC49,SC50	1.00	1.00
Average			1.00	1.00

test data sets of the same size were then fed into the PRCL network, whereby recall and precision performance measure were taken.

The results can be described as follows. The initial 10, 20, 30, 40, and 50 random centers became 10, 10, 10, 19, and 23, respectively. Table 7.1, 7.2, 7.3, 7.4, and 7.5 show the recall and precision measure so obtained. Table 7.6 shows the comparison of recall and precision performance of all 5 clusters. Note that RPCL algorithm yields a 100% precision with no fault classification, hence closely-related software component clusters. Moreover, similar software components can be retrieved in less attempts with fewer number of centers from the indexing structure (the recall percentage). A slight drop in the recall measure merely implies more centers in each equivalent class (suggesting further subdivision may be recommended).

Table 7.2: RPCL recall and precision performance of 20 random initialized centers.

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
9	SC1,SC2,SC3,SC4,SC5	SC1,SC2,SC3,SC4,SC5	1.00	1.00
11	SC6,SC7,SC8,SC9,SC10	SC6,SC7,SC8,SC9,SC10	1.00	1.00
7	SC11,SC12,SC13,SC14,SC15	SC11,SC12,SC13,SC14,SC15	1.00	1.00
13	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
18	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
15	SC26,SC27,SC28,SC29,SC30	SC26,SC27,SC28,SC29,SC30	1.00	1.00
5	SC31,SC32,SC33,SC34,SC35	SC31,SC32,SC33,SC34,SC35	1.00	1.00
3	SC36,SC37,SC38,SC39,SC40	SC36,SC37,SC38,SC39,SC40	1.00	1.00
8	SC41,SC42,SC43,SC44,SC45	SC41,SC42,SC43,SC44,SC45	1.00	1.00
20	SC6,SC47,SC48,SC49,SC50	SC46,SC47,SC48,SC49,SC50	1.00	1.00
Average			1.00	1.00

Table 7.3: RPCL recall and precision performance of 30 random initialized centers.

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
12	SC1,SC2,SC3,SC4,SC5	SC1,SC2,SC3,SC4,SC5	1.00	1.00
14	SC6,SC7,SC8,SC9,SC10	SC6,SC7,SC8,SC9,SC10	1.00	1.00
5	SC11,SC12,SC13,SC14,SC15	SC11,SC12,SC13,SC14,SC15	1.00	1.00
14	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
2	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
27	SC26,SC27,SC28,SC29,SC30	SC26,SC27,SC28,SC29,SC30	1.00	1.00
28	SC31,SC32,SC33,SC34,SC35	SC31,SC32,SC33,SC34,SC35	1.00	1.00
8	SC36,SC37,SC38,SC39,SC40	SC36,SC37,SC38,SC39,SC40	1.00	1.00
18	SC41,SC42,SC43,SC44,SC45	SC41,SC42,SC43,SC44,SC45	1.00	1.00
29	SC6,SC47,SC48,SC49,SC50	SC46,SC47,SC48,SC49,SC50	1.00	1.00
Average			1.00	1.00

Table 7.4: RPCL recall and precision performance of 40 random initialized centers.

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
23	SC1,SC2,SC3,SC4,SC5	SC1,SC2	0.40	1.00
4	SC1,SC2,SC3,SC4,SC5	SC3,SC4,SC5	0.60	1.00
1	SC6,SC7,SC8,SC9,SC10	SC6,SC7	0.40	1.00
18	SC6,SC7,SC8,SC9,SC10	SC8,SC9,SC10	0.60	1.00
3	SC11,SC12,SC13,SC14,SC15	SC11,SC12	0.40	1.00
12	SC11,SC12,SC13,SC14,SC15	SC13,SC14,SC15	0.60	1.00
35	SC16,SC17,SC18,SC19,SC20	SC16,SC17	0.40	1.00
8	SC16,SC17,SC18,SC19,SC20	SC18,SC19,SC20	0.60	1.00
30	SC21,SC22,SC23,SC24,SC25	SC21,SC25	0.40	1.00
28	SC21,SC22,SC23,SC24,SC25	SC22,SC23,SC24	0.60	1.00
16	SC26,SC27,SC28,SC29,SC30	SC26,SC29	0.40	1.00
29	SC26,SC27,SC28,SC29,SC30	SC27,SC28,SC30	0.60	1.00
34	SC31,SC32,SC33,SC34,SC35	SC31,SC32	0.40	1.00
39	SC31,SC32,SC33,SC34,SC35	SC33,SC34,SC35	0.60	1.00
27	SC36,SC37,SC38,SC39,SC40	SC36,SC37,SC38,SC39,SC40	1.00	1.00
26	SC41,SC42,SC43,SC44,SC45	SC41,SC42	0.40	1.00
37	SC41,SC42,SC43,SC44,SC45	SC43,SC44,SC45	0.60	1.00
25	SC6,SC47,SC48,SC49,SC50	SC46,SC47	0.40	1.00
13	SC6,SC47,SC48,SC49,SC50	SC48,SC49,SC50	0.60	1.00
Average			0.53	1.00

Table 7.5: RPCL recall and precision performance of 50 random initialized centers.

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
5	SC1,SC2,SC3,SC4,SC5	SC1,SC2	0.40	1.00
19	SC1,SC2,SC3,SC4,SC5	SC3,SC4	0.40	1.00
1	SC1,SC2,SC3,SC4,SC5	SC5	0.20	1.00
37	SC6,SC7,SC8,SC9,SC10	SC6,SC7	0.40	1.00
22	SC6,SC7,SC8,SC9,SC10	SC8,SC9,SC10	0.60	1.00
21	SC11,SC12,SC13,SC14,SC15	SC11,SC12	0.40	1.00
33	SC11,SC12,SC13,SC14,SC15	SC13,SC14,SC15	0.60	1.00
25	SC16,SC17,SC18,SC19,SC20	SC16,SC17	0.40	1.00
45	SC16,SC17,SC18,SC19,SC20	SC18,SC19,SC20	0.60	1.00
6	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC25	0.60	1.00
46	SC21,SC22,SC23,SC24,SC25	SC23	0.20	1.00
43	SC21,SC22,SC23,SC24,SC25	SC24	0.20	1.00
2	SC26,SC27,SC28,SC29,SC30	SC26,SC29	0.40	1.00
42	SC26,SC27,SC28,SC29,SC30	SC27,SC28	0.40	1.00
32	SC26,SC27,SC28,SC29,SC30	SC30	0.20	1.00
7	SC31,SC32,SC33,SC34,SC35	SC31,SC32	0.40	1.00
18	SC31,SC32,SC33,SC34,SC35	SC33,SC34,SC35	0.60	1.00
36	SC36,SC37,SC38,SC39,SC40	SC36,SC37	0.40	1.00
29	SC36,SC37,SC38,SC39,SC40	SC38,SC39,SC40	0.60	1.00
4	SC41,SC42,SC43,SC44,SC45	SC41,SC42	0.40	1.00
30	SC41,SC42,SC43,SC44,SC45	SC43,SC44,SC45	0.60	1.00
44	SC6,SC47,SC48,SC49,SC50	SC46,SC47	0.40	1.00
10	SC6,SC47,SC48,SC49,SC50	SC48,SC49,SC50	0.60	1.00
Average			0.43	1.00



Table 7.6: RPCL recall and precision performance comparison.

Number of Initialized Center	Number of Centers Select for Indexing Structure	Recall	Precision
10	10	1.00	1.00
20	10	1.00	1.00
30	10	1.00	1.00
40	19	0.53	1.00
50	23	0.43	1.00

### 7.4.2 Coarse Grain FSC Selection

The FSC algorithm was evaluated in a similar manner as that of RPCL. From the 50 training data sets of 10 equivalent class, five trials with different rejection ratio ( $\eta$ ) groups (0.15-0.20, 0.25, 0.30, and 0.35-0.50) were conducted to derive the final cluster centers and the measure their recall and precision performance with the help of 50 testing data sets.

Table 7.7, 7.8, 7.9, and 7.10 show the recall and precision results of the initial 0.15-0.20, 0.25, 0.30, and 0.35-0.50 rejection ratio ( $\eta$ ). Table 7.11 shows the comparison of recall and precision performance of all 4 rejection ratio ( $\eta$ ) groups.

Table 7.11 shows the recall and precision results based on FSC which, in most cases, classifies the software component correctly. The precision performance yields 98-100% depending on the rejection ratio.

### 7.4.3 Fine Grain RPCL Selection

The selection process utilizes cluster indexing structure obtained from Table 7.1 to retrieve the most suitable candidate software component. An essential selection criterion that places the weighted emphasis on structure, function, and behavior of the software

Table 7.7: FSC recall and precision performance of 0.15-0.20 rejection ratio ( $\eta$ ) value

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
SC1	SC1,SC2,SC3,SC4,SC5	SC1	0.20	1.00
SC3	SC1,SC2,SC3,SC4,SC5	SC2,SC3,SC4,SC5	0.80	1.00
SC6	SC6,SC7,SC8,SC9,SC10	SC6,SC7	0.40	1.00
SC8	SC6,SC7,SC8,SC9,SC10	SC8,SC9,SC10	0.60	1.00
SC11	SC11,SC12,SC13,SC14,SC15	SC11	0.20	1.00
SC12	SC11,SC12,SC13,SC14,SC15	SC12	0.20	1.00
SC15	SC11,SC12,SC13,SC14,SC15	SC13,SC14,SC15	0.60	1.00
SC19	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
SC22	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
SC26	SC26,SC27,SC28,SC29,SC30	SC26	0.20	1.00
SC28	SC26,SC27,SC28,SC29,SC30	SC27,SC28,SC29,SC30	0.80	1.00
SC31	SC31,SC32,SC33,SC34,SC35	SC31	0.20	1.00
SC33	SC31,SC32,SC33,SC34,SC35	SC32,SC33,SC34,SC35	0.80	1.00
SC37	SC36,SC37,SC38,SC39,SC40	SC36,SC37	0.40	1.00
SC38	SC36,SC37,SC38,SC39,SC40	SC38,SC39,SC40	0.60	1.00
SC43	SC41,SC42,SC43,SC44,SC45	SC41,SC42,SC43,SC44,SC45	1.00	1.00
SC46	SC46,SC47,SC48,SC49,SC50	SC46	0.20	1.00
SC50	SC46,SC47,SC48,SC49,SC50	SC49,SC50,SC49,SC50	0.80	1.00
Average			0.56	1.00

Table 7.8: FSC recall and precision performance of 0.25 rejection ratio ( $\eta$ ) value

Cluster Number Selected as Index	Software Component Relevant	Software Component Retrieved	Recall	Precision
SC1	SC1,SC2,SC3,SC4,SC5	SC1	0.20	1.00
SC3	SC1,SC2,SC3,SC4,SC5	SC2,SC3,SC4,SC5	0.80	1.00
SC6	SC6,SC7,SC8,SC9,SC10	SC6,SC7	0.40	1.00
SC8	SC6,SC7,SC8,SC9,SC10	SC8,SC9,SC10	0.60	1.00
SC11	SC11,SC12,SC13,SC14,SC15	SC11,SC12	0.40	1.00
SC15	SC11,SC12,SC13,SC14,SC15	SC13,SC14,SC15	0.60	1.00
SC19	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
SC22	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
SC26	SC26,SC27,SC28,SC29,SC30	SC26,SC27,SC28,SC29,SC30	1.00	1.00
SC28	SC26,SC27,SC28,SC29,SC30	SC26,SC27,SC28,	1.00	1.00
SC33	SC31,SC32,SC33,SC34,SC35	SC31,SC32,SC33,SC34,SC35	1.00	1.00
SC37	SC36,SC37,SC38,SC39,SC40	SC36	0.20	1.00
SC38	SC36,SC37,SC38,SC39,SC40	SC37,SC38,SC39,SC40	0.80	1.00
SC43	SC41,SC42,SC43,SC44,SC45	SC44,SC45,SC44,SC45	0.80	1.00
SC50	SC46,SC47,SC48,SC49,SC50	SC46,SC47,SC48,SC49,SC50	1.00	1.00
Average			0.73	1.00

Table 7.9: FSC recall and precision performance of 0.30 rejection ratio ( $\eta$ ) value

Cluster Number	Software Component Relevant	Software Component Retrieved	Recall	Precision
Selected as Index				
SC3	SC1,SC2,SC3,SC4,SC5	SC1,SC2,SC3,SC4,SC5	1.00	1.00
SC6	SC6,SC7,SC8,SC9,SC10	SC6	0.20	1.00
SC8	SC6,SC7,SC8,SC9,SC10	SC8,SC9,SC10	0.60	1.00
SC15	SC11,SC12,SC13,SC14,SC15	SC11,SC12,SC13,SC14,SC15	1.00	1.00
SC19	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
SC22	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
SC28	SC26,SC27,SC28,SC29,SC30	SC7,SC26,SC27,SC28,SC29,SC30	1.00	0.83
SC33	SC31,SC32,SC33,SC34,SC35	SC31,SC32,SC33,SC34,SC35	1.00	1.00
SC38	SC36,SC37,SC38,SC39,SC40	SC36,SC37,SC38,SC39,SC40	1.00	1.00
SC43	SC41,SC42,SC43,SC44,SC45	SC41,SC42,SC43,SC44,SC45	1.00	1.00
SC50	SC46,SC47,SC48,SC49,SC50	SC46,SC47,SC48,SC49,SC50	1.00	1.00
Average			0.89	0.98

Table 7.10: FSC recall and precision performance of 0.35-0.50 rejection ratio ( $\eta$ ) value

Cluster Number	Software Component Relevant	Software Component Retrieved	Recall	Precision
Selected as Index				
SC3	SC1,SC2,SC3,SC4,SC5	SC1,SC2,SC3,SC4,SC5	1.00	1.00
SC8	SC6,SC7,SC8,SC9,SC10	SC6,SC8,SC9,SC10	0.80	1.00
SC15	SC11,SC12,SC13,SC14,SC15	SC11,SC12,SC13,SC14,SC15	1.00	1.00
SC19	SC16,SC17,SC18,SC19,SC20	SC16,SC17,SC18,SC19,SC20	1.00	1.00
SC22	SC21,SC22,SC23,SC24,SC25	SC21,SC22,SC23,SC24,SC25	1.00	1.00
SC28	SC26,SC27,SC28,SC29,SC30	SC7,SC26,SC27,SC28,SC29,SC30	1.00	0.83
SC33	SC31,SC32,SC33,SC34,SC35	SC31,SC32,SC33,SC34,SC35	1.00	1.00
SC38	SC36,SC37,SC38,SC39,SC40	SC36,SC37,SC38,SC39,SC40	1.00	1.00
SC43	SC41,SC42,SC43,SC44,SC45	SC41,SC42,SC43,SC44,SC45	1.00	1.00
SC50	SC46,SC47,SC48,SC49,SC50	SC46,SC47,SC48,SC49,SC50	1.00	1.00
Average			0.98	0.98

Table 7.11: FSC recall and precision performance comparison

Rejection Ratio ( $\eta$ )Value	Number of Centers Select for Indexing Structure	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 7.12: RPCL software component selection with different degree of significance.

Structural	Functional	Behavioral	SC1	SC2	SC3	SC4	SC5	Component Selected
0.1	0.1	0.8	8.244	7.542	7.775	7.509	8.223	SC4
0.1	0.8	0.1	7.146	7.269	7.486	7.565	7.717	SC1
0.8	0.1	0.1	4.369	4.277	4.390	4.392	4.554	SC2
0.3	0.3	0.3	5.928	5.726	5.895	5.840	6.148	SC2

component is used to determine the degree of significance every participating software component. The most suitable component is one which has the highest degree of significance. This value is obtained by a simple procedure described below.

Suppose X denotes some arbitrarily chosen software requirements expressed in matrix form. A simple look-up over the indexing structure yields SC4, SC1, SC2, and SC2, depending on the degree of significance in structural, functional, and behavioral weights. This is shown in Table 7.12.

Table 7.13: FSC software component selection with different degree of significance

Structural	Functional	Behavioral	SC16	SC17	SC18	SC19	SC20	Component Selected
0.8	0.1	0.1	4.255	4.231	3.943	4.083	4.328	SC18
0.1	0.8	0.1	7.220	7.328	6.868	6.834	7.684	SC19
0.1	0.1	0.8	7.537	7.277	7.205	7.366	7.482	SC18
0.3	0.3	0.3	5.702	5.648	5.404	5.484	5.846	SC18

#### 7.4.4 Fine Grain FSC Selection

The FSC selection process was carried out in the same fashion as that of the RPCL, using cluster indexing structure from Table 7.7. The results are depicted in Table 7.13.

## CHAPTER VIII

### Conclusion

#### 8.1 Concluding results

This work has demonstrated the viability of formal approach and Z specification to define and classify existing software components based on their structural, functional, and behavioral property according to CMT guidelines. The assessment so obtained is further classified according to the de-facto standard certification procedures to ensure the closest similarity of the component being retrieved. The rigor and correctness of formal application to software component identification, specification, classification, and certification can ease the burden of software reuse in today's cyber-pace development cycle. As a consequence, the ever-growing software development cost and time can be reduced, as well as considerable improvement on the end-product quality.

This research 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.

The classification process utilizes four models to determine the proper model representation, namely, MDC, SOM, FSC, and RPCL. The MDC method is used as the baseline for comparative purpose. The remaining methods are unsupervised ANN techniques to be compared. A comprehensive experiment has been conducted to assess the

applicability of each model. All models yielded very good classification results. From precision, recall, and computation time, the results are apparent that RPCL outperforms other methods. The SOM took the longest time (over 30,000 time units) to train, as oppose to lesser magnitude (below 300 time units) performance by RPCL, FSC, and MDC. As far as classification is concerned, RPCL has the highest accuracy and requires moderate amount of training time. Under unsupervised fixed number of clusters situation, the RPCL method yields the highest precision, recall, and computation time performance. On the contrary, the performance of FSC method degrades drastically from 96% to 85% in precision, as explained in Chapter 6. Consequently, the RPCL is adopted as the proposed approach in classification and indexing for better retrieval efficiency.

Extensive computations are performed to arrive at a collection of clusters whose centers represent the structural, functional, and behavioral properties of member software components. The next step is to index all cluster centers of software components. As such, subsequent reference and retrieval can be carried out efficiently through this indexing mechanism. This process is called *software component classification*. An experiment was conducted to assess the validity of the proposed approach. It turned out to be quite satisfactory.

The proposed classification and indexing approach offers a viable step toward automated component repository storage and retrieval management.

## 8.2 Future work

Each ANN approach presented in this research possesses a number of inherent limitations of its own. There might be different advanced ANN techniques that are more efficient and suitable to software development classification application. The optimal mix of



structural, functional, and behavioral degree of significance can then be automatically computed by the underlying NN technique.

In addition, alternate indexing structures such as trees, pointers, hashing, or multi-layered hierarchical index will improve the speed and efficiency of retrieval operation.

It is envisioned that this work will furnish a viable basis for software certification. The rigor of formal specification will serve as a means for obtaining reliable component measure, which may either be direct (actual reliability) or indirect (Mean Time Between Failure) forms of acceptance or rejection measure. Future development will greatly support prevalent reuse with the help of reliable software classification. This in turn will enhance the productivity and quality of software products. The entire creation, classification, and certification process of software components will eventually evolve to machine learning research endeavor. As a consequence, component-based software reuse can be realized as their hardware counterpart.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## References

- [1] Baum, L. and Becker, M. Generic components to foster reuse. *Proc. 37th Int'l Conf. on Technology of Object-Oriented Languages and Systems TOOLS-Pacific* (2000): 266-277.
- [2] Baraldi, A. and Blonda, P., "A survey of fuzzy clustering algorithms for pattern recognition Part 1", *IEEE Trans. on Systems, Man, and Cybernetics-Part B: Cybernetics*, 29, 6 (1999): 778-785.
- [3] Baraldi, A. and Blonda, P., "A survey of fuzzy clustering algorithms for pattern recognition Part 2", *IEEE Trans. on Systems, Man, and Cybernetics-Part B: Cybernetics*, 29, 6, (1999): 786-801.
- [4] Bergner, K., Rausch, A., and Sihling, M., "Componentware-The big picture", <http://www.sei.cmu.edu/cbs/icse98/papers/p6.html> (June 1, 2004), in *Int'l Workshop on Component-Based Software Engineering*, (1998).
- [5] Caldiera, G. and Basili, V. R., "Identifying and qualifying reusable software components", *Computer*, 24, 2, (1991): 61-70.
- [6] Chang C., Chum, W. C., Liu, C., and Yang, H., "A formal approach to software components classification and retrieval", in *Proc. 21st Annual Int'l Computer Software and Applications Conf. COMPSAC '97*, (1997): 264-269.
- [7] Chen, P., Hennicker, R., and Jarke, M., "On the retrieval of reusable software components", in *Proc. 2nd Int'l Workshop on Advances in Software Reuse Software Reusability*, (1993): 99-108.
- [8] Charters, S. M., Knight, C., Thomas, N., and Munro, M., "Visualization for informed decision making; From code to components", in *Proc. 14th Int'l Conf. on Software*

*Engineering and Knowledge Engineering : SEKE'02*, ACM Press, in cooperation with ACM-SIGSOFT, Ischia, Italy, July 15-19, (2002): 765-772.

- [9] Cheung, Y. and Xu, L., "An RPCL-Based Approach for Markov Model Identification with Unknown State Number", *IEEE Signal Processing Letters*, (2000): 284-287.
- [10] Chiu, S., "Method and Software for Extracting Fuzzy Classification Rules by Subtractive Clustering", *Biennial Conf. of the North American Fuzzy Information Processing Society - NAFIPS '96*, (1996): 461-465.
- [11] Devanbu, P., Brachman, R. J., Selfridge, P. G., and Ballard, B. W., "LaSSIE: A knowledge-base software assistant", *Communications of the ACM*, (1991): 34-49.
- [12] Edwards, S. H., Gibson, D. S., Weide, B. W., and Zhupanov, S., "Software Component Relationships", <http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/edwards/edwards.html>, (June 2004).
- [13] Eklund, P., Kallin, L., and Riissanen, T., Fuzzy Systems, *Lecture notes prepared for courses at Department of Computing Science at Aumea University Sweden*, (February 2000).
- [14] Ehikioya, S. A., "A formal model for the reuse of software specifications", *IEEE Canadian Conf. on Electrical and Computer Engineering*, 1, (1999): 283-288.
- [15] King, I. and Lau, T. K., "Performance analysis of clustering algorithms for information retrieval in image databases", in *Int.l Joint Conf. on IEEE World Congress on Computational Intelligence*, (1998): 932-937.
- [16] Hafedh, M., Fatma, M., and Ali, M., "Reusing software: Issues and research directions", *IEEE Trans. on Software Engineering*, (1995): 528-562.
- [17] Haykin, S., *Neural Network*, Prentice Hall, (1999): 256-312.

- [18] Hong, S. B., and Kim, K., "Classifying and retrieving software components based on profiles", in *Proc. Int'l Conf. on Information, Communications and Signal Processing (ICICS)*, 3, (1997): 1756-1760.
- [19] Jeng, J., and Cheng, B. H. C., "Using Formal Methods to Construct a Software Component Library", *Lecture Notes in Computer Science*, vol. 717, in *Proc. 4th European Software Engineering Conf.*, (1993): 397-417.
- [20] Jeng, J., and Cheng, B. H. C., "A formal approach to reusing more general components", in *Proc. 9th Knowledge-Based Software Engineering Conf.*, (1994): 90-97.
- [21] Kan, L. T., *Rival Penalized Competitive Learning For Content-Based Indexing*, A dissertation for the degree of masters of philosophy, The Chinese University of Hong Hong, June 1998.
- [22] Klir, G. J. and Folger, T. A., *Fuzzy Sets, Uncertainty and Information*, Prentice Hall, (1988): 355.
- [23] Kohonen, T., Self-organized formation of topologically correct feature maps, *Biological Cybernetics*, (1982): 43.
- [24] Kohonen, T., The self-organizing map, *Proc. IEEE*, (1990): 1464-1480.
- [25] Kohonen, T., *Self-Organizing maps*, Springer-Verlag, Berlin, (1995).
- [26] Lano, K., and Haughton, H., *Object-Oriented Specification Case Studies*, Printice Hall, England.
- [27] Li, X. Q. and King, I., "Hierarchical Rival Penalized Competitive Learning binary tree for multimedia feature-based indexing", in *Proc. First Int'l Workshop on Intelligent Multimedia Computing and Networking (IMMCN2000)*, Atlantic City, New Jersey, (2000).

- [28] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., “An Information Retrieval Approach for Automatic Constructing Software Libraries”, *IEEE Trans. on Software Engineering*, (1991): 800-813.
- [29] Meilir, P. J., *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, The United States, (2000).
- [30] Nakkrasae, S. and Sophatsathit, P., “A Formal Approach for Specification and Classification of Software Components”, in *Proc. 14th Int’l Conf. on Software Engineering and Knowledge Engineering : SEKE’02*, ACM Press, in cooperation with ACM-SIGSOFT, Ischia, Italy, (July 15-19, 2002): 773-780.
- [31] Nishida, F., Takamatsu, S., Fujita, Y., and Tani, T., “Semi-Automatic Program Construction from Specification Using Library Modules”, *IEEE Trans. on Software Engineering*, (1991): 853-870.
- [32] Ostertag, E., Hendler, J., Diaz, R. P., and Braun, C., “Computing similarity in a reuse library system: An AI Base Approach”, *ACM Trans. on Software Engineering and Methodology*, (1992): 205-228.
- [33] Pedrycz, W., “Computational Intelligence as an Emerging Paradigm of Software Engineering”, in *Proc. 14th Int’l Conf. on Software Engineering and Knowledge Engineering: SEKE’02*, ACM Press, in cooperation with ACM-SIGSOFT, Ischia, Italy, July 15-19, ACM Press, (2002): 7-14.
- [34] Perry, D. E. and Popovitch, S. S., “In quire: Predicate-Based Use and Reuse”, in *Proc. 8th Knowledge-Based Software Engineering Conf*, (1993): 144-151.
- [35] Potter, B., Sinclair, J., and Till, D., *An Introduction to Formal Specification and Z, 2nd Edition*. Prentice Hall, England, (1996).

- [36] Pressman, R. S., *Software Engineering, A Practitioner's Approach*, 4th Edition, The McGraw-Hill Companies, Inc., The United States, (1997).
- [37] Saeki, M., "Behavioral specification of GOF design patterns with LOTOS", in *Proc. 7th Asia-Pacific Software Engineering Conf.*, APSEC, (2000): 408-415.
- [38] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, 1st Edition, Addison-Wesley, England, (1998).
- [39] Tvrđy, I., "Formal approach to reusable formal specifications", in *Proc. 6th Mediterranean Electrotechnical Conf.*, (1991): 1045-1048.
- [40] Wing, M. J., "A Specifier's Introduction to Formal Methods", *IEEE Computers*, (1990): 8-24.
- [41] Woodcock, J. C. P., "Structuring specifications in Z", *Software Engineering Journal*, 4, 1, (1989): 51-66.
- [42] Krzyzak, L. Xu, A. and Oja, E., "Rival penalized competitive learning for clustering analysis, RBF net, and curve detection", *IEEE Trans. on Neural Networks*, 4, 4, (1993): 636-648.
- [43] Xu, L., "Rival penalized competitive learning, finite mixture, and multisets clustering", in *The 1998 IEEE Int'l Joint Conf. on IEEE World Congress on Computational Intelligence*, Anchorage, Alaska, 3, (1998): 2525-2530.
- [44] Zaremski, A. M. and Wing, J. M., "Specification Matching of Software Components", *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 6, 4, (1997): 333-369.
- [45] Zaremski, A. M. and Wing, J. M., "Signature matching, a tool for using software libraries", *ACM Trans. on Software Engineering and Methodology (TOSEM)*, (1995): 146-170.

- [46] Zaremski, A. M. and Wing, J. M., “Specification matching software components”,  
in *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*,  
(1995): 6-17.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย



**APPENDICES**

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย



## Appendix A

### A-1 Abstract Data Type Repository (ADTR)

This section illustrates an abstract data type example, component equivalent classes, and component matrix formulation and transformation.

#### A-1.1 Structural Property Equivalence

Component name equivalent classes are composed of

$$EC(S_{1,1}) = LIFO\_STR$$

$$EC(S_{1,2}) = LILO\_STR$$

$$EC(S_{1,3}) = GRAPH\_STR$$

$$EC(S_{1,4}) = LIST\_STR$$

...

$$EC(S_{1,T_{s1}}) = \dots$$

Subcomponent name equivalent classes are composed of

$$EC(S_{2,1}) = LIFO\_STR$$

$$EC(S_{2,2}) = LILO\_STR$$

$$EC(S_{2,3}) = GRAPH\_STR$$

...

$$EC(S_{2,T_{s2}}) = \dots$$

Common Class name equivalent classes are composed of

$$EC(S_{3,1}) = Attribute\_STR$$

$$EC(S_{3,2}) = Method\_STR$$

...

$$EC(S_{3,T_{s3}}) = \dots$$

**Signature name equivalent classes are composed of**

$$EC(S_{4,1}) = LIFO\_PUSH$$

$$EC(S_{4,2}) = LIFO\_POP$$

$$EC(S_{4,3}) = LIFO\_UPDATE$$

$$EC(S_{4,4}) = LIFO\_CHECKDATA$$

$$EC(S_{4,5}) = LILO\_INSERT$$

$$EC(S_{4,6}) = LILO\_DELETE$$

$$EC(S_{4,7}) = LILO\_UPDATE$$

...

$$EC(S_{4,T_{s4}}) = \dots$$

**Interaction name equivalent classes are composed of**

$$EC(S_{5,1}) = LIFO\_Pushing$$

$$EC(S_{5,2}) = LIFO\_Popping$$

$$EC(S_{5,3}) = LIFO\_Updating$$

$$EC(S_{5,4}) = LIFO\_EmptyChecking$$

$$EC(S_{5,5}) = LILO\_Inserting$$

$$EC(S_{5,6}) = LILO\_Deleting$$

$$EC(S_{5,7}) = LILO\_Updating$$

$$EC(S_{5,8}) = LILO\_EmptyChecking$$

...

$$EC(S_{5,T_{s5}}) = \dots$$

## A-1.2 Functional Property Equivalence

Function name equivalent classes are composed of

$$EC(F_{1,1}) = LIFO\_PUSH$$

$$EC(F_{1,2}) = LIFO\_POP$$

$$EC(F_{1,3}) = LIFO\_UPDATE$$

$$EC(F_{1,4}) = LIFO\_CHECKDATA$$

$$EC(F_{1,5}) = LILO\_INSERT$$

...

$$EC(F_{1,T_{f1}}) = \dots$$

Input data equivalent classes are composed of

$$EC(F_{2,1}) = Integer$$

$$EC(F_{2,2}) = Character$$

$$EC(F_{2,3}) = String$$

$$EC(F_{2,4}) = Real$$

...

$$EC(F_{2,T_{f2}}) = \dots$$

Local Data equivalent classes are composed of

$$EC(F_{3,1}) = Integer$$

$$EC(F_{3,2}) = Character$$

$$EC(F_{3,3}) = String$$

$$EC(F_{3,4}) = Real$$

...

$$EC(F_{3,Tf3}) = \dots$$

**Output Data equivalent classes are composed of**

$$EC(F_{4,1}) = Integer$$

$$EC(F_{4,2}) = Character$$

$$EC(F_{4,3}) = String$$

$$EC(F_{4,4}) = Real$$

...

$$EC(F_{4,Tf4}) = \dots$$

**Pre-Expression equivalent classes are composed of**

$$EC(F_{5,1}) = Single\_Exp$$

$$EC(F_{5,2}) = AND\_Exp$$

$$EC(F_{5,3}) = OR\_Exp$$

$$EC(F_{5,4}) = NOT\_Exp$$

...

$$EC(F_{5,Tf5}) = \dots$$

**Post-Expression equivalent classes are composed of**

$$EC(F_{6,1}) = Single\_Exp$$

$$EC(F_{6,2}) = AND\_Exp$$

$$EC(F_{6,3}) = OR\_Exp$$

$$EC(F_{6,4}) = NOT\_Exp$$

...

$$EC(F_{6,T_{f6}}) = \dots$$

### A-1.3 Behavioral Property Equivalence

Behavioral name equivalent classes are composed of

$$EC(B_{1,1}) = LIFO\_Pushing$$

$$EC(B_{1,2}) = LIFO\_Popping$$

$$EC(B_{1,3}) = LIFO\_EmptyChecking$$

$$EC(B_{1,4}) = LILO\_Inserting$$

$$EC(B_{1,5}) = LILO\_Deleting$$

...

$$EC(B_{1,T_{b1}}) = \dots$$

These two predefined equivalent classes are used to identify the last three equivalent classes.

Behavioral State equivalent classes are composed of

$$EC(B_{2,1}) = Idle\_LIFO$$

$$EC(B_{2,2}) = Check\_LIFO$$

$$EC(B_{2,3}) = Push\_LIFO$$

$$EC(B_{2,4}) = Pop\_LIFO$$

$$EC(B_{2,5}) = Update\_LIFO$$

$$EC(B_{2,6}) = Idle\_LILO$$

$$EC(B_{2,7}) = Check\_LILO$$

$$EC(B_{2,8}) = Insert\_LILO$$

...

$$EC(B_{2,T_{b2}}) = \dots$$

Behavioral Action equivalent classes are composed of

$$EC(B_{3,1}) = ReturnStatus\_LIFO$$

$$EC(B_{3,2}) = CheckEmpty\_LIFO$$

$$EC(B_{3,3}) = PushData\_LIFO$$

$$EC(B_{3,4}) = PopData\_LIFO$$

$$EC(B_{3,5}) = UpdateData\_LIFO$$

$$EC(B_{3,6}) = ReturnStatus\_LILO$$

$$EC(B_{3,7}) = CheckEmpty\_LILO$$

$$EC(B_{3,8}) = InsertData\_LILO$$

...

$$EC(B_{3,T_{b3}}) = \dots$$

### **Behavioral Start State**

Behavioral Start State equivalent classes are the set of predefined start states.

### **Behavioral Action**

Behavioral Action equivalent classes are composed of the cartesian product between predefined state, predefined action, and predefined state. These equivalent classes will be re-labeled as the new equivalent classes. For example if all of predefined state equivalent class is equal 3 and predefined action equivalent class is equal 4, the behavioral action equivalent class is equal to 36 equivalent classes.

### **Behavioral Stop State**

Behavioral Stop State equivalent classes are the set of predefined terminating states.

Other equivalent ADTR classes in the software component repository can be defined by the users or system developers, namely, *STACK*, *QUEUE*, *TREE*, and *LIST*.

## A-2 Component Requirement Example

Component = { {PriorityQueue}, { }, { }, {Insert, Delete, Update}, {Interact1, Interact2, Interact3, Interact4}}

$$S_1 = \{PriorityQueue\} = \{S_{1,2}\}$$

$$S_2 = \{\} = \phi$$

$$S_3 = \{\} = \phi$$

$$S_4 = \{Insert, Delete, Update\} = F = \{S_{4,5}, S_{4,6}, S_{4,7}\}$$

$$S_5 = \{Interact1, Interact2, Interact3, Interact4\} = B = \{S_{5,5}, S_{5,6}, S_{5,7}, S_{5,8}\}$$

$$\begin{aligned} F_1 &= \{\{InsertPQ\}, \{Input1 : integer, Input2 : real, Input3 : real\}, \{Local1 : integer, Local2 : string\}, \{Output1 : integer\}, \{PreEx1ANDPreEx2\}, \{PostEx1\}\} \\ &= \{\{F_{1,5}\}, \{F_{2,1}, F_{2,4}, F_{2,4}\}, \{F_{3,1}, F_{3,3}\}, \{F_{4,1}\}, \{F_{5,2}\}, \{F_{6,1}\}\} \end{aligned}$$

$$F_2 = \{\{DeletePQ\}, \{Input1 : integer, Input2 : integer\}, \{Local1 : integer\}, \{Output1 : integer\}, \{PreEx1\}, \{PostEx1\}\}$$

$$F_3 = \{\{UpdatePQ\}, \{Input1 : integer\}, \{Local1 : integer\}, \{Output1 : integer\}, \{PreEx1ORPreEx2\}, \{PostEx1\}\}$$

Equivalent class mapping of  $F_2$  and  $F_3$  can be carried out in the same manner as  $F_1$ .

$$\begin{aligned} B_1 &= \{\{PQInserting\}, \{IdleStatePQ\}, \\ &\{IdleStatePQ\_CheckActionPQ\_CheckStatePQ\}, \end{aligned}$$

$$\begin{aligned}
& \text{CheckStatePQ\_ReturnActionPQ\_InsertStatePQ,} \\
& \text{InsertStatePQ\_InsertActionPQ\_IdleStatePQ}, \{\text{IdleStatePQ}\} \\
& = \{\{B_{1,4}\}, \{B_{2,6}\}, \{B_{2,6}\_B_{3,7}\_B_{2,7}, B_{2,7}\_B_{3,6}\_B_{2,8}, B_{2,8}\_B_{3,8}\_B_{2,6}\}, \{B_{2,6}\}\} \\
& B_2 = \{\{\text{InteracPQ2}\}, \{\text{StartStateName1}\}, \{\text{BehavioralActionName1}, \\
& \text{BehavioralActionName2}, \text{BehavioralActionName3}, \text{BehavioralActionName4}\}, \\
& \{\text{StopStateName1}\}\} \\
& B_3 = \{\{\text{InteracPQ3}\}, \{\text{StartStateName1}\}, \{\text{BehavioralActionName1}, \\
& \text{BehavioralActionName2}, \text{BehavioralActionName3}\}, \{\text{StopStateName1}\}\} \\
& B_4 = \{\{\text{InteracPQ4}\}, \{\text{StartStateName1}\}, \{\text{BehavioralActionName1}, \\
& \text{BehavioralActionName2}, \text{BehavioralActionName3}, \text{BehavioralActionName4}\}, \\
& \{\text{StopStateName1}\}\}
\end{aligned}$$

Equivalent class mapping of  $B_2$ ,  $B_3$ , and  $B_4$  can be carried out in the same manner as  $B_1$ .

Alternatively, if the component requirement is STACK, the equivalent class mapping becomes

$$\text{Component} = \{\{\text{STACK}\}, \{\}, \{\}, \{\text{Insert}, \text{Delete}, \text{Update}\}, \{\text{Interact1}, \text{Interact2}, \\
\text{Interact3}, \text{Interact4}\}\}$$

$$S_1 = \{\text{STACK}\} = \{S_{1,1}\}$$

$$S_2 = \{\} = \phi$$

$$S_3 = \{\} = \phi$$

$$S_4 = \{\text{Insert}, \text{Delete}, \text{Update}\} = F = \{S_{4,1}, S_{4,2}, S_{4,3}\}$$

$$S_5 = \{\text{Interact1}, \text{Interact2}, \text{Interact3}, \text{Interact4}\} = B = \{S_{5,1}, S_{5,2}, S_{5,3}, S_{5,4}\}$$

Likewise,



$$\begin{aligned}
F_1 &= \{\{InsertSTACK\}, \{Input1 : integer, Input2 : real, Input3 : real\}, \{Local1 : \\
integer, Local2 : string\}, \{Output1 : integer\}, \{PreEx1ANDPreEx2\}, \{PostEx1\}\} \\
&= \{\{F_{1,1}\}, \{F_{2,1}, F_{2,4}, F_{2,4}\}, \{F_{3,1}, F_{3,3}\}, \{F_{4,1}\}, \{F_{5,2}\}, \{F_{6,1}\}\}
\end{aligned}$$

By the same token,

$$\begin{aligned}
B_1 &= \{\{STACKInserting\}, \{IdleStateSTACK\}, \\
&\{IdleStateSTACK\_CheckActionSTACK\_CheckStateSTACK, \\
&CheckStateSTACK\_ReturnActionSTACK\_InsertStateSTACK, \\
&InsertStateSTACK\_InsertActionSTACK\_IdleStateSTACK\}, \{IdleStateSTACK\}\} \\
&= \{\{B_{1,1}\}, \{B_{2,1}\}, \{B_{2,1}-B_{3,2}-B_{2,2}, B_{2,2}-B_{3,1}-B_{2,3}, B_{2,3}-B_{3,3}-B_{2,1}\}, \{B_{2,1}\}\}
\end{aligned}$$

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## Appendix B

### B-1 Z language and Notations

This section furnishes only standard Z language syntax and notations used in this dissertation. Full coverage of Z can be found in [35] [41]

#### B-1.1 Syntax and Notations

1. The *basic types* of a specification are declared by enclosing them in square brackets, for example:

[Book, Car]

[Item]

2. A *declaration* may take the form  $x : T$  or  $x : S$ , where  $T$  is a type or  $S$  is a subset of a type. *Constraints* may be added by following the declaration(s) with a  $| Pred$  part which expresses some relationships amongst the declared variables. If the declaration is global, then an axiomatic description is needed (see below). The following examples show how a collection of variables may be declared, some are of the same type, some are of different types:

$m, n : N$

$i, j : 1 .. 8 \mid i \neq j$

$e1, e2 : S; s : PS \mid e1 \in s \wedge e2 \notin s$

3. New sets can be created from old ones using the powerset and cartesian product operators, i.e.,  $P$  and  $\times$ . If  $P$  and  $\times$  are applied to types, new types results. Every Z variable and expression belongs to one and only one type, although it may belong to many different sets.

4. An abbreviation definition introduces a global constant which takes the following

form:

$$name == \{ \text{declaration part} \mid \text{predicate part} \}$$

or

$$name == \{ \text{declaration part} \mid \text{predicate part} \bullet \text{Expression part} \}$$

For example,

$$\text{Signature } (\sigma) == \{ O_P : \text{set of } (n_O, In_O, Local_O, Out_O, Pre_O, Post_O) \mid$$

$$\forall n_O : \text{operation name}$$

$$In_O : P \text{ input parameter}$$

$$Local_O : P \text{ local variable}$$

$$Out_O : P \text{ output parameter}$$

$$Pre_O : \text{expression}$$

$$Post_O : \text{expression} \bullet$$

$$(n_O, In_O, Local_O, Out_O, Pre_O, Post_O) \in O_P \Rightarrow$$

$$n_O \in \text{ProperOperational\_name} \wedge$$

$$In_O \in F \text{ input parameter} \wedge$$

$$Local_O \in F \text{ local variable} \wedge$$

$$Out_O \in F \text{ output parameter} \wedge$$

$$Pre_O \in \text{expression} \wedge$$

$$Post_O \in \text{expression} \}$$

Note that,

$==$  is used to mean “is defined as”

$\Rightarrow$  is the connective stands between two true propositions. Such a proposition as “X is bigger than any city in Europe” can paraphrase to “For every city c, if c is in Europe then X is bigger than c” and can be formally written as follows:

$$\forall c : \text{city} \bullet c \text{ is in Europe} \Rightarrow X \text{ is bigger than } c$$

## B-1.2 Schema

Schema notation describes some aspects of a system, which is identified by the schema name. There are some points to note about schema notation:

- The name of the schema is introduced in the top line of the box.
- The central horizontal line separates the declaration part from the predicate part of the schema as depicted in Figure B-1. Figure B-2 is an example of schema notation.

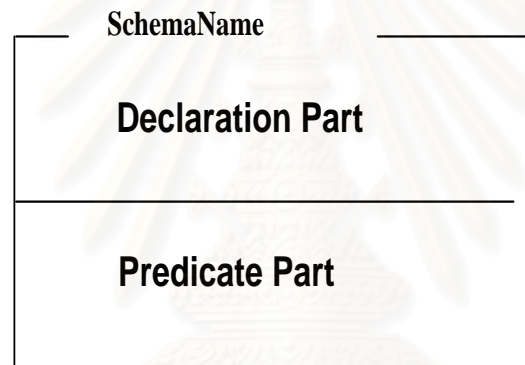


Figure B-1: Format Schema Type.

- If the predicate is of the form  $P1 \wedge P2$ , then  $P1 \wedge P2$  may be written on separate lines and the  $\wedge$  symbol elided; this may of course be extended to any number of conjunctions making up the predicate part.
- Similar declarations  $D1; D2$  may be written on separate lines, in which case, the semicolon can be omitted.
- By analogy with the declaration part, a predicate  $P1 \wedge P2$  may also be written as  $P1; P2$ . This also applies to any number of conjunctions.

SpecifyInsert	
<i>Max</i> : <i>N</i>	
<i>Insert?</i>	: operation name
{ <i>Data ?</i> : <i>integer</i> }	: P input parameter
{ <i>Index_A ?</i> : <i>integer</i> , <i>Top_S?</i> : <i>integer</i> , <i>A?</i> : <i>ArrayType</i> }	: P local variable
{ <i>A'?</i> : <i>ArrayType</i> }	: P output parameter
<i>Pre?</i> : <i>len(A)</i> < <i>Max</i>	: expression
<i>Post?</i> : <i>len(A')</i> = <i>len(A)</i> + 1 $\wedge$ <i>A'((len(A)) = Data</i> $\wedge$	
( <i>Index_A'</i> = <i>Index_A</i> + 1 $\vee$ <i>Top_S'</i> = <i>Top_S</i> + 1)	: expression
<i>Insert Signature!</i>	: $\sigma$
<hr/>	
<i>Max</i> $\in$ 100 $\wedge$	
<i>Insert ?</i> $\in$ F strings $\wedge$	
{ <i>Data?</i> : <i>integer</i> }	$\in$ F input parameter $\wedge$
{ <i>Index_A ?</i> : <i>integer</i> , <i>Top_S?</i> : <i>integer</i> , <i>A?</i> : <i>ArrayType</i> }	$\in$ F local variable $\wedge$
{ <i>A'?</i> : <i>ArrayType</i> }	$\in$ F output parameter $\wedge$
<i>Pre?</i> : <i>len(A)</i> < <i>Max</i>	$\in$ expression $\wedge$
<i>Post?</i> : <i>len(A')</i> = <i>len(A)</i> + 1 $\wedge$ <i>A'((len(A))=Data</i> $\wedge$	
( <i>Index_A'</i> = <i>Index_A</i> + 1 $\vee$ <i>Top_S'</i> = <i>Top_S</i> + 1)	$\in$ expression $\wedge$
<i>Insert Signature!</i>	$\in$ $\sigma$

Figure B-2: Example of Signature Specification.

- The order of declaration makes no difference to the meaning of a schema nor does the order of the conjunctions in the predicate.
- $P S$  is the set of all subsets of the set  $S$  and  $F S$  is the set of all finite subsets of a given set  $S$ . This can be expressed as follows:

$$F S \subseteq P S$$

- input identifier has ? as final character.
- output identifier has ! as final character.
- preceding objects are represented in plain variables, whereas the corresponding succeeding objects by dashed variables.

## VITA

**Name:** Mr.Sathit Nakkrasae.

**Date of Birth:** 16<sup>th</sup> March 1974.

**Education:**

- Ph.D. Program in Computer Science, Department of Mathematics, Chulalongkorn University, Thailand, (June 2000 - July 2004)
- Visiting Ph.D. researcher in CACS at University of Louisiana at Lafayette, Louisiana, United State, (January 2002 - December 2002).
- M.Sc. in Information Science, King Mongkut's Institute of Technology Ladkrabang, Thailand. (October 1995 - February 1998).
- B.Sc. in Computer Science at Ramkhamhaeng University, Bangkok, Thailand. (June 1992 - April 1995).

**Publication:**

- S. Nakkrasae, P. Sophatsathit, and W. R. Edwards, Jr., "Fuzzy Subtractive Clustering Based Indexing Approach For Software Components Classification", in *the International Journal of Computer & Information Science*, vol. 5 No.1, pp. 63-72, March 2004.
- S. Nakkrasae, P. Sophatsathit, and W. R. Edwards, Jr., "Fuzzy Subtractive Clustering based Indexing Approach For Software Components Classification," in *Proc. 1st ACIS Int'l Conf. on Software Engineering Research & Applications (SERA'03)*, San Francisco, USA, June 25-27, pp. 100-105, 2003.
- S. Nakkrasae and P. Sophatsathit, "A Formal Approach for Specification and Classification of Software Components", in *Proc. of the Fourteenth Int. Conf. on Software Engineering and Knowledge Engineering: SEKE'02*, in cooperation with ACM-SIGSOFT, Ischia, Italy, July 15-19, pp. 773-780, 2002.

**Scholarship and awards:** Office of the Commission for Higher Education.