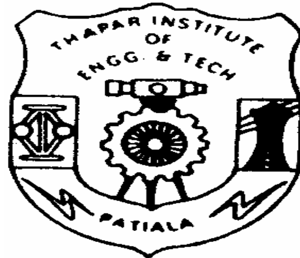
HYBRID APPROACH FOR THE CLASSIFICATION OF CANDIDATE COMPONENTS

A thesis
submitted for the partial fulfillment of requirement
for the award of the Degree
of

Master of Engineering
in
(Software Engineering)



Under the guidance of
Mr. Rajesh.K.Bhatia
Assistant Professor
Computer Science and Engineering Department
Thapar Institute of Engineering and Technology,Patiala

Submitted By
Damandeep Kaur
(8023124)

Computer Science and Engineering Department
Thapar Institute of Engineering and Technology,Patiala
(Deemed University),Patiala-147001(India)
June 2004

Acknowledgement

To discover, analyze and to present something new is to venture on an untrodden path towards an unexplored destination is an arduous adventure unless one gets a true torchbearer to show the way. This enlightening guidance, I found in my revered guide Mr. Rajesh.K.Bhatia, Assistant Professor, Computer Science & Engineering Department, Thapar Institute of Engineering & Technology (Deemed University), Patiala, without whose patronization it was never possible to give final shape to this thesis. I express my heartfelt gratitude towards him for his valuable guidance, encouragement, constant involvement, inspiration and the enthusiasm with which he solved my difficulties.

I shall be failing in my duties if I do not express my deep sense of gratitude towards Mrs. Seema Bawa, Assistant Professor & H.O.D, Computer Science & Engineering Department, Thapar Institute of Engineering & Technology, Patiala, who has been a constant source of inspiration for me.

Last but not the least, I express my heartfelt thanks to my parents, other members of the staff, my friends and well – wishers for co-operation, which they were always ready to extend.

Damandeep Kaur

Roll no. 8023124

Abstract

Software reuse is a way of increasing productivity and quality of a software system. An important prerequisite for successful software reuse is the choice of the most suitable components. Main problem encountered when reusing the component libraries is component retrieval; i.e. finding the components in the library that can be used in the construction of a specific information system. The formal specifications represent software that has been implemented and verified for correctness. Case-base Reasoning (CBR) system use various techniques to match a situation as a problem description, i.e. a case, to a database and known case. The goal of case retrieval is to return case that is most similar to the input attributes. Lastly rough-fuzzy hybridization mainly focuses to discover redundancies and dependencies between given features of a data to be classified The approach presented in this thesis is not fully dependent on any one concept, instead benefits of all the three concepts together in the retrieval and classification of a candidate component. In the present work, a case-generation approach using rough-fuzzy hybridization for retrieving the candidate component used in the previous projects to be reused in the query application has been proposed. The approach is based on rough-fuzzy hybridization and structural matching and a model is proposed on the basis of this approach. The proposed model reduces the domain search and helps in retrieving the best suitable candidate component.

Declaration

I hereby certify that the work which is being presented in the thesis entitled "Hybrid Approach for Classification of Candidate Components " in the partial fulfillment of the requirements for the award of the degree of Master of Engineering (Software Engineering) of Thapar Institute of Engineering & Technology (Deemed University), Patiala, is an authentic record carried out under the supervision of Mr. Rajesh K. Bhatia. The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other University.

Damandeep Kaur

This is to certify that the above statement made by the candidate is correct and true to best of knowledge.

Mr. Rajesh. K. Bhatia
Assistant Professor
Computer Sc. and Engg. Department
Thapar Institute of Engg. And Tech., Patiala

Countersigned by

(Seema Bawa)
Assistant Professor & Head,
Computer Sc. & Engg. Department,
Thapar Institute of Engg. & Technology,

(Dr. D. S. Bawa)
Dean (Academic Affairs)
Thapar Institute of Engg & Technology,
Patiala.

The M.E. (Thesis) Viva-Voice examination of Damandeep Kaur, Roll No. 8023124, M.E. (Software Engineering), Thapar Institute of Engineering and Technology, Patiala has been held on

Supervisor **External Examiner**

Contents

Abstract ***i***

Declaration *ii*

Acknowledgement *iii*

Contents ***iv***

List of Figures and Tables *vi*

Organization of Thesis vii

Chapter 1. Introduction **1-8**

1.1 Reuse Expectation 2

1.2 Reuse Process 2

1.3 Reuse Life Cycle 3

1.3.1 Qualification and Generalization 4

1.3.2 Classification 5

1.3.3 Evolution 5

1.3.4 Retrieval 6

1.3.5 Understanding and Adaptation 7

1.3.6 Composition 7

1.4 Reuse Benefits 7

1.5 Reuse Drawbacks 8

1.6 Summary 8

Chapter 2. Retrieval & Classification Concepts **9-23**

2.1 Information Retrieval System 9

2.2 Measures of Retrieval System 10

2.3 Main Approaches to Classification & Retrieval 11

2.3.1 Classification Scheme 11

2.3.2 Automatic Indexing 12
2.3.3 Knowledge-based Approaches 15
2.3.4 Formal Specification 17
2.3.5 Behavioral-based Retrieval 17
2.3.6 Neural network-based Retrieval 20
2.3.7 Browsing 21
2.3.8 Hypertext 22
2.4 Summary 23
Chapter 3. Hybrid Approach for Classification of **Candidate Components using Rough-Fuzzy** *24-45*
3.1 Problem Description 24
3.2 Z Specification 25
3.3 Structural Matching 27
3.4 Case-Based Reasoning 30
3.5 Rough-Fuzzy Theory 31
3.6 Proposed Model for Component Classification - Reuse MRC 34
3.6.1 Database for Z specifications of Components 34
3.6.2 Steps for Matching, Retrieval & Classification 35
3.6.3 Case Study 41
Chapter 4 Conclusion & Future Scope 47-49
4.1 Conclusion 47
4.1.1 Assumptions and Constraints 48
4.2 Future Scope 49

References *50*

Appendix A **54**

Appendix B **57**

List of Papers **60**

List of Figures and Tables

Figures

Figure 1.1 Reuse Life Cycle 3

Figure 1.2 Main activities in Development For Reuse 6

Figure 1.3 Main activities in Development With Reuse 7

Figure 2.1 Software Retrieval Approach 10

Figure 2.2 Facet Classification 12

Figure 2.3 Automatic Indexing
13
Figure 2.4 Syntactic Method Example
14
Figure 2.5 Case Frames
16
Figure 2.6 Process of Behavioral Process
19
Figure 2.7 Browsing
22
Figure 3.1 Lattice of Satisfaction criteria
30
Figure 3.2 Generic Structure of CBR
31
Figure 3.3 ReuseMRC Tool
34
Figure 3.4 FlowChart
39
Figure 3.5 Initial search results
41
Figure 3.6 “sf9” Specification
42
Figure 3.7 “sf10” Specification
42
Figure 3.8 “sf11” Specification
42
Figure 3.9 Software Attributes for Query
43
Figure 3.10 Final Query Results
45
Figure 4.1 Measures of Reuse Effectiveness
47

Tables

Table 3.1 Software attributes with membership values

36 3.2 Assigned Fuzzy membership values

38

Table 3.3 Query Component Specification

41

Table 3.4 Query Results after Structural Matching

42

Table 3.5 Candidate Application Retrieved from CBR

44

Table 3.6 Judgement Table

44

Table 3.7 Rough-fuzzy membership value decision

46

Thesis is organized in four main parts, covering Details of Software Reuse, Component Retrieval Techniques, idea of case-based reasoning, rough-fuzzy theory and a model proposed for the software candidate classification.

Chapter 1 provides the basic introduction about the software reuse. Reuse benefits and drawbacks have been dealt with.

Chapter 2 gives the overview of the current trends and various component retrieval techniques.

Chapter 3 describes the proposed model for the classification of the software candidate components using rough-fuzzy hybridization.

The work done has been concluded along with future directions in Chapter 4.

CHAPTER 1 Introduction

In mid 60's, software began to develop from an art or craftsmanship to a profession. It was pushed ahead by the quest for better control of software costs, quality and time to develop a system. The birthplace of the term software engineering, which subsumes this thinking, was seen at the 1968 NATO Conference in Garmisch, Germany. There, the main focus was on finding ways to fight the software crisis.

As a solution to the software crisis, McIlroy proposed the idea of highly standardized software building blocks as a key to industrial like software mass production. Given a large repository of such components, together with automated techniques for customizing these components to the developer's needs.

The developer rather asked the question “Which component shall I *use*?” than “Which component shall I *build*?” this idea of reusing software assets was persuasive enough for the research community to adopt it at once. What is Software reuse [1, 2, 3, and 4] after all? The idea is a very simple one and in the literature of software engineering one can find a variety of definitions.

“Software reuse is using existing software artifacts during the construction of a new software system”.

“The process of implementing or updating software systems using existing software assets.”

Both definitions do not limit the software artifacts to source code only.

1.1 Reuse Expectations

There are three main reason [5], why the effort to build and maintain a reuse process should pay off and help to deliver software, which is less costly:

(a) Quality

The increase in quality is due to an increase in reliability, consistency, manageability, and standardization.

(b) Productivity

The increase in productivity is due to the higher capacity of programmers, since less effort goes into documentation and testing, and the maintainability is being simplified.

(c) Time to market

By saving time through incorporating existing assets into a software system, time to market can be reduced drastically, too.

1.2 Reuse process

Reuse may occur in two different ways [6, 7]: -

1. Systematic reuse:

- This is also called institutionalized reuse
- It is the style of software development, in which the inclusion of reusable products during all stages of software development i.e. requirements, analysis, design, implementation, and test is an integral part of the development process.

2. Opportunistic reuse:

- This style happens in an ad hoc-manner.
- During the software development, individuals recognize patterns within their work, which occurred already in former projects. Such patterns can range from products like analysis documents, algorithms, to subroutines or whole subsystems. According to their individual knowledge, these persons then incorporate assets into the current product.

Effective Systematic reuse needs a well-defined way to generate assets and to use them. Therefore, defined processes for

1. Development *for* reuse
2. Development *with* reuse must be established.

Such processes, which are different from that of a conventional software development process, should be adapted to the special needs of a reuse centric paradigm.

1.3 A reuse life cycle

Software reuse involves two major complementary activities as shown in figure 1.1: -

1. Developing reusable software
2. Reusing existing software.

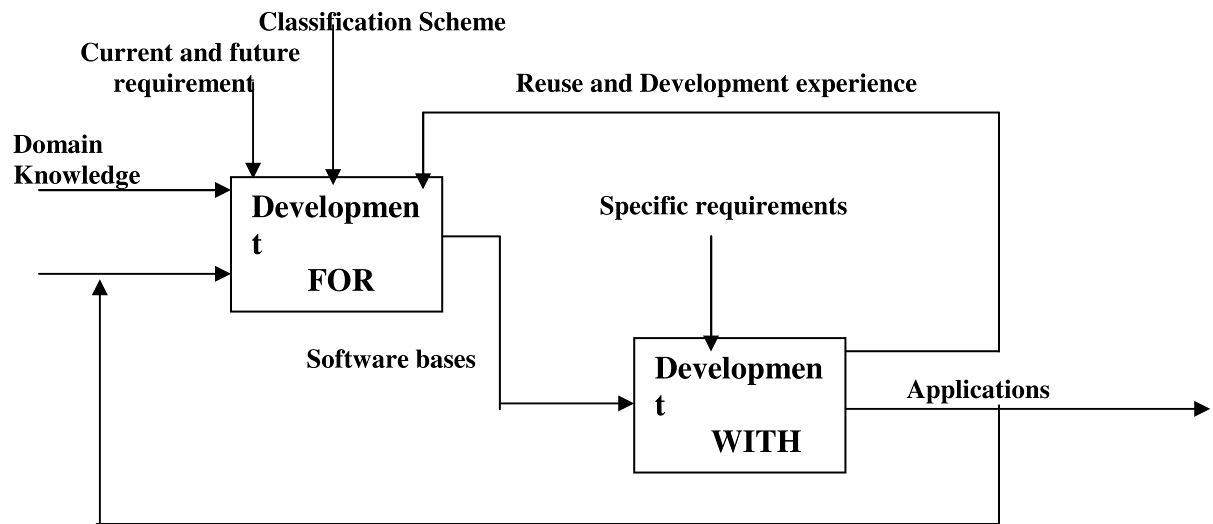


Figure1.1 Reuse Life Cycle

Development FOR Reuse or Application Engineering [6,7,8] involves

- Identifying that can be widely reused.
- Defining components and understanding them for reuse.
- Classifying and storing the reusable components in a library.
- Representing reusable components in a standard form.

Development WITH Reuse or Application Development, [6,7,8] involves

- Find the reusable components
- Understand the reusable components
- Modify the reusable components
- Combine and incorporate the reusable components

In spite of their different roles, Development FOR Reuse and Development WITH Reuse are related and complementary activities.

1.3.1 Qualification and generalization

General-purpose software components constructed by identifying common features of specific software components. The generalization process is supported by a particular abstraction mechanism e.g. functional abstraction, abstract data types, and object classes.

Component Qualification [7,8,9] ensures that a candidate component will perform the function required, will properly "fit" into the architectural style specified for the system and will exhibit the quality characteristics e.g. performance, reliability, usability that are required for the application.

1.3.2 Classification

The classification [7, 8, 9] process catalogues a software component in a software base by mapping the component into an internal representation structure through which the component can be later retrieved.

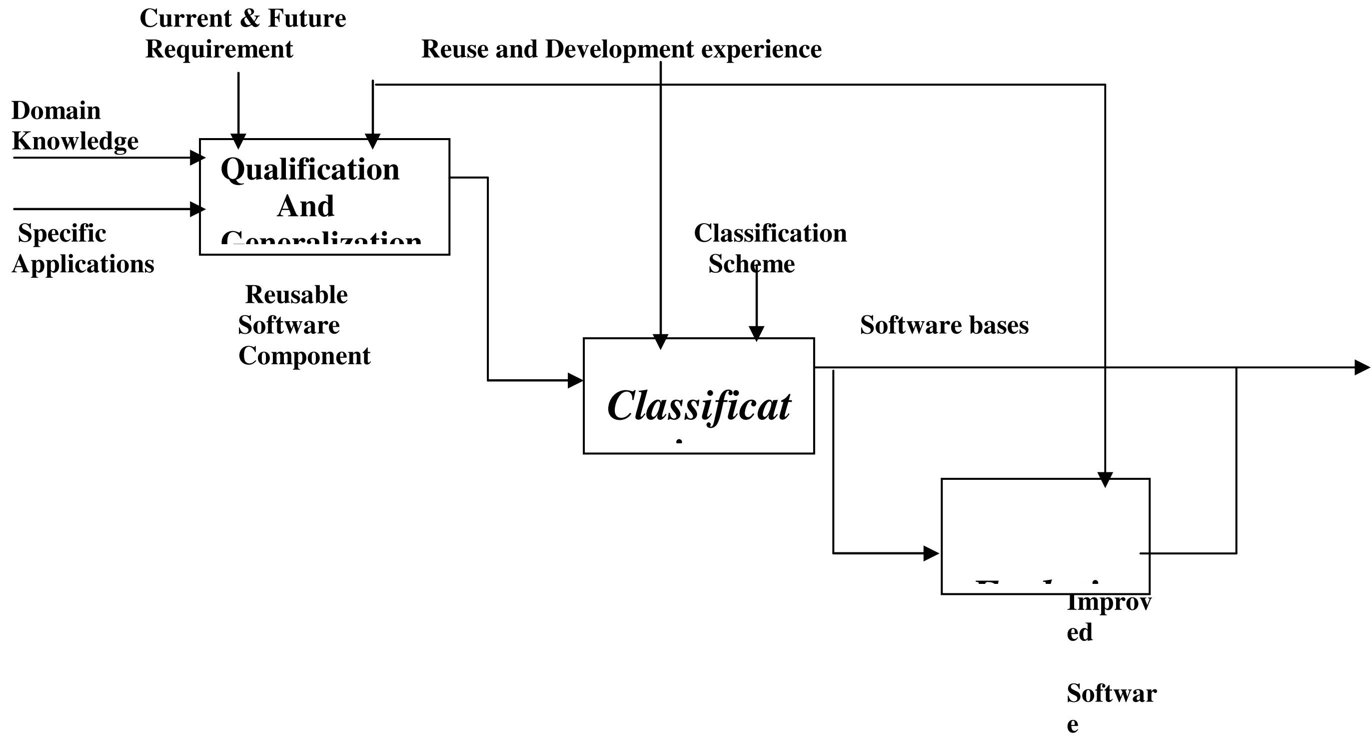
In traditional classification mechanisms for software reuse, this internal representation structure consists of a set of terms that describe the component according to categories in a classification scheme or representation language. The assignment of terms called indexing can be either free or under a controlled vocabulary. The controlled vocabulary establishes the terms that can be used in the indexing process.

1.3.3 Evolution

Software bases must evolve to be adapted to the evolution [8, 9] of the application domain and to improve the information they contain, in order to make the retrieval more effective. This requires:

- to maintain in the software base only those components with an actual potentiality for reuse and remove the obsolete ones;
- to reclassify or to reorganize the internal representation of components to make their retrieval easier and more effective;
- to improve the available information for the understanding and adaptation of the components and even
- to redesign the software components or their hierarchies to make them more reusable.

For that, it is necessary to keep information of the reuse experience that different users have had with the system, e.g. the frequency with which a component is reused, the general adaptability of a component to the user's requirements and the appropriateness of the internal representation of the components.



bases

Figure 1.2 - Main activities in Development FOR Reuse

1.3.4 Retrieval

Through the retrieval process [7, 8, 9] potential reusable software components, satisfying specific user's requirements, are searched and selected from software bases. The searching can be done by browsing a hierarchy of software components in the software base, by following hypertext or hypermedia links in the software base or by linear retrieval.

Retrieval mechanisms in reuse systems can also include measures for assessing the reuse potential and ease of adaptation of the retrieved components like frequency of reuse, quality of documentation and complexity of the component like size, number of inputs and outputs, etc.

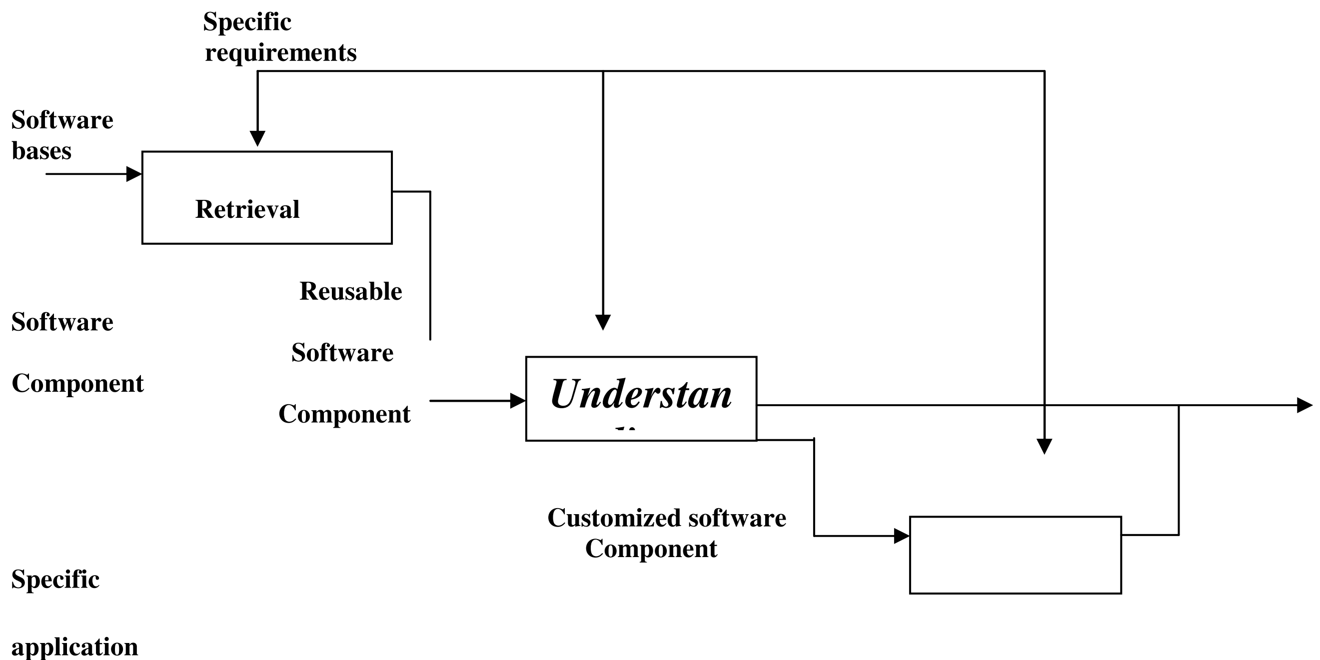


Figure 1.3 - Main activities in Development WITH Reuse

1.3.5 Understanding and Adaptation

A retrieved component satisfying the user's requirements can be reused as a black box by instantiating it or by specialization e.g. by inheritance to customize it to the particular requirements of an application. The adaptation task is related to the understanding of the component. More difficult is the understanding of a component, more difficult its adaptation to the new requirements.

1.3.6 Composition

Component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application.

1.4 Reuse Benefits

Adopting reused-based development process attracts a number of well-organized economic and psychological benefits to both the end-users and developers. These include the following:

- ✓ Savings in costs and time
- ✓ Increase in productivity
- ✓ Increase in reliability
- ✓ Increase in ease of maintenance
- ✓ Improvement in documentation and testing
- ✓ High Speed and low cost replacement of aging systems

1.5 Reuse Drawbacks

At the same time, in practice, radical gains in productivity and quality cannot be achieved due to some preconceptions held by developers and their management. These are:-

- ✓ Reusing code, as compared with development of entirely new systems , is boring
- ✓ No Formal training in reusing code and design effectively
- ✓ Reuse process is too slow
- ✓ Cost of maintaining reusable libraries is prohibitive.

1.6 Summary

In this chapter, the concepts of software reuse, reusability and software library has been explored. We have discussed the benefits of reusing software components. A reuse process model is also considered which shows the incorporation of software reuse process in the development life cycle. Although the demand for reusable building blocks was already realized a long time ago, many issues remain open. The principal problems of classification and retrieval of software artifacts have been discussed in the next chapter. Various current approaches to software classification and retrieval have been analyzed, principally on their advantages and limitations to cope with the problems.

CHAPTER 2
Retrieval and Classification concepts

In software information retrieval [10, 11], the main purpose is not to retrieve all the documents in which a particular pattern exists but rather those best describing the desired component functionality. Retrieval effectiveness is a critical factor in these kinds of systems.

2.1 Information retrieval system

The purpose of any Software Retrieval [11] approach is to satisfy a variety of information's needs. A query or request formulates an information need and the Software Retrieval approach will answer with a list maybe empty of information items in a database. The majority of the information items are in the form of pure text called documents in spite of the increasing importance of multimedia information items

consisting of formatted text, graphics, sound, video sequences, etc. Information item databases can also be composed of software development products, like requirements and design specifications, executable programs, test cases, etc and such a database is called a software base**.**

Figure 2.1 shows the major functions of any Software Retrieval approach:

- Indexing;
- Search strategy;
- Matching and, usually, similarity analysis;
- Evaluation of retrieved items and, possibly, feedback for query refinement.

The process of indexing or cataloguing is essentially a classification process that is performed through conceptual analysis of the information item.

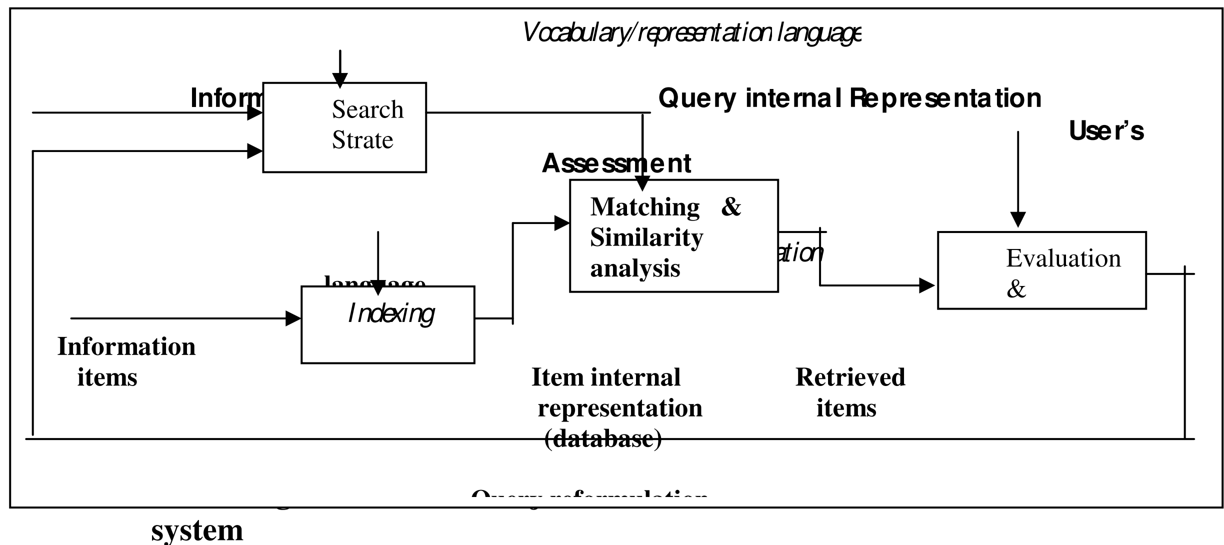


Figure 2.1 Software Retrieval Approach

2.2 Measures of retrieval effectiveness

The effectiveness of a Retrieval [5] approach expresses how well the produced output, i.e. the ranked list of retrieved items satisfies information need. This is accomplished by retrieving as many useful items and as few of the useless ones as possible.

The usual measures of retrieval effectiveness are recall and precision. Both measures are based on the user's relevance assessments following the retrieval process. Recall measures the completeness of the output, i.e. the ability of the system to retrieve useful relevant information. Precision measures the relevance of the output, i.e. the ability of the system to reject useless irrelevant or noise material.

2.3 Main approaches to software classification and retrieval

This section proposes taxonomy and summarizes major recent approaches [7, 8, 9] for software classification and retrieval:

- Classification schemes
- Automatic-indexing
- Knowledge-based approaches
- Formal specifications
- Behavior-based retrieval
- Neural network-based retrieval
- Browsing
- Hypertext

2.3.1 Classification schemes

Main approaches for software classification and retrieval propose a classification scheme [13,14] for cataloguing components in a software base. The schemes specify some attributes called facets to be used as descriptors of a software component, mostly focusing on the action that a component performs and on the objects manipulated by the component.

Prieto-Diaz proposes a faceted classification scheme [15] for cataloguing software components. This scheme has also been adopted or generalized by other reuse proposals. The scheme uses facets as descriptors of software components. The scheme consists of four facets: Function performed by the component, Object manipulated by the function, Data structure where the function takes place, medium or location, and System to which the function belongs.

The three first facets are considered sufficient to describe the functionality of a component. For example, the following terms, corresponding to each one of the facets we have circled on the vocabulary of figure 2.2, describe the grep family of Unix commands that search function facet, a string objects facet in a file medium/data structure/location facet. The same terms and facets along with the term line-editor in the system facet could be used to describe part of the functionality of line editor commands.

A Simplified Faceted Scheme for UNIX components

DOMAIN ⟶ Unix Tools

{by action}	{by Object}	{by data structure}	{by system}
get	file-names	buffer	line-editor
put	identifier	tree	text-formatter
update	line-number	table	
append	character	file	
check	number	archive	
detect	expression		
locate	entry		
search	declaration		
evaluate	line		

Figure 2.2 Facet Classification

2.3.2 Automatic indexing

Free-text indexing systems automatically extract keywords from queries in natural language and these keywords are used to locate software components. Similarly, software components are classified in a software base by indexing them according to keywords extracted from the natural language documentation of the software components.

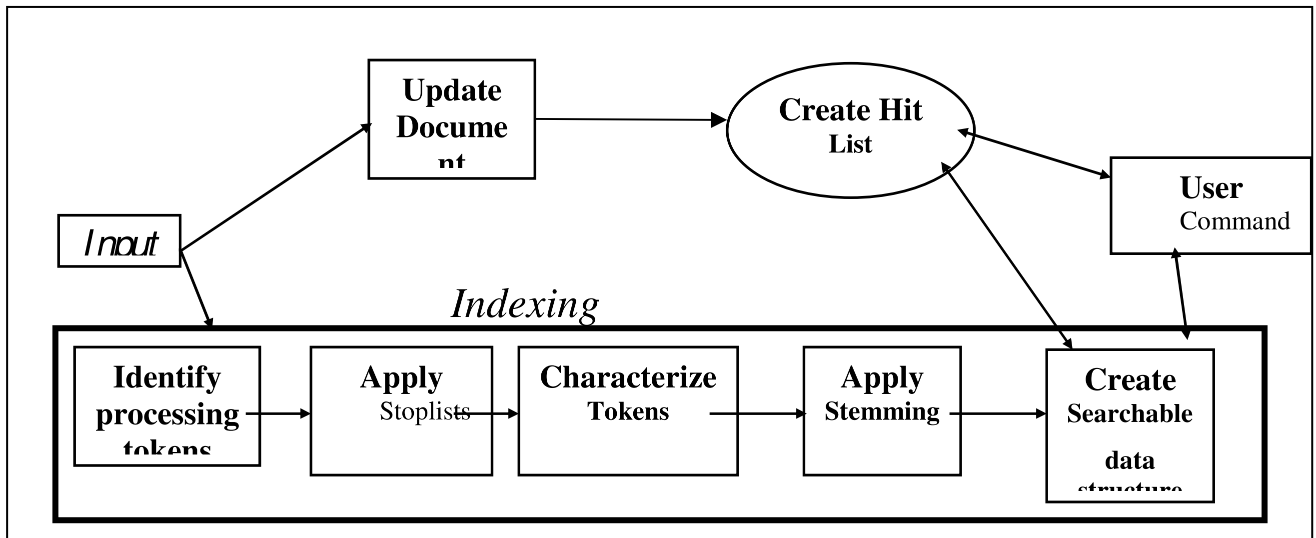


Figure 2.3 Automatic Indexing

Automatic indexing [17] methods in information retrieval are of two types:

(a) Statistical approaches

- Statistical approaches [8] to information retrieval are based on the assumption that high frequency words, i.e. words that occur very frequently in a collection, are not

useful in distinguishing or discriminating sets of information items. Then, most of these approaches use a stop list to exclude high frequency words from documents and queries in natural language.

Salton describes this procedure for the SMART system:

- Use a stop list to eliminate common function words.
- Use a stemmer to generate word stems.
- Take pairs of the remaining word stems, and let each pair define a phrase provided that the distance in the text between the components of the phrase does not exceed a desired threshold, and that at least one of the components of each phrase is a high frequency term.

(b) Linguistic approaches

In most linguistic approaches to IRSs, NLP techniques have been applied at the morpholexical, lexical, syntactic and semantic level.

1. Morpholexical level

Morpholexical processing techniques work at the single word level and consist in identifying both the standard forms and the grammatical classes of a word by looking up a dictionary or lexicon. For example, lexical processing would determine that the word "searching" is either the present continuous tense of the verb "search" or and adjective. For instance, for the word "searching", both the standard forms "search" and "searching" will be identified, associated with the verb or adjective grammatical categories, respectively.

2. Syntactic level

Syntactic analysis [8] is used to determine the structure of a sentence. The syntax is specified in a grammar establishing the set of rules that determine which ordering of words are allowed and parsing strategies are used to recognize the structure of the sentence.

Syntactic analysis in information retrieval has been mostly restricted to identify noun phrases. For instance, figure 2.4 shows the phrases that could be generated through syntactic analysis from the sentence *"search a paragraph tag in a text file."* Syntactic analysis permits the identification of different syntactic compounds like the main verb, the main object and a prepositional phrase in the sentence and the phrases can be extracted from each syntactic compound.

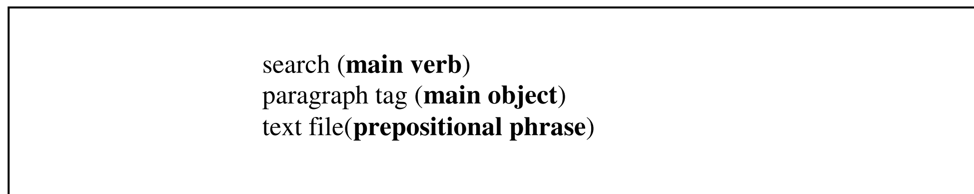

search (**main verb**)
paragraph tag (**main object**)
text file(**prepositional phrase**)

Figure 2.4 Syntactic method Example

3. Semantic level

The semantic level of NLP concerns with meaning and maps grammatically parsed text into knowledge representation formalism. Semantic analysis should disambiguate the meaning of sentences. Sentences can be ambiguous either because words within them are ambiguous i.e. they have more than one meaning or because they have many syntactic interpretations.

A famous example is the sentence "I saw the boy with the telescope", in which "with the telescope" says either "which boy I saw" or "how I saw him".

2.3.3 Knowledge representation approaches

Knowledge-based [18] software retrieval systems make some kind of lexical, syntactic and semantic analysis of natural language specifications of software components without pretending to completely understand the documents. They are based on a knowledge base, which stores semantic information about the application domain, and about natural language itself.

(a) Case frames

The most common semantic representation structure used in information retrieval and the one used in this work is a case frame. Case frame approaches [18] are based on the linguistic case theory of Fillmore described also in most NLP and AI literature. Systems using this approach parse natural language input text into units called case frames that are part of a given grammar called case frame grammar.

A case frame has a head concept modified by a set of related concepts or cases also called thematic roles. The head concept, a noun or a verb must match the input sentence somewhere for the frame to

be considered for parsing. A case marker is a word pattern that is a prefix of the subsidiary phrases

e.g. a preposition. The rest of the phrase, called the case filler, is another pattern or even another case frame. Figure 2.5 depicts two case frames associated with the verb search as a head concept. These semantic structures should allow the parsing of sentences like:

Case frame (a)

The command searches for a string in a file.

The package searches for a pattern within a text file.

Case frame (b)

The command searches a text file for a pattern.

The package searches a program for a string.

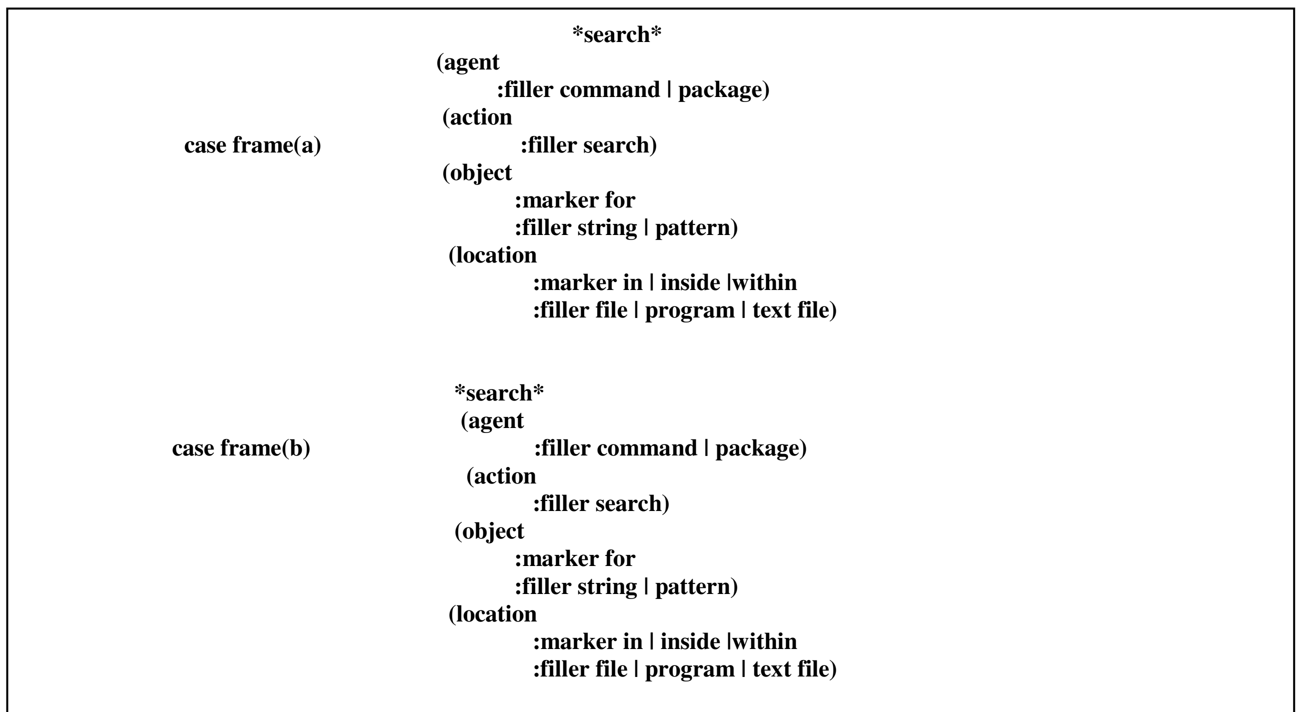


Figure 2.5 Case Frames associated with some of the meanings of the verb *"search"*

(b) Conceptual dependency theory

The Conceptual Dependency theory, developed by Schank is similar to the case theory in that the representation of the meaning of a sentence revolves around the action of the sentence and includes some number of cases to relate the other parts of the sentence to the action.

The "basic axiom" of conceptual dependency theory is: ``For any two sentences that are identical in meaning, regardless of language, there should be only one representation of that meaning in conceptual dependency". Conceptual dependency representations, CD, are made up of a very small number of semantic primitives, which include primitive acts and primitive states with associated attribute values. Relations among concepts are called dependencies and there is a fixed number of this, each represented graphically by a special kind of arrow.

2.3.4 Formal specifications

Formal interface specifications provide a solution for the requirement representation problem. Formal specifications [19] provide precise descriptions of Problem requirements; Component function; Component structure.

This effort considers formal interface specifications in three aspects of component reuse: (i) Retrieval of potential solutions (ii) Assessment of correctness (iii) Component adaptation using architectures. Retrieval and adaptation of individual components is a process of finding and modifying components that solve a problem.

2.3.5 Behavior-based retrieval

Behavioral retrieval [20] works by exploiting the executability of software components. Programs are executed using components, and the responses of components are recorded. Retrieval is achieved by selecting those components whose responses with respect to the program are closest to a pre-determined set of desired responses.

The behavioral retrieval scheme is outlined as follows: Take an abstract view of components, in terms of their execution responses to programs. Collections of responses can be meaningfully partially ordered. How the lattice

induced by the partial order can be used to find best behavioral approximations if an exact match cannot be found to the component desired.

(a) An Execution Model

In order to develop a theory of behavioral retrieval, an execution model is required to explain how components execute programs and generate output responses. In this execution model, a component is represented as a relation between programs and responses. Formally, a component *c* can be declared as:

$$c : Prog \leftrightarrow Response$$

(a) Ordering Responses

Behavioral retrieval of components critically depends upon having a sound basis for behavioral comparison; otherwise library retrieval operations would not select the most appropriate components

In order to obtain behavior $\mathcal{B}$ from a collection of responses $\mathcal{R}$, we apply the following filter function:

$$
\begin{array}{|l}
: \mathbb{P}\, Response \rightarrow Behaviour \\
\hline
\forall \mathcal{R} : \mathbb{P}\, Response \bullet \\
\quad (\mathcal{R}) = \{ r : \mathcal{R} \mid \nexists s : \mathcal{R} \bullet s \subset r \}
\end{array}
$$

The behavior $\mathcal{B}$ of a component can be now be denoted as: $\mathcal{B} = (c(\!|\ \{p\}\ |\!))$. Components whose behaviors with respect to a given program are identical are said to be *behaviorally equivalent*.

(b) The Partial Order and its Interpretation

Consider two such sets $\mathcal{B}_1$ and $\mathcal{B}_2$ constructed by executing a program *p* on two different components. A partial orders over behaviors $\mathcal{B}_1$ and $\mathcal{B}_2$ is defined as:

$$
\begin{array}{l}
\mathcal{B}_1 \sqsubseteq \mathcal{B}_2 \Leftrightarrow \\
\quad \forall t : \mathcal{B}_2 \bullet \exists s : \mathcal{B}_1 \bullet s \subseteq t
\end{array}
$$

(c) Retrieving using Behaviors

The retrieval process is now defined using the *meet* and *join* operations. A behavioral query consists of two parts:

- the program to execute (called *p*), which simulates the operating environment of the required component; and

- the desired behavior (called *b*) of the required component when operating in the environment defined by *p*.

Given a query (*p*,*b*), behavioral retrieval proceeds as shown in Figure 2.6

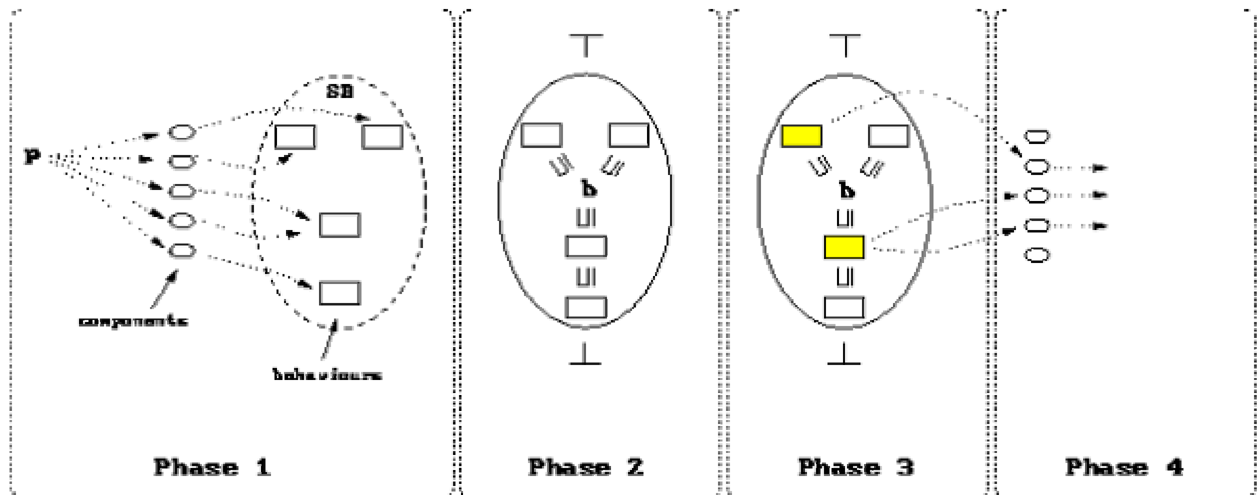


Figure 2.6 Process of Behavioral Retrieval

Phase 1: the program is executed on each component in the library, resulting in a set of behaviors *SB*;

Phase 2: the set of behaviors *SB* is by definition embedded in the lattice of $\sqsubseteq$. The position of the desired behavior *b* in the lattice is found.

Phase 3: the closest ancestor and descendant behaviors of *b* in *SB* (if any) are selected, using the lattice operators *meet* and *join*.

Phase 4: the components, which responded to program p with those closest behaviors (if any), are returned as the result of the retrieval.

The retrieval process is sound in the sense that if a set of behaviorally equivalent components *C* with a behavior exactly matching the desired behavior exists in the

component library. Then all members of *C* and no other components are returned as a result of the retrieval process i.e. perfect recall and precision.

2.3.6 Neural network-based retrieval

In retrieval systems based on a classification scheme, similarity computation is usually performed by using a conceptual distance graph, which establishes the similarities between terms in the controlled vocabulary of the classification scheme.

Manual adjustment of the graph is considered unreasonable as the scale of the software base increases, because the number of connections in the conceptual graph is considered as combinatorial explosive.

Neural network-based methods approach [21] the similarity computation in a different way. Instead of using conceptual distance graphs, they use artificial neural network technology to structure software bases according to the functional similarity of the stored software components, i.e. components that exhibit similar behavior are stored near to each other. Thus, the similarity between components is made explicit because it is determined according to the geographical closeness between components.

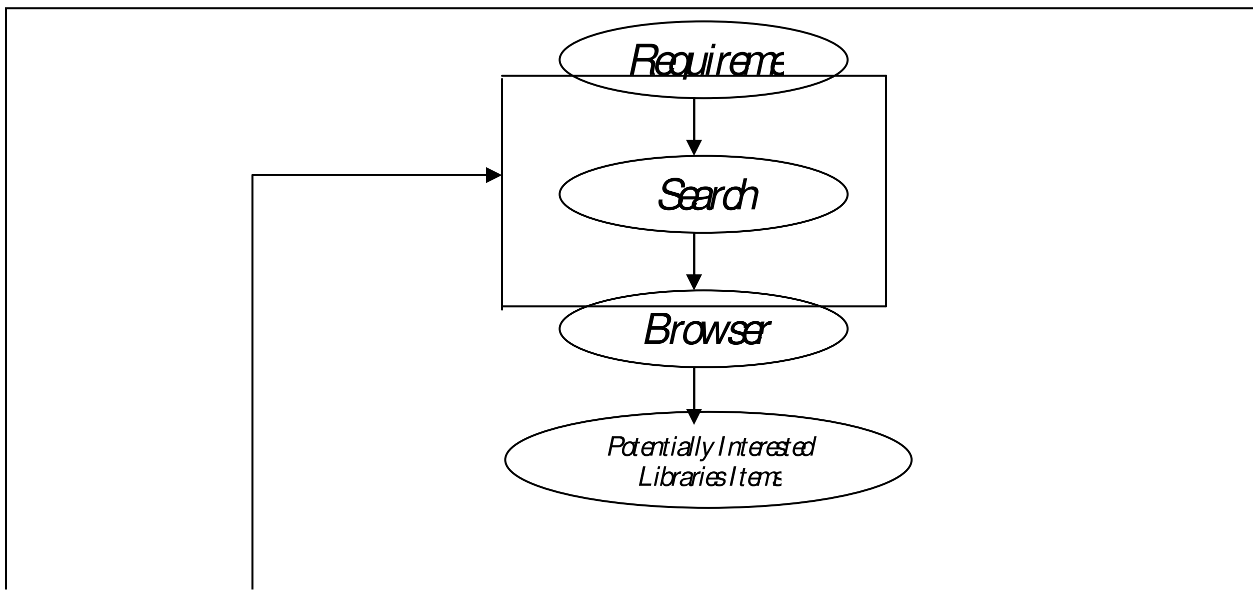
Neural networks provide a suitable framework for incrementally adapting the conceptual distance weight based upon user's relevance assessments. Single term indexing techniques are used to extract from both user's queries and software descriptions the keywords used by the learning algorithm of the artificial neural network in classification and retrieval activities. Neighborhood functions used by the learning algorithm are based on traditional measures of similarity of the vector space model.

2.5.7 Browsing

Browsing [22] is considered to be search where the goal is not well defined a priori because, for example, the goal is ultimately dependent on what is discovered during the search. Browsing is viewed as a user navigating through a graph where the nodes represent library items and the arcs connecting relationships. Thus for a library to be browseable it must be representable as a labeled graph.

Browsing consists of inspecting candidates for possible extraction, but without a predefined criterion. Main Operation is navigation, which determines in what order the components are visited at all. Browsing supports a bottom-up design approach: the library is first inspected and then the system is designed (i.e. composed) to take maximal advantage of the library.

Its main concern is thus recall: components should not be rejected unless they are absolutely irrelevant. Browsing usually works stepwise and denotes the set of all components, which are visible in one step as the focus. It does not require any search key but works on a pre-processed, usually hierarchical navigation structure which be computed from an indexing scheme. Figure 2.7 shows an abstract rendering of the normal browsing task.



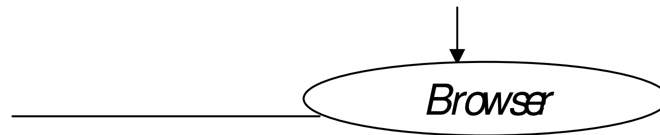


Figure 2.7 Browsing

2.3.8 Hypertext

A hypertext [8] represents a non-linear form of documents. Basic building blocks for hypertexts are links and nodes. Nodes are associated with a unit of information and can be of different types, depending on various criteria e.g. the class of data stored (plain text, graphics, audio, video, and executable program). Links represent a non-symmetric relation between nodes leading to a directed graph. Within nodes one can define anchors, which serve as source or target of links.

Users are able to move through a hypertext document by following links embedded in the document, an activity that is called browsing.

Hypertexts are well suited to represent the complex relationships between and within software components. An example is the javadoc tool, a program coming with the Java development environment. It allows generating automatically a hypertext document from the object structures and documentation found in java source code. The description is rather

simple, because it does not allow defining different views to the software components or a hierarchical structure other than the class' inheritance structure or object associations.

2.4 Conclusion

Current approaches to software classification and retrieval have been analyzed, principally on their advantages and limitations to cope with the retrieval problem. A taxonomy of more recent approaches has been explored: classification schemes using a controlled vocabulary, automatic searching and indexing, behavioral based approaches, neural network approaches, formal approaches and knowledge based approaches. A new proposed hybrid model for classifying candidate components has been considered in the next chapter.

Benefits both Case-base reasoning and Rough-fuzzy theory to further classify the candidate components have been exploited. The basic problem is to retrieve most similar candidate component for software reuse. A hybrid approach for classifying candidate components used in various previous projects to be reused in the current project has been tried because this reduces the domain search. Formal Specifications of the software components are stored in the library. Retrieval for the candidate

component is made using Structural matching where both signature and formal specification of the query component is matched with existing software components in the library. Classification of the candidate components is done using case-based reasoning and rough-fuzzy theory on the basis of various software features. The outcome is the rough-fuzzy membership values for reuse decisions for the current application in reference to the previous projects.

3.1 Problem Description

Primary problem encountered when reusing the existing component libraries is finding the most suitable candidate component from the library that can be reused in a specific

application. Consider a query for which matching components are retrieved using Structural Matching [24] and more than one says five components are retrieved. An important decision arises at this point is how to decide which candidate component is most suitable component to be reused. In the present work both case-based reasoning [24,29] and rough-fuzzy theory [30,31] is used as decision-making tools for the classification of the software candidate component. Some software features e.g. structural matching of the retrieved candidate components is identified and various previous similar cases are retrieved and rough-fuzzy values are assigned to them.

According to Dubois and Prade [25]:

> Fuzzy set theory is used for linguistic representation of patterns, thereby producing a granulation of the feature space. Rough set theory is used to obtain dependency rules which model informative regions in the granulated feature space. The fuzzy membership functions corresponding to the informative regions are stored as cases. Case retrieval is made using a fuzzy similarity function. Unlike existing case selection methods, the cases here are cluster granules, and not sample points. Also, the cases involve reduced number of relevant features with variable size. The algorithm is suitable for mining data sets, large both in dimension and size, due to its low time requirement in case generation as well as retrieval.

Thus proposed case-based system using rough-fuzzy hybridization helps in reducing the domain search and the use of Z specification further helps to find the most suitable match from the database.

3.2 Z Specification

The Z notation [32] is a model-oriented formal language developed by the Programming Research Group at Oxford University Computing Laboratory in the early 80's. Since then, Z has been used to specify a wide spectrum of the software systems including database systems, transaction systems, distributed computing systems, and operating systems [33]. The most notable success of Z is the specifications of CICS Application Programming Interface (API) by IBM United Kingdom Laboratories at Hursley Park [34].

Approximately 37,000 lines of codes were produced from Z specifications and designs, and it was reported that the code has approximately 2.5 times fewer problems than the

code that was not specified in Z.Z is a non-executable but strongly – typed specification language. ZTC is a type-checker for Z, which determines if there are syntactical and typing errors in Z specifications. There is no compiler for Z. However, there are tools to animate, or execute, subsets of Z.

Not all systems support 16-bit Unicode, the definitive representation of Z characters. A mark-up is a mapping to or from mapping to or from the Unicode representation. LATEX mark-up [35] based on 7-bit ASCII, which is suitable for processing by tools to render Z characters in their mathematical form. The mark-ups described show how to translate between a 'mark-up token', string of ASCII mark-up characters, into the corresponding string of Z characters. Remaining individual mark-up characters that do not form a special mark-up token, such as digits, Latin letters, and much punctuation, are converted directly to the corresponding Z characters, from ASCII-xy to Unicode U+00xy.

A LATEX command is a backlash '\' followed by a string of alphabetic characters, up to the first non-alphabetic character. At the moment only a limited subset of LATEX markup is used for implementation of proposed model.

Following is an example of segment of a Z specification:

The LATEX input for the above specification is given below:

\begin{spec}

```
\begin{schema}{Counter}
	ctr: \nat
\where
	0 ≤ ctr ≥ max
\end{schema}
\end{spec}
```

A complete set of markup for the Z notation is given in Appendix B.

3.3 Structural Matching

This includes all the approaches that find relevant component by using its structural characteristics.

Signature Matching: This scheme relies on the type matching under type-transformation. Consider two kinds of matching, function matching and module matching for software library components. The signature of the function is its type; the signature of module is a multi-set of user-defined types and multi-set of function signatures. An assumption is made for signature matching that set of allowable signatures is known.

Allowable signatures can be: simple types (i.e. integers, naturals), constructed types (i.e. record of other simple or constructed types), user defined types (i.e. domain-specific programmed types), type variables (i.e. type variable x may contain any other type).This type of matching is efficient where an application engineer knows the complete or partial signatures definition of the required software component.

Specification Matching: It is a process of identifying the behavioral relationship by checking the logical relationship between specifications of software components. The

process is based on matching the predicates in logic for a given theory of predicate equivalence. In specification matching, queries are formal requirement specifications (e.g. the specification of a signature) and the system retrieves relevant software from a library of formally specified components by invoking a theorem prover to determine if component specifications satisfy the requirements. Systems using this kind of matching, uses proof theorem algorithm to match the query with the component library specifications. It involves advance knowledge about specification languages.

Given a problem theory, **p**, a set of component theories, $S = \{c_0, \ldots, c_n\}$ and a satisfaction criteria $sat : c, p \rightarrow bool$ retrieval can be defined as searching S for some set C such that:

$$C = \{c : S \mid sat(c, p)\} \qquad (1)$$

(d) Satisfaction Criteria

Specification matching is a process of searching a solution space by verifying the satisfaction relationship, sat, and holds between component and problem requirement specifications. Figure

1 shows this lattice of satisfaction relationships ordered by logical containment. In Figure 3.1, I is a unary predicate defining input precondition over a component or problem domain. O is a binary predicate defining output postcondition over a component or problem domain and range. Quantifiers and parameters are excluded to ease readability.

Plug-in match is the strongest of the satisfaction relationships and requires that:

(i) Any legal problem input is a legal component input

(ii) Any legal component output is a legal problem output.

If a component and problem satisfy this condition, the component is guaranteed to solve the problem. However, the condition is too strong, as the second conjunct must hold even when the first does not. Even when the component is not required to process input, it must produce a correct output.

Zaremski and Wing provide a weaker condition, Weak Plug-in match that requires:

(i) Any legal problem input is a legal component input; and

(ii) Any legal component output resulting from a legal component input is a legal problem output.

This is weaker than Plug-in match is because the output must be legal only when a legal component input is processed. This is again too strong, as the second conjunct must hold for any component input, not just those that are legal problem inputs.

The set of components satisfying a problem specification is defined as

$$S_p = \{c : S \mid sat(c, p)\}$$

Where

Sat is defined as Satisfies match:

$$sat(c, p) \Leftrightarrow (I_p \Rightarrow I_c) \wedge (I_p \wedge O_c \Rightarrow O_p)$$

The retrieval problem is formulated as a search of S for elements of S_P. Because the search space is not ordered or otherwise structured, efficient deterministic search algorithms are not useful. Furthermore, the inference activity required for assessing satisfaction is impractical for component libraries of any reasonable size. To alleviate this problem, necessary conditions are used to prune the search space prior to application of the satisfaction condition.

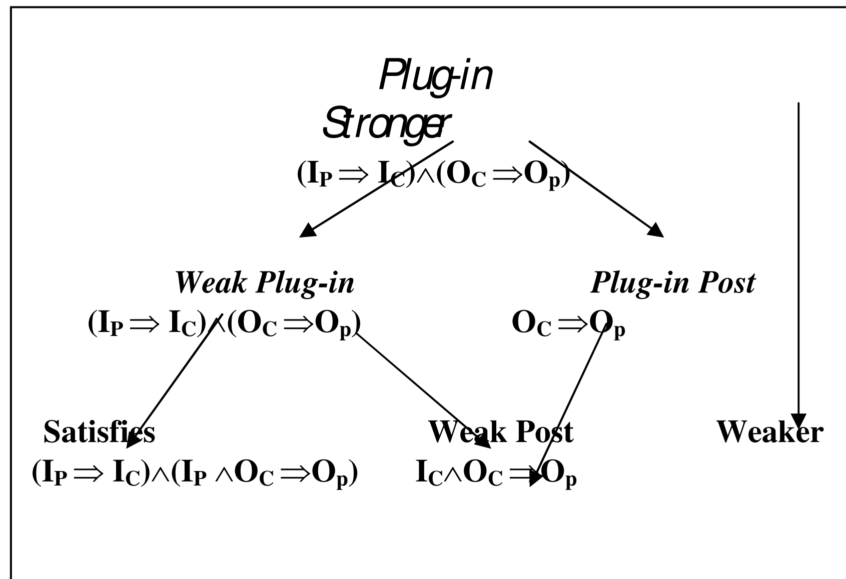


Figure 3.1: Lattice of Satisfaction Criteria

3.4 Case-Based Reasoning

Case-base reasoning (CBR) systems use various techniques to match a situation or a problem description (a case) to a database and known cases. The goal of case retrieval is to return case

that is the most similar to the input specification. The exact meaning and use of the term case varies from system to system. In general a case is a unique knowledge entity describing a problem and a solution. A case can be represented as a single database object or broken into two or more associated objects.

A typical case will have the following fields:

- ✓Title
- ✓Problem description
- ✓Cause (or justification)
- ✓Solution

The diagram shown in figure 3.2 shows a generic structure of a CBR knowledge representation. Usually the representation is

- ✓A problem points to one or more cases.
- ✓A case has a single solution.
- ✓A question can influence one or more cases.

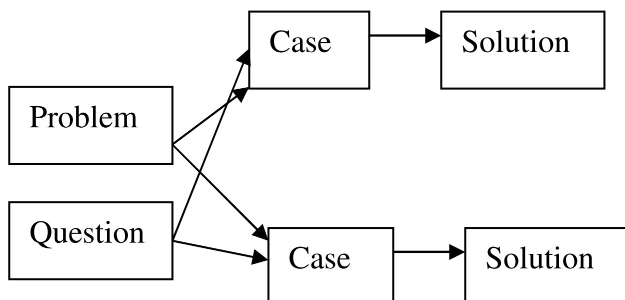


Figure 3.2: Generic structure of a CBR

3.5 Rough-Fuzzy Theory

A fuzzy set can be represented by a family of crisp sets using its first-level sets, whereas a

rough set can be represented by three crisp sets. Fuzzy Set theory assigns to each object degree of belongingness (membership) to represent an imprecise/vague concept. The focus of rough set theory is on the ambiguity caused by limited discernibility of objects. The rough-fuzzy sets are analyzed, with emphasis on their structures in terms of crisp sets.

A rough fuzzy set is a pair of fuzzy sets resulting from the approximation of a fuzzy set in a crisp approximation space. It is a pair of crisp set. It is a pair of fuzzy sets in which all elements in the same equivalence class have the same membership. The membership of an element is determined by the original memberships of those entire elements equivalent to that element.

Fundamental principle of a rough set-based learning system is to discover redundancies and *dependencies* between the given features of a data to be classified. Approximate a given concept both from below and from above, using lower and upper approximations. Rough set learning algorithms can be used to obtain rules in IF-THEN form from a decision table. Extract Knowledge from database (decision table w.r.t. objects and

attributes → remove undesirable attributes (knowledge discovery) → analyze data dependency → minimum subset of attributes (reducts)).

Let U denote a finite and non-empty set called the universe, and let $R \subseteq U \times U$ denote an equivalence relation on U, i.e., R is a reflexive, symmetric and transitive relation. If two elements x; y in U belong to the same equivalence class, i.e., xRy, we say that they are indistinguishable. The pair apr_R = (U; R) is called an approximation space. The equivalence relation R partitions the set U into disjoint subsets. It defines the quotient set U/R consisting of equivalence classes of R.

The equivalence class $[x]_R$ containing x plays dual roles. It is a subset of U if considered in relation to the universe and an element of U/R if considered in relation to the quotient set. The empty set and equivalent classes are called the elementary sets. The union of one or more elementary sets is called a composed set. The family of all composed sets is denoted by Com (apr). It is subalgebra of the Boolean algebra 2U formed by the power set of U.

Given an arbitrary set $A \subseteq U$, it may not be possible to describe A precisely in the approximation space apr_R = (U; R). Instead, one may only characterize A by a pair of lower and upper approximations. This leads to the concept of rough sets.

A rough set is interpreted by three ordinary sets:

Reference set: $A \subseteq U$,

Lower Approximation: $\underline{apr_R}(A) = \{x \varepsilon U \mid [x]_R \subseteq A\}$,
Upper Approximation: $\overline{apr_R}(A) = \{x \varepsilon U \mid [x]_R \cap A \neq \emptyset\}$

Let μ_A and μ_R denote the membership functions of A and R, respectively. The physical meaning of lower and upper approximations may be understood better by the following two expressions:

$\mu_{\underline{apr}R}(A)^{(x)} = \inf \{ 1 - \mu_R(x, y) \mid y \notin A\}$, Eq2
$\mu_{\overline{apr}R}(A)^{(x)} = \sup \{\mu_R(x, y) \mid y \in A\}$ Eq3

The rough-fuzzy membership function of a component x ε A for the fuzzy output class $apr_R \subseteq U$ is defined by

$l_{aprR}(x) = \frac{|F \cap A|}{|F|}$ Eq4

Where $F = [x]_R$ and $|A|$ means the cardinality of the fuzzy set A and is defined by

$|A| = \Sigma_{x \varepsilon A} \mu_A(x)$

The intersection operation is defined as

$\mu_{A \cap B} = \min \{\mu_A, \mu_R\}, \forall x \varepsilon A$

3.6 Proposed System for software component retrieval –ReuseMRC

It has explored that how reusable software components help to improve the quality and productivity of software. The use of formal specifications to specify the components removes the ambiguity and inconsistency from the specification. The concept of rough-fuzzy hybridization is used for case generation. In the present work Z notation has been used to specify the components and to add them in the library so that they can be retrieved from it. LaTeX- markup is used to specify components in Z notation.

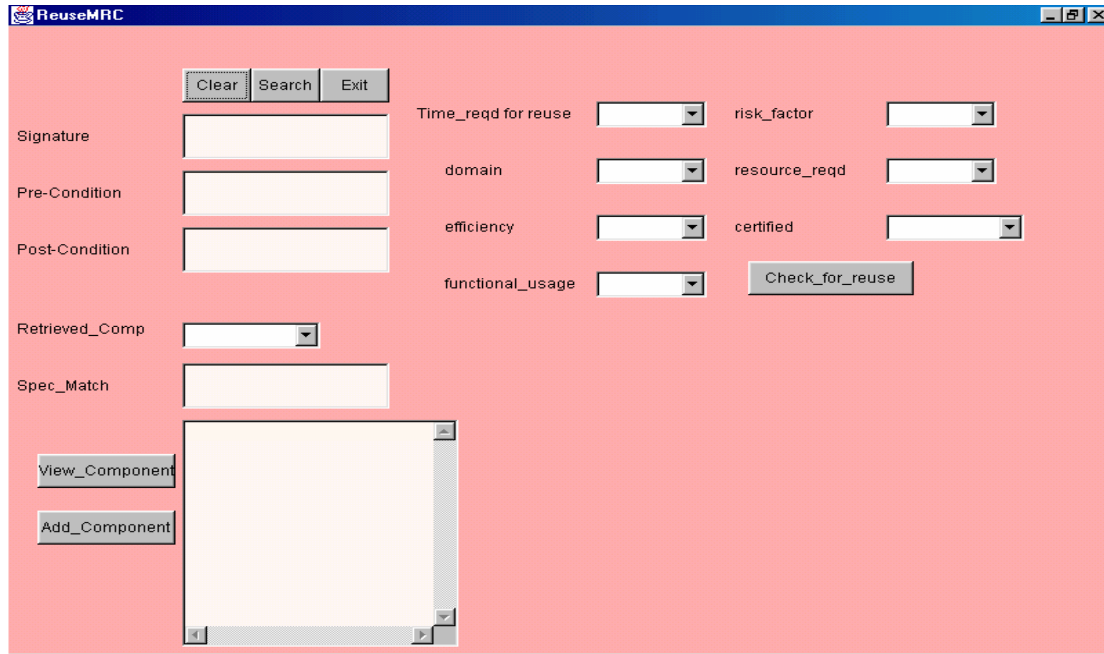


Figure 3.3 ReuseMRC Tool

3.6.1 Database of ReuseMRC tool

Our database consists of Z specifications of the components using LaTeX-markup language and case history of reused components in various applications along with the judgments of software developer regarding reuse of a component.

1.) specT – In this table there are total five fields named specid, schema name, signature, precondition and postcondition.

Field " schema name" contains the name of Z schema, field signature contains the signature of the Z component and predicate part of the Z specification is in stored the fields precondition and postcondition, which is in Z notation using the LATEX markup.

2.) caseT - In this table there are ten fields containing various software attributes along with the specid and app_id. Various software attributes contains the assigned fuzzy values regarding their weightage in the reuse decisions. Various software attributes are listed in the Table 3.1 .

3.) judgeT – This table consist six fields specid ,app_id and four reuse decisions. The table contains data related to the previous reuse judgement regarding a Z component.

3.6.2 Steps for Matching ,Retrieve & Classification:

The approach is a five-step process. The input given by the user is the query component whose specifications are written in Z Specification. The first step is to retrieve components from the software repository whose signature matches with input signature. Then taking components as our domain for search we find whether any

specification match is found for every

component in that domain.

The specification match is categorized as shown in figure3.1 and the membership values for fuzzy decision regarding specification matching are:

(i) If the Specification match found is Exact match, Plug-In, Plug-In Post then it is assigned HIGH membership value.

(ii) If the Specification match found is Guarded Plug-In, Guarded Post then it is assigned MEDIUM membership value.

(iii) Else Specification match is assigned LOW.

Hence, we get one of the eight attributes of software, which other than the query specifications has to be selected by the user.

Following are the eight attributes of the software and the further matching is done on their basis:

No	Attribute	Linguistic Variable
1.	**Specification**	**HIGH**(if above >75% specifications match)
		MEDIUM(if 35%-75% specifications match)
		LOW((if less<35% specifications match)
2.	**Efficiency**	**HIGH**(Defect Density <5 defects/KLOC)
		MEDIUM(Defect Density 5-10 defects/KLOC)
		LOW(Defect Density >10 defects/KLOC)
3.	**Functional Usage**	**HIGH**(Called >50 times)
		MEDIUM(Called 20-50 times)

		LOW(Called <20 times)
4.	**Risk Factor**	**HIGH**(Halts the System from working)
		MEDIUM(Degraded performance)
		LOW(Negligible effect on system functioning)
5.	**Domain**	**PL**(Product-line)
		BC(Business / Commercial)
		AF(Allied Field)
		MD(Miscellaneous domain)
6.	**Time_reqd for reuse**	**HIGH**(<4 person-weeks to adapt)
		MEDIUM(4-10 person-weeks to adapt)
		LOW(>10 person-weeks to adapt)
7.	**Resource Utilization**	**HIGH**(if >75% of available hardware ,software and memory is used)
		MEDIUM(if 30-75% of available hardware ,software and memory is used)
		LOW(if <30% of available hardware ,software and memory is used)
8.	**Certified**	**YES**(Component is certified)
		NO(Component not certified)

Table 3.1: Software attributes with their membership values and description.

After all the eight attributes to be considered for rough-fuzzy sets are obtained from the user, now the user has to decide on a past application(s) that best matches the present query application.

These candidate applications (obtained by the CBR search) are stored for reuse with decisions on the level of reuse w.r.t a specific component as:

(a) Reused as it is **(re-as)**(l_{R1})

(b) Reused adapting the component(**re-adapt)** (with minor code modification) (l_{R2})

(c) Reused adapting the specification**(re-specs)** (with design modifications) (l_{R3})

(d) Reused developing afresh**(re-new)** (l_{R4})

Level of Reuse	Fuzzy Membership Value
(re-as)(l_{R1})	0.9
(re-adapt)(l_{R2})	0.3

(re-specs)(l_{R3})	0.2
(re-new)(l_{R4})	0.1

Table 3.2 Assigned fuzzy membership values to level of reuse

In the CBR table as shown in Table 3.6, the roughness occurs because of different decisions on the level of reuse for two similar applications whereas fuzziness in the decision occurs due to overlapping, ill-defined boundaries. With a reuse history of a component in the case-base, using rough-fuzzy sets we compute the lower and upper membership values of similar applications. Finally the rough-fuzzy membership function is used to compute the

membership value of a component’s application for decisions.

The flow chart describing the process of matching, retrieving the software components and classifying as per their reuse decisions in various software applications are shown in the figure 3.4.

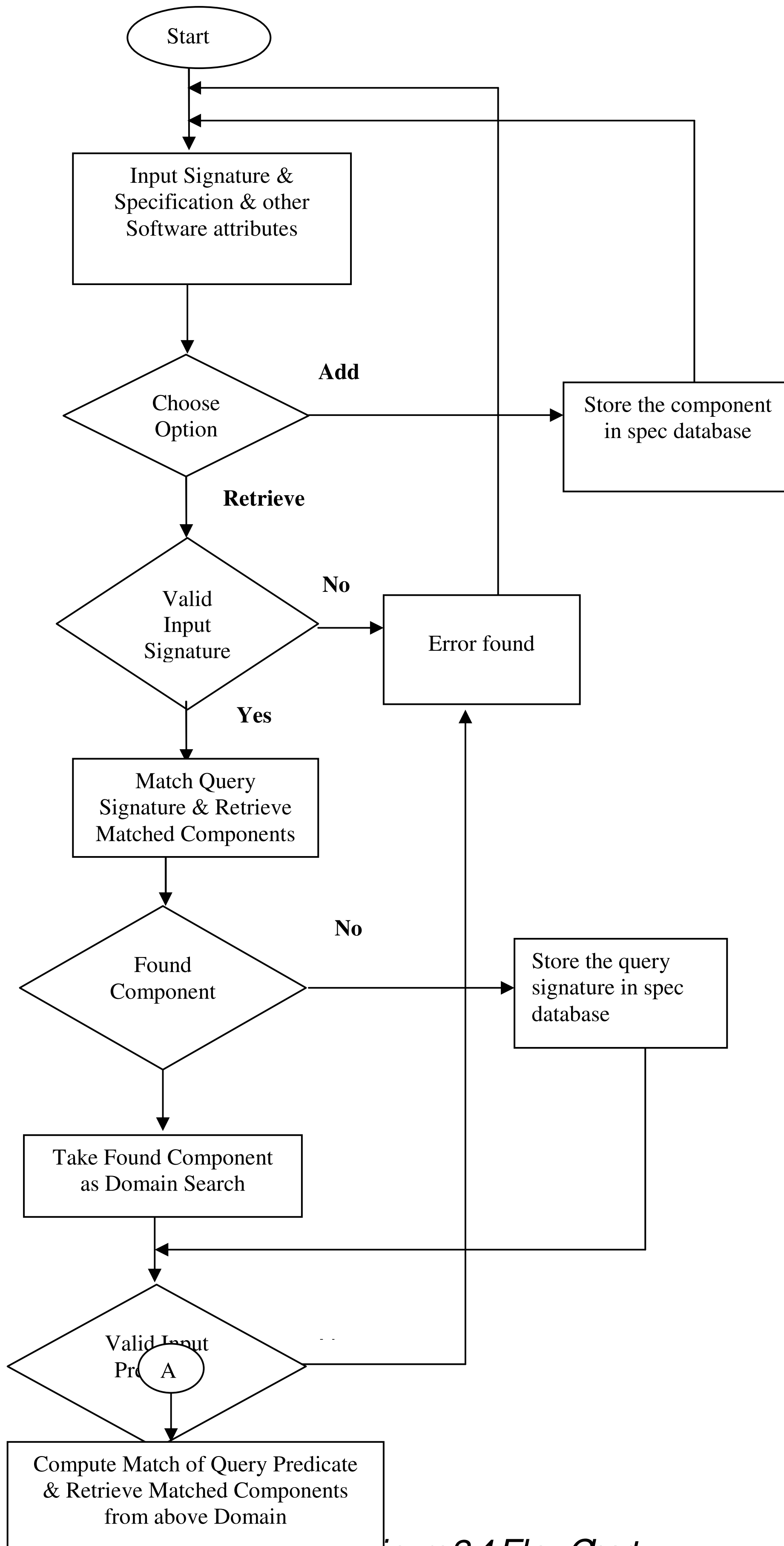
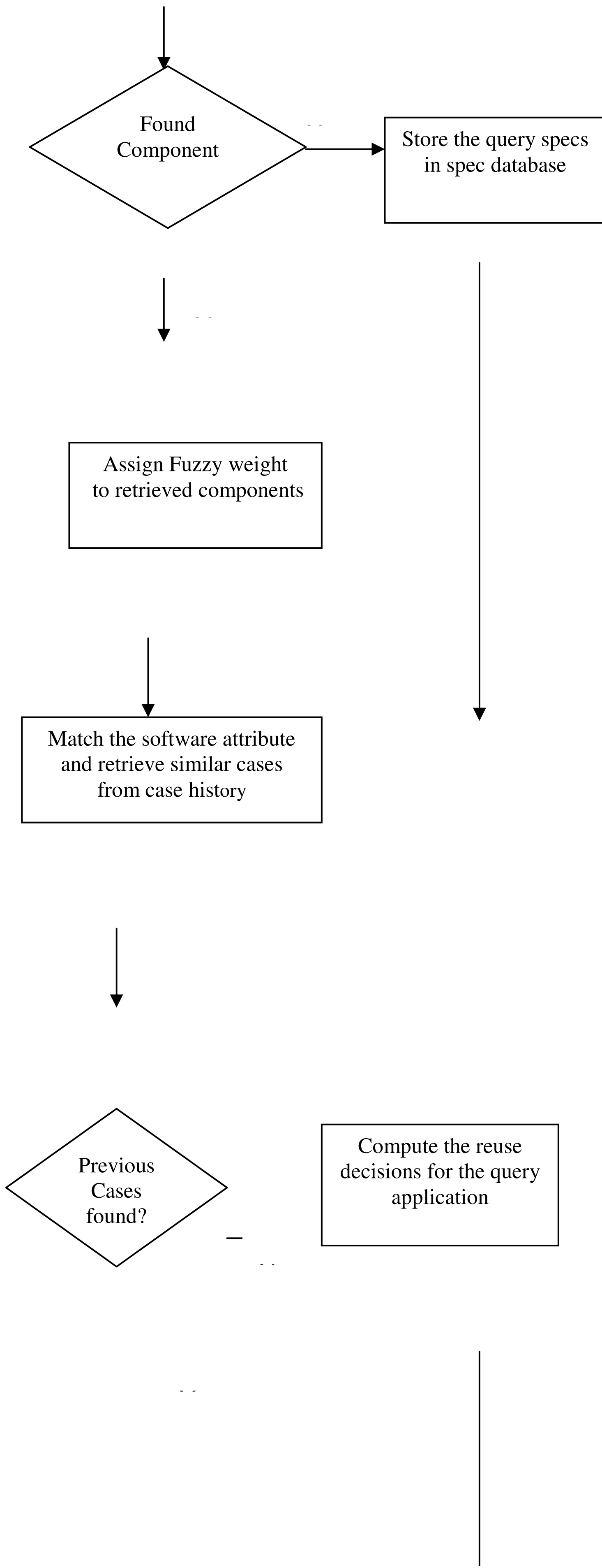


Figure 3.4 Flow Chart


Found
Component
Store the query specs
in spec database
Assign Fuzzy weight
to retrieved components
Match the software attribute
and retrieve similar cases
from case history
Previous
Cases
found?
Compute the reuse
decisions for the query
application

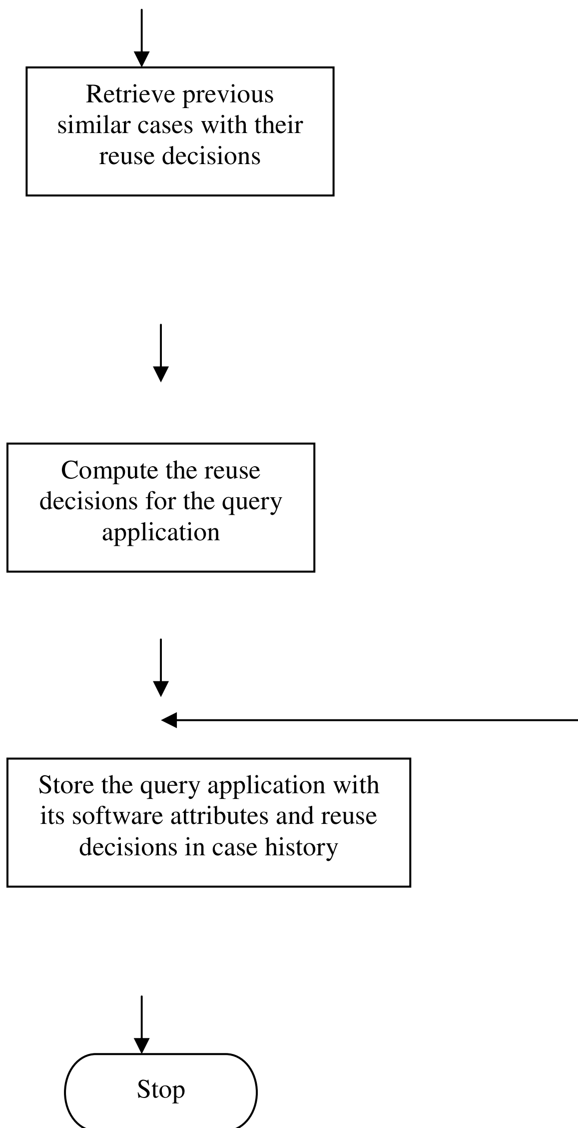


Figure 3.4(contd.)

3.6.3 Case Study

Consider a software repository consisting 50 components whose specifications are written in Z notations using LATEX markup. Given a query component Q1 to be used in N1 (file cache manager) application is to be searched in the above repository using ReuseMRC as shown in Fig 3.6, where Q1 is a component representing a delete function which deletes a given number of files from the filecache. The specifications required by the user are shown in Table 3.3

Signature	Integer implies Integer
Precondition	Size (%fcache*) >= %d?
Postcondition	Size (%fcache2') = Size (%fcache1*) - %d?

Table 3.3 Query Component Specification

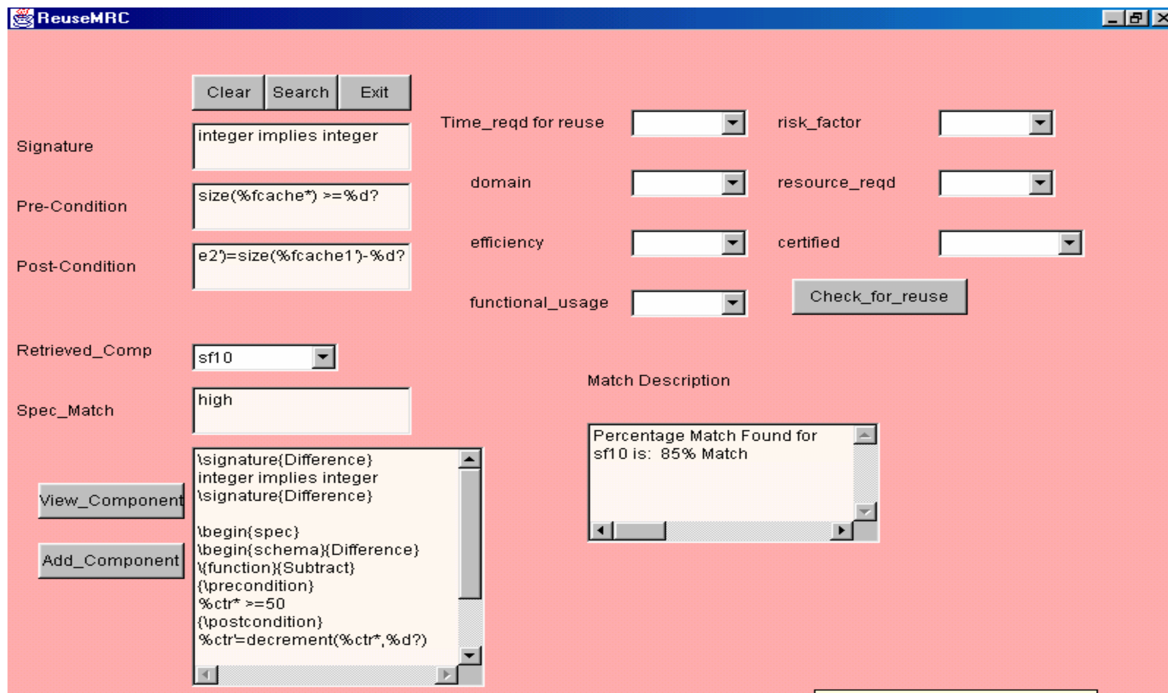


Figure 3.5 Initial search results

To begin with search, Structural matching (signature and specification) on Q1 is performed and three (sf9, sf10, sf11) components out of 50 components are retrieved along with their specification match as shown in Table 3.4, and Figure 3.6,3.7,3.8.

```
\signature{Counter}
integer implies integer
\signature

\begin{spec}
\begin{schema}{Counter}
{\function}{Decrement}
{\postcondition}
%ctr'=decrement(%ctr*,%d?)
\end{schema}{Counter}
\end{spec}
```

Figure 3.6 "sf9" Specifications

```
\signature{Difference}
integer implies integer
\signature

\begin{spec}
\begin{schema}{Difference}
{\function}{Subtract}
{\precondition}
%ctr'>=50
{\postcondition}
%ctr'=decrement(%ctr*,%d?)
\end{schema}{Difference}
\end{spec}
```

Figure 3.7 "sf10" Specifications

```
\signature{Counter}
integer implies integer
\signature

\begin{spec}
\begin{schema}{Counter}
{\function}{Decrement}
{\precondition}
%size(ctr')>=%d?
{\postcondition}
%ctr'=decrement(%ctr*,%d?)
```

Figure3.8 sf11 Specifications

Comp No	Weight Assigned	Percentage Match
Sf9	Medium	70%
Sf10	High	85%
Sf11	High	95%

Table 3.4: Query Results after Structural Matching

Taking every retrieved component separately, the level of reuse is computed. The software attributes as shown in figure 3.9 are filled by the user for further refining the search and to find the way these components are used in previous projects. Table 3.5: Candidates Applications retrieved from CBR which reuses component sf10 and whose same spec_match weight assigned as N1.

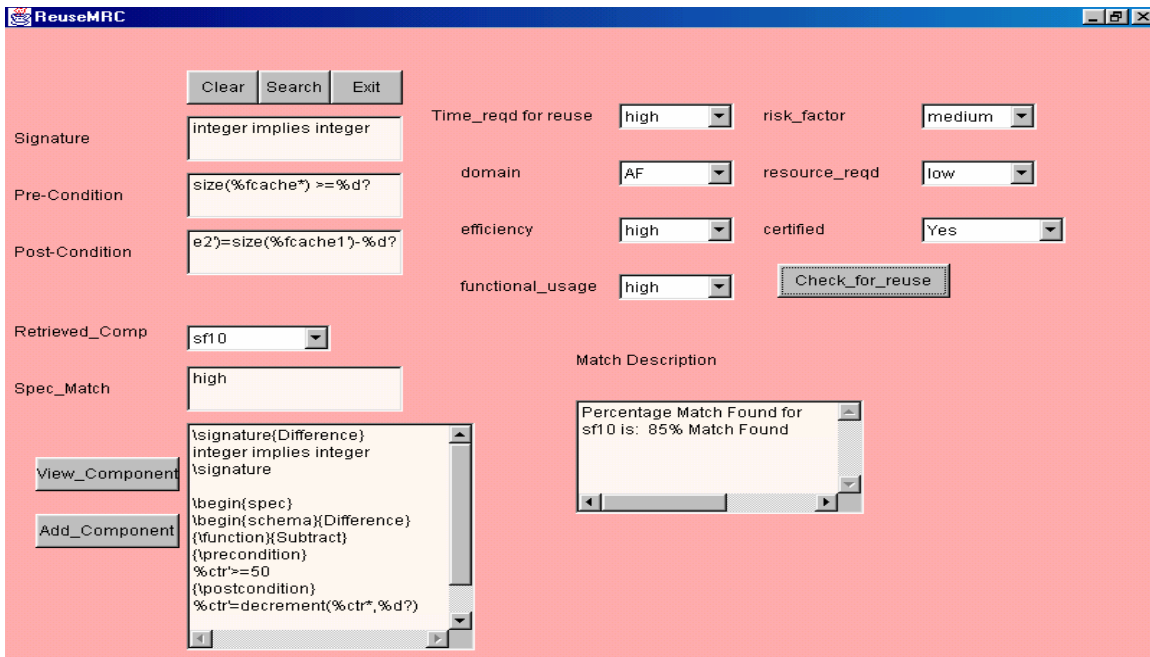


Figure 3.9 Software Attributes for the query application (N1)

Various software attributes for sf10 are enumerated as

N1: *high, high, AF, high, medium, low, yes*

There are about 70 applications, which have used the components of our software repository either as-it is, or with some changes. There are four applications which are using sf10 with three (A2, A6, and A9) having all software attributes same as N1.

Software Attributes	A2	A6	A7	A9	N1
Time_reqd_for_reuse	High	High	Low	High	High
Domain	AF	AF	AF	AF	AF
Efficiency	High	High	Medium	High	High
Functional_usage	High	High	Low	High	High
Risk_Factor	Medium	Medium	Low	Medium	Medium
Resource_Reqd	Low	Low	Low	Low	Low
Certified	Yes	Yes	No	Yes	Yes

Table 3.5: Candidates Application retrieved from CBR

Appl	Re-As(R_1)	Re-Adapt(R_2)	Re-Specs(R_3)	Re-New(R_4)	Actual Decision
A2	0.3	0.9	0.3	0.1	**R_2**
A6	0.8	0.3	0.2	0.1	**R_1**
A9	0.3	0.9	0.3	0.1	**R_2**

Table 3.6: Judgement table containing judgement of a user regarding reuse of a component

For the decision (R_1), the membership values of the equivalence class {A2, A6, and A9}

are calculated by using eq2, eq3 and judgement table shown in Table 3.6

$^{\mu\underline{R}}(R_1)^{([X]_R)} = \inf\{^{\mu}(R_1)\ (A2,A3,A9)\}$ $\quad$ $^{\mu R}(R_1)^{([X]_R)} = \sup\{^{\mu}(R_1)\ (A2, A3)\}$

$= \inf\{0.3,0.8,0.3) = 0.3$ $\quad$ $= \sup\{0.3,0.8,0.3\}=0.8$

Similarly for R_2, R3 and R_4

$^{\mu\underline{R}}(R_2)^{([X]_R)} = 0.3$ $\quad$ $^{\mu\underline{R}}(R_3)^{([X]_R)} = 0.2$ $\quad$ $^{\mu\underline{R}}(R_4)^{([X]_R)} = 0.1$

$^{\mu R}(R_2)^{([X]_R)} = 0.9$ $\quad$ $^{\mu R}(R_3)^{([X]_R)} = 0.3$ $\quad$ $^{\mu R}(R_4)^{([X]_R)} = 0.1$

Using the eq4, compute the membership value of a component's application for decisions

$R_{1,}$ R2, R3 and $R_{4.}$

A_{R1} = {(A2, 0.3)(A1, 0.8)}

$$l_{Dc}(A2,A3) = \frac{|\{A2,A3\} \cap A_{R1}|}{|2|}$$

$$= \frac{\Sigma_{x\varepsilon X} \min\{\mu_{A2,A3}(x), \mu A_{R1}\}}{2} = (0.8 + 0.3)/2 = 0.55$$

In similar way, the other values are calculated. These are summarized in Table 3.8 and Figure 3.10,which shows the final results.

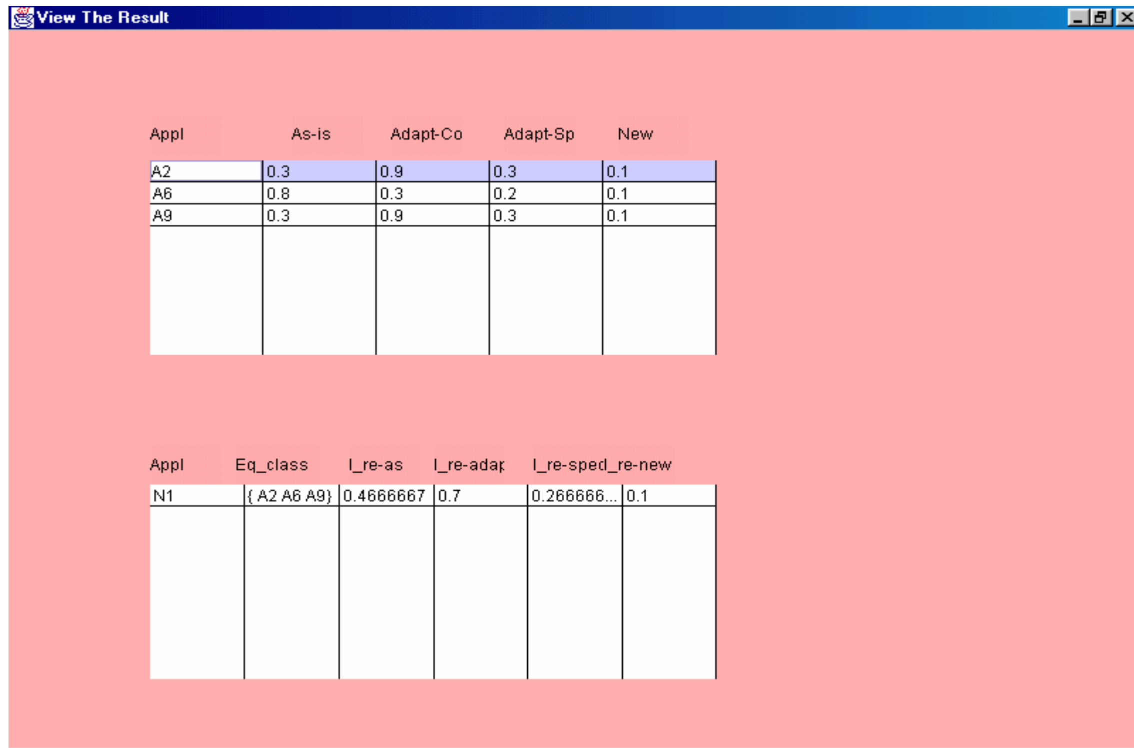


Figure 3.10Final Search Results

No	Equivalence Class	(l_re-as) l_{R1}	(l_re-adapt) l_{R2}	(l_re-spec) l_{R3}	(l_re-new) l_{R4}
N1	{A2,A6,A9}	0.46	0.7	0.26	0.1

Table 3.7 Rough-fuzzy membership values for decisions

The attributes of the query application N1 are matched with previous applications. The software attributes of N1 matches with the equivalence class {A2, A3}. Table 4 shows the membership values for the reuse decisions R_1, R_2, R_3, R_4. These values indicate that query application N1's requirements are similar to applications A2 and A3 and they can be reused as **re-as** (R_1) with membership value 0.55; and **re-adapt** (R_2) with membership value 0.6; **re-specs** (R_3) with membership value 0.3; **re-new** (R_4) with membership value 0.1. Similarly Reuse level can be computed for sf9 and sf11 and then finally can be compared to choose the most suitable component for the reuse.

CHAPTER 4

Conclusions & Future Scope

This model is implemented using formal specifications of a component and its signature as input. Signature of the component is used to retrieve the matched components and serve them as domain search. Formal specifications are further used to retrieve the appropriate components. The search is further refined using rough-fuzzy hybridization.

4.1 Conclusions

The model proposed in current work provide decision-making tool that provides software developers with a method to decide the most appropriate component from the retrieved candidate components, which can be most suitable for reuse. This model mainly focuses on domain search, as the domain search reduces more and more refined form of results are obtained, in other words, chances of finding exact match increases.

Some important results are shown below: -

- ✓ Figure 4.1 shows reuse effectiveness of the proposed model. In the situations where abundance of component is given, high recall is not important. Nowadays software is available via the Internet and a very large number of components, which are, developed somewhere in the world offer similar functionality. In such situations

precision becomes much more important than recall. If an application engineer retrieves a small number of high quality components, which fit his needs, she/he does not bother much about components missed in the search. Similarly, the proposed model is mainly applicable in above situations where abundance of component is given, so here also precision becomes more important than recall.

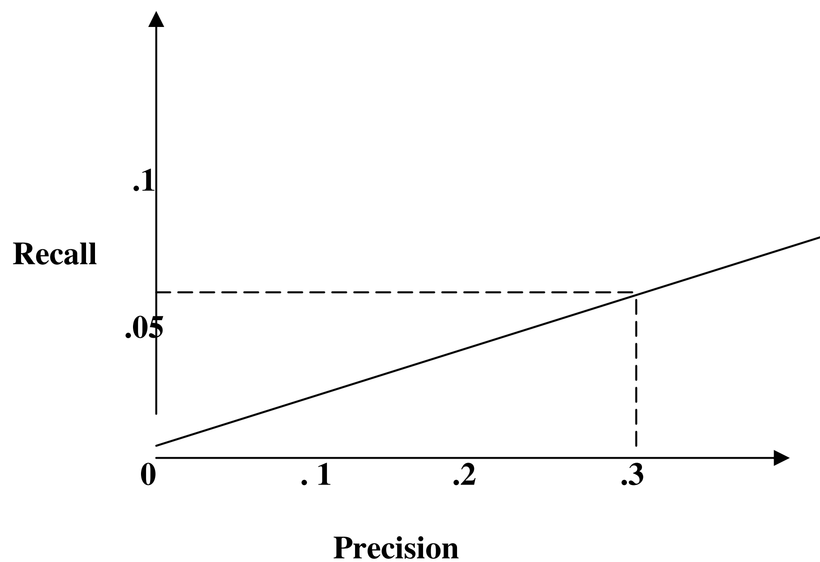


Figure 4.1 Measures of Reuse Effectiveness

- ✓ Another important result is that the use of rough-fuzzy sets increased the probability

  Of finding the suitable components for reuse when exact matches are not available or very few in number.

4.1.1 Some Assumptions and Constraints of Proposed Model

Reuse Decisions under special situations

1. If no candidate component is found after performing signature matching then query component (Q1) is added as a new component in software repository. The query application (N1) using this new added component is given the reuse decision of **re-as** (l_{R1}) and stored in the CBR information repository.

2. If no candidate component is found after performing specification matching then query component (Q1) is added as a new component in software repository. The query application (N1) using this new added component is given

the reuse decision of **re-as** (l_{R1}) and stored in the CBR information repository.

3. If candidate component is yet not reused by any application then query application (N1) using this new added component is given the reuse decision of **re-as** (l_{R1}) and stored in the CBR information repository.

4.2 Future Scope

A major concern is to extend the work for more accuracy and optimization in the component retrieval process.

Further work can be explored in the following directions: -

- ✓Learning algorithms can be implemented so that a query, which has been once asked, should give the reuse decision only for best match.
- ✓Type Checker for Z notation like ZTC should be incorporated in the tool for checking the entered specification before processing it.
- ✓At present, the ReuseMRC system has just implemented for the subset of the Z notation. It can be extended and implemented for every Z markup.

✓A Z specification editor can be added in the tool for the ease of users.

Appendix - A
Glossary of Z Notation

Names

a,b	**identifiers**
d,e	**declarations (e.g.,** a: A; b, ...: B ...**)**
f,g	**functions**
m,n	**numbers**
p,q	**predicates**
s,t	**sequences**
x,y	**expressions**
A,B	**sets**
C,D	**bags**
Q,R	**relations**
S,T	**schemas**
X	**schema text (e.g.,** d, d | p**, or** S**)**

Definitions

a == x	**Abbreviation definition**
a ::= b | ...	**Free type definition**
[a]	**Introduction of a given set (or** [a,...]**)**
a _	**Prefix operator**
_ a	**Postfix operator**
_ a _	**Infix operator**

Logic

true	**Logical true constant**
false	**Logical false constant**
$\neg p$	**Logical negation, *not***
$p \land q$	**Logical conjunction, *and***
$p \lor q$	**Logical disjunction, *or***
$p \Rightarrow q$	**Logical implication**
$p \Leftrightarrow q$	**Logical equivalence**
$\forall X \bullet q$	**Universal quantification**
$\exists X \bullet q$	**Existential quantification**
(**let** a == x; ... $\bullet$p)	**Local definition**

Sets and expressions

$x = y$	**Equality**
$x \neq y$	**Inequality**
$x \in A$	**Set membership**
$x \notin A$	**Non-membership**
$\varnothing$	**Empty set**
$A \subseteq B$	**Set inclusion**
$A \subset B$	**Strict set inclusion**
$\{ x, y, ... \}$	**Set display**
$\{ X \bullet x \}$	**Set comprehension**
$(\lambda X \bullet x)$	**Lambda expression**
(**let** a == x; ... $\bullet$y)	**Local definition**
if p **then** x **else** y	**Conditional expression**
(x,y, ...)	**Tuple**
(x,y)	**Pair**
$A \times B \times ...$	**Cartesian product**
$\mathbb{P} A$	**Power set**
$A \cap B$	**Set intersection**
$A \cup B$	**Set union**
$A \setminus B$	**Set difference**
first x	**First element of an ordered pair**
second x	**Second element of an ordered pair**
# A	**Number of elements in a set**

Relations

$A \leftrightarrow B$	**Binary relation (**$\mathbb{P}(A \times B)$**)**
$a \mapsto b$	**Maplet (**(a,b)**)**
dom R	**Domain of a relation**
ran R	**Range of a relation**
$Q \fatsemi R$	**Forward relational composition**
$Q \circ R$	**Backward relational composition (**$R \fatsemi Q$**)**
$A \lhd R$	**Domain restriction**
$A ⩤ R$	**Domain anti-restriction**
$A \rhd R$	**Range restriction**
$A ⩥ R$	**Range anti-restriction**
$R ⦇A⦈$	**Relational image**
$R^{\sim}$	**Inverse of relation**

R^+	**Transitive closure**
$Q \oplus R$	**Relational overriding**
a $\underline{R}$ b	**Infix relation**

Functions

$A \rightarrow\!\!\!\!\!\!+ B$	**Partial functions**
$A \rightarrow B$	**Total functions**
$A \rightarrowtail\!\!\!\!\!\!+ B$	**Partial injections**
$A \rightarrowtail B$	**Total injections**
$A \rightarrowtail\!\!\!\!\!\rightarrow B$	**Bijections**
f x	**Function application (or** f(x)**)**

Numbers

$\mathbb{Z}$	**Set of integers**
$\mathbb{N}$	**Set of natural numbers** { 0, 1, 2, ... }
$\mathbb{N}_1$	**Set of strictly positive numbers** { 1, 2, ... }
m+n	**Addition**
m-n	**Subtraction**
m*n	**Multiplication**
m **div** n	**Division**
m **mod** n	**Remainder (modulus)**
$m \leq n$	**Less than or equal**
m < n	**Less than**
$m \geq n$	**Greater than or equal**
m > n	**Greater than**
m .. n	**Number range**
min A	**Minimum of a set of numbers**
max A	**Maximum of a set of numbers**

Sequences

seq A	**Set of finite sequences**
seq_1 A	**Set of non-empty finite sequences**
iseq A	**Set of finite injective sequences**
$\langle\rangle$	**Empty sequence**
$\langle$x, y, ... $\rangle$	**Sequence display**
$s \frown t$	**Sequence concatenation**
head s	**First element of a sequence**
tail s	**All but the head element of a sequence**
last s	**Last element of a sequence**
front s	**All but the last element of a sequence**
s **in** t	**Sequence segment relation**

Schema Calculus

$S \widehat{=} [\ X\ ]$	**Horizontal schema**
[T; ... | ...]	**Schema inclusion**
z.a	**Component selection (given** z:S**)**
θS	**Binding**
$\neg S$	**Schema negation**
$S \wedge T$	**Schema conjunction**

$S \vee T$	**Schema disjunction**
$S \fatsemi T$	**Schema composition**
$S >> T$	**Schema piping**

Conventions

a?	**Input to an operation**
a!	**Output from an operation**
a	**State component before an operation**
a’	**State component after an operation**
S	**State schema before an operation**
S’	**State schema after an operation**
ΔS	**Change of state**
ΞS	**No change of state**

List of Papers

1. Damandeep Kaur, “ Comparison of Various Component Retrieval Schemes”, Published on page 136-138 in National Conference of Software Engineering on Principle and Practices ,held at T.I.E.T, Patiala,on 5th-6th March 2004.
2. Damandeep Kaur, Rajesh K.Bhatia,” Various Computational Models for Information Retrieval “, Communicated in International Conference on Coginitive Systems, to be held at Delhi in Nov 2004.