CS-94-10 October 3, 1994

Organizing Reusable Software Repositories Through Heuristic Clustering

by

Khuzaima S. Daudjee

A thesis submitted to the
Faculty of Graduate Studies, York
University in partial fulfillment of the
requirements for the degree of

Master of Science

Thesis Supervisor: Prof. A. A. Toptsis

Graduate Programme in Computer Science
Department of Computer Science, York University
4700 Keele Street, North York, Ontario
CANADA M3J 1P3

(October 1994)

Organizing Reusable Software Repositories through Heuristic Clustering

by

Khuzaima S. Daudjee

Submitted to the Faculty of Graduate Studies on
October 3, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

As software reuse becomes more prominent and accepted in industry, systems and tools for software reuse become a key aspect in achieving successful reuse of software artifacts. A major problem with such tools is the classification and retrieval of the software components. In order to search for and retrieve the conceptually closest software component from a software repository, components need to be classified in some manner. We address this problem by proposing two heuristic clustering schemes to organize software repositories. We contend that our proposed schemes automatically organize repositories containing descriptions of software components, and that this organization easily supports a retrieval method for the software components to be reused. Our clustering schemes classify components that have been represented using a knowledge representation-based language, and eliminate the need for manual effort for classification when the repository's contents are changed. Our proposed methods are tested on a small, but realistic, software collection. The experiments indicate that while both methods attain satisfactory performance with respect to the number of retrieved components that are relevant, the second method presented in this thesis performs very satisfactorily in terms of the measure of relevant components retrieved, and the proportion of retrieved components that are relevant.

Key Words: software reuse, software repositories, heuristic clustering, repository organization.

Thesis Supervisor: ***Prof. Anestis A. Toptsis***

Acknowledgement

First of all, my gratitude goes to my thesis supervisor Prof. Anestis Toptsis who first introduced to me the area of software reuse, for his ideas, his insight, his guidance, his patience, and for the financial support he provided me for the duration of this thesis. He also read many drafts of the thesis, and always provided valuable suggestions that greatly improved the content and presentation of this work. I'll always remember our interesting discussions after our meetings! I would also like to thank Prof. Jia Xu for providing constructive feedback that also improved the presentation of the thesis, and for serving on the examining committee. Sincere thanks also go to Profs. Minas Spetsakis and George Tourlakis for serving on the examining committee. I will always be indebted to Prof. Michael Jenkin for making me feel "at home" during my first few months at York, and for his advice throughout my stay here. Prof. Tim Brecht has been both, a friend and a faculty member to me. His support in terms of academic and non-academic matters has been of substantial help. While at York, I don't know what I would have done without the constant help and support of Patricia Plummer. She always made the most complicated regulations look simple, and was always there to help when the need arose. My stay at York would not have been as pleasant had it not been for the friends I made. It was great to know Kaushik, Ragab, Arun, Nian, Ken, Choi, Yahya, Ahmad, Jason, Neal, Carl, Song, Bernie, Henry and all other fellow graduate students who made learning at York more fun. Special thanks go to Ragab for his constant help with many of the intricacies of LaTeX, Unix, etc. I can never forget the love, upbringing and the sacrifices that my parents have made for me throughout my life. They taught me much about life, and about the value of love, trust and respect. My brother Munib was instrumental in guiding me through the 'critical years' of my life; I couldn't have asked for a better role model. Finally, I would like to thank my wife Lisa for her love, understanding, unwavering patience, support, friendship, encouragement, and for always being there for me. Without her, this thesis would not have been possible. Thus, it is to her that I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is of growing importance for corporations to have effective reuse of software artifacts as they invest in developing and maintaining large software systems [29].

Software reuse is the process of developing software for a new system by using the software from other systems, thereby lessening or even avoiding the need for developing new software from scratch [12].

It would be cost-effective, especially for large software R&D corporations, if software artifacts were to be reused. Recent studies estimate that a $30 dividend is paid for a $1 invested on software reuse over a four-year period [2]. Although there have been several success stories for software reuse [5], [14], [11], [30], [29], there is a large number of technical, managerial, and legal issues to be resolved before software reuse becomes effective. The key technical issues are in the areas of domain analysis, classification of software components, interoperability of software repositories, adaptation of software components, reuse of system designs and architectures, and software metrics that quantify eligibility for reuse [20]. Reportedly, no standards exist for any of these areas.

It is almost always impossible to find an existing software component that will exactly meet some of the requirements of the new software system. However, it is quite possible to find a software component that meets, at least partially, some of the functionality required of the new system.

In this case, there is always a need to modify the existing component so that it can be reused in the system to be developed. Note that this could be any artifact from the software lifecycle. However, in order to find the closest component to meet

the software developer's needs, one has to search for such a component, knowing well that this could be an expensive and exhaustive search, especially if there are hundreds of components that are potential candidates to be reused. In large-scale reuse, this search for components can take a substantial amount of computing time and resources. A key part of almost all software reuse systems is the repository - also called the library- which is used to store software artifacts in various forms (or descriptions). This is where classification of software components plays a key part. If software components are classified (or organized) within the software repository in some manner, then this will greatly aid the reuser in searching and retrieving the conceptually closest component to the query.

There have been a number of industrial projects to reuse code [11] [30] [29] and some for reusing other products from the software lifecycle, such as design artifacts [5]. As long as a sufficient number of components are provided, a system can be built that, given a description of a component in some form, will search and retrieve "similar" components from the repository. By similar, we mean those components that provide, to varying extents, at least some of the functionalities of the desired software component. Similarity in this context is measured using a model that provides for a uniform method of computing similarity (in quantifiable terms) between software components.

In this thesis, we propose two techniques for organizing a repository of software components. We model the software reuse environment as set out in [10] [28] [7], where the librarian has the central task of constructing functional descriptions from incoming software modules being delivered by application developers, as well as organizing the library. The retrieval tool is available for use to all reusers. The librarian constructs functional descriptions using some off-the-shelf knowledge representation language. The reuse environment model is shown in Fig. 1.1 below. The domain of the librarian is shown by the oval in Figure 1.1. This also outlines the librarian's basic tasks, especially in an environment where inter-company reuse is taking place. Note that while the application developers interact with the librarian by delivering components as well as receiving software reuse related knowledge (e.g. number of domains for which reusable components exist), the retrieval tool is available to all reusers (systems analysts, application developers and other software engineering professionals).

The organization of the repository is done automatically, and its structure is transparent to the prospective software reuser. Also, our scheme handles, by default, automatic and transparent repository reorganization in the event of insertion of new components, and deletion or modification of existing ones. The effectiveness of the

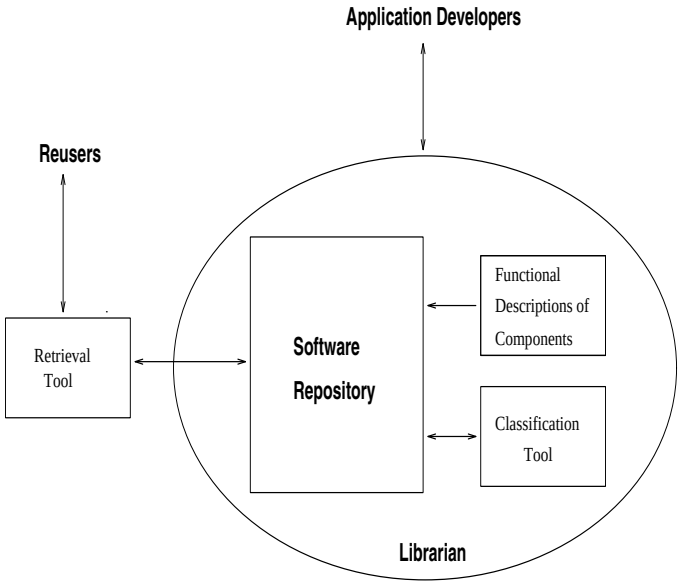

Figure 1.1: Model of Reuse Environment

proposed schemes is quantified by testing our developed prototype for the software component retrieval process on a software collection in the domain of data structures. The tests illustrate that our proposed work certainly aids the software reuse process, and in the case of our second repository organization scheme, very encouraging values are attained for the proportion of relevant components retrieved, and the proportion of retrieved components which are relevant.

This thesis is organized as follows. Chapter 2 describes previous solutions to the software repository organization problem. Chapter 3 outlines the details of the similarity computation method. Chapter 4 describes our schemes. Chapter 5 gives the performance evaluation of the schemes. Chapter 6 contains a comparison of our work versus the previous related work. Chapter 7 concludes our work and discusses future research directions.

Chapter 2

Related Work

A number of software repository organization methods have been reported. The methods outlined below provide an overall coverage in terms of the different approaches and techniques.

2.1 The Faceted Approach

Prieto-Diaz proposed a classification scheme that uses facets [23] (as in library science). This scheme uses synthesis (as opposed to breakdown) of basic classes and a group of these basic classes make up a facet. A set of functional specifications is provided by the reuser, and the reuser then searches a library of available components to find candidates that satisfy the specification. The software component being searched for is represented using a tuple of terms that represent each facet. In the case of similar terms, these synonyms are grouped together and form part of the thesaurus. The thesaurus is used to control the keyword vocabulary. The faceted classification scheme was considered to represent the model of a programmer's knowledge of software. Thus, it could also serve to be a good model for building user interfaces. The facets in [23] were implemented in a relational database system, comprising tables containing some test terms for each facet.

An important part of the classification scheme is the use of a conceptual graph which measures closeness between terms in any one facet. Since obtaining the conceptually closest component is a matter of efficient search and retrieval, the classification of components is key to the retrieval problem. In a conceptual graph, all possible terms that belong to a facet are interconnected using weighted edges. The weights

are representative of how conceptually close (or far) is one term from another.

The order of the facets is fixed; the order provides syntactic representations of the test components which take the place of sentential structures. The fixed order of facets in which the query has to be made for retrieval is a restriction to the scheme's flexibility. Also, assigning or selecting more than one term in each facet cannot be done [22].

Another disadvantage is that the construction of the conceptual graph is a highly time-consuming process. The maintenance of such a graph as more terms or features are added can be cumbersome since it will demand a restructuring of the whole graph.

2.2 The Natural Language Approach

Maarek et al [15] focus on building a library consisting of *indices* of objects that are contained in it. An *index* in this context is a characterizing attribute, or profile, for each object. Since it is important to include functionality information in indices, and this information is hard to obtain, their scheme relies on analyzing the natural language documentation. This documentation comes in the form of Unix-like man pages, and thus, their method is dependent on documentation available in this form.

Maarek et al are of the opinion that indexing units richer than the conventional single term index can be used, and propose to use the notion of lexical affinity derived from linguistics. They define lexical affinity between two units of language as the correlation of the common appearance when uttering the units in the language.

However, in contrast to Maarek et al's opinion, Salton [25] states that " The available options in phrase generation appear limited, and the introduction of costly and refined methodologies may bring only marginal improvements". The lexical affinities are extracted by using a technique of passing a window over the text. The size of the window is decreased as it reaches the end of the sentence (so as not to cross it). After the lexical affinities are extracted, the potential ones are stored such that they are representative of the singular form for nouns, and the infinitive form for verbs in them. The final lexical affinities are selected from the above list of extracted ones by the frequency of appearance -which the authors believe is an indication of their relevance. To reduce the possibility of having lexical affinities that do not contain the most information, they evaluate the resolving power of a lexical affinity, and based upon this, evaluate the lexical affinity.

For classifying the software, a Hierarchical Agglomerative Clustering (HAC) technique is used. From the index built using the above method, an inverted file index is obtained from a profile repository built during indexing. A profile in this case is the object that represents the reusable object, while the profile is a set of characterizing attributes of the object. The HAC clustering method constructs hierarchies over a set of items whereby each internal node is a cluster of items, and the leaves are where the single items are kept. The HAC method builds levels of clusters iteratively. These level clusterings also form coarser partitions iteratively. Thus, two steps are done iteratively: identifying the two clusters that are most similar, and merging them together into a single cluster. It is worth noting that by doing the above, [6] points out that hierarchical agglomerative clustering methods often end up doing chaining, which would merely be an ordering of components rather than grouping them according to functional similarity.

2.3 CATALOG

CATALOG [9] is a prototype software information system built at AT&T Bell Labs. Classification and retrieval is done on a small number of components that form a system built for interactive and reliability analysis. While the source code is not indexed, and is stored as a portion of records, the text of descriptions required for every module is stored as individual records. Inverted lists of indexing terms contain all single terms. The retrieval is done by supplying Boolean combinations of search terms, and partial matching techniques such as phonetic matching.

The drawbacks of this approach include the restriction on the user having to know exactly the right terms to search for (as in the Unix online manual). Also, upon a user supplying valid search terms to match the indices, relevant information might not be retrieved because it could be represented with other index terms in the module text descriptions.

The interface does not guide the user efficiently in restricting retrieval on more customary terms. Furthermore, the experimentation does not cover effective ways of representing software, and is considered more of an ad hoc approach to reusability.

2.4 CLIS

CLIS [4] employs a knowledge-based information retrieval (KBIR) model for its software reuse library system. The underlying function that this model in based on is given by the tuple <H, C, Q, M>, where

H: is the Hierarchical Concept Graph (HCG) representing the relationships between index terms utilized for the representation of software components and queries.

C: set of software components to be retrieved for reuse.

Q: set of queries. A Query has to be a Boolean expression over the index terms.

M: serves as the function that computes similarity between a query and a software component.

In CLIS, the relations between keywords make up the HCG. This is the knowledge base containing 'generalization' relations such as "part-of", "is-a", and "broader-than". The HCG is a directed acyclic graph in which a concept is a node. The node is used as an index term. The HCG is also used for computing the similarity distance of the reuser's query to the software components. The similarity distance is determined by the estimation of distance between two concepts. This distance estimation is computed as the distance between two nodes (concepts) by considering the path length between the nodes. Weights are used on the edges of the HCG to represent conceptual closeness between nodes connected by edges. The conceptual distance is computed by summing up the weights on the edges when traversing the shortest path between any two nodes.

Since CLIS employs the faceted classification model, it suffers from some of the same problems as the faceted approach. These are: construction and maintenance of the conceptual graph being extremely time-consuming; the need for the query to match the exact number of facets; the reuser is being required to know exactly what items to search for (as in CATALOG above).

2.5 The REUSE System

REUSE (REUsing Software Efficiently) [1] was developed with the intention of assisting software engineers in cataloging and retrieving existing software information. The software components in REUSE are represented -according to the information retrieval model- as documents made up of paragraphs, sentences and words.

A chain of user menus is provided within the REUSE system. The system thesaurus has a dictionary of all the menu keywords and other words which can be considered important for the organization wanting to reuse software. The system of menus represents the required information associated with each component, and any additions or modifications to the components will require modifications to the menu system (this can be onerous since extensive additions, for example, would require extensive changes to the menu system).

An enhanced version of the inverted index that includes additional word location information is used. The stored word locations are used to make numeric comparisons for indentifying adjacent words.

Structured information in software components comprises a specific keyword together with a response field. These are necessary since the system has to request data based on this relationship, and the keyword-response relationship must be preserved. This enables search for words within a paragraph and also within a document.

For the determination of whether a word is utilized in the response field of the specific keyword, both of them must appear in the same paragraph of the same document. This is a certain drawback of the REUSE system, since some important information not appearing contiguously would be missed. As the authors [1] point out, since the component classification list form the basis for the user menus, the number of component types can never be reduced. This a highly restrictive attribute of the REUSE system.

2.6 AIRS

AIRS (AI-based Reuse System) [21] was initially developed as a system to reuse Ada packages. However, it has, through enhancements, evolved into a general tool for reuse.

AIRS uses facets [23] together with a semantic network approach. The facets are used to ease the complex and extremely task-intensive process of creating a semantic network. Frames in a hierarchical network contain information about the software objects, their grouping, and the relationships between them. The frames system was originally proposed by Charniak et at [3], and the one used in AIRS is a modification of it.

The model for similarity computation and classification in AIRS is based on three types of software objects. Features are used to describe a software component. These features are similar to the facets described earlier. A component is a set of (f, t) pairs, where f is a feature in a given feature space while t being a term of f. A package is a collection of software components.

Comparison of components' descriptions is performed by comparing the distance between their respective descriptions. Two types of relations are used for this: the subsumption relation and the closeness relation. Subsumption refers to the scenario when a component in the software base can directly provide the functionality of the queried component. The closeness relation provides a similarity measure when the queried component can be obtained from an existing one through modification. The subsumption relation is expressed using a directed acyclic graph. Every node represents a component, and an arc between a source node B and a destination node A means that B can be reused to construct A. Weights on the arcs are used as a measure of the estimated effort required to construct A from B. In the case of the closeness relation, a feature graph allows the arrangement of terms (of features) in a directed graph. Weights are once again used to represent the expected effort required to obtain term t_2 given term t_1. Both feature and subsumer graphs are designed through the intuition and knowledge of the domain being modeled.

The drawbacks with the AIRS system are:

1. All software components are defined through the use of a fixed set of features. Also, a component must be described in terms of all the features of the class.

2. It is difficult for the library designer to maintain the class hierarchy upon addition of objects to it.

3. The classification and retrieval mechanism is highly intertwined with the method of computing similarity. This makes it quite difficult to incorporate new methods for computing similarity into the system.

Chapter 3

Software Component Representation

3.1 Representing the functionality of components

The software components to be inserted into the repository are expressed in terms of their functionality, using a knowledge representation language based on Telos [18]. The importance of using a Telos-like language to represent knowledge about information systems is discussed in [18], as is the value of using a knowledge representation language.

We use the representation technique and structure outlined in [8] [10] to represent the components. In this representation method, each component is expressed as a *functional description* (FD). A FD consists of one or more *features*. A feature is a triple *(verb, noun, weight)*, where the *verb* is the action or operation performed, the *noun* is the object upon which the operation is performed, and the *weight* is a number indicating the relative importance of the feature within its FD. In other words, this weight represents the degree of functionality that a particular *verb-noun* pair contributes to a software component. A sample *functional description* (FD) with three features is shown below.

```
FD: open-file  L
    read-file  M
    close-file L
```

The letters M,L represent weights for Medium and Low respectively. The mapping

between the letters and weight values is

```
Very High = 1
High      = 1/2
Medium    = 1/4
Low       = 1/8
Very Low  = 1/16
```

In case of synonyms verbs or nouns, a thesaurus is used to store those synonyms.

3.2 Software Component Similarity

Given any two software components, our repository organization schemes need to compute their similarity. The similarity between two components is a number indicating how close the two components are, for the purpose of using them interchangeably during software development. Several similarity computation methods have been reported in various disciplines and contexts. Possibly, the most well-known are the ones dealing with text retrieval [27] [25]. Since we deal with software artifacts rather than plain text documents, we feel that the techniques described in [8] [10] are most suited for our purposes. The similarity computation method [1] in [8] [10] takes as input two software components expressed as functional descriptions and returns a number between 0 and 1, which indicates how similar the two components are. The closer this number is to 1, the more similar the components are, and vice versa. For example, a similarity value of 1 between two components means that the one component can completely replace the other in the software development process. While [8] does not address the usability issue of their technique for computing similarity, it has been used in the Ithaca Software Information Base, which is a large ESPRIT reuse project [16].

[8] gives an excellent step-by-step example of the similarity computation method. For convenience, we also provide such an example later in this section. Following the terminology in [8], given two components expressed as FDs, one will be referred to as the *source* FD and the other as the *stored* FD. The computation involves several matrices as described below.

[1] As outlined in [8], this similarity computation is non-symmetric, non-transitive, and does not exhibit the triangular inequality property.

1. The EQ (equivalence) matrix expresses the degree of keyword compatibility between the i-th feature of the source FD and the j-th feature of the stored FD.[2] EQ is an *f-source* $\times$ *f-stored matrix*, where *f-source* is the number of features in the source FD and *f-stored* is the number of features in the stored FD.

2. The IMP matrix shows the degree of satisfaction that a source description is compatible (or can be replaced) with a stored description. Its entries show the importance between the j-th feature of a stored FD, and the i-th feature of a source FD. This importance is computed as $min(1,\ B/A)$, where B is the weight of a $verb-noun$ pair of the source FD, and A is the weight of a $verb-noun$ pair of the stored FD. The values of A and B are members of the set $\{1,\ 1/2,\ 1/4,\ 1/8,\ 1/16\}$, which represent weight values corresponding to {VH (very high), H (high), M (medium), L (low), VL (very low)}. IMP is an *f-stored* $\times$ *f-source* matrix, where *f-source* is the number of features in the source FD, and *f-stored* is the number of features in the stored FD. For any entry of the EQ such that EQ[i][j] = 0, in the IMP matrix, the entry IMP[j][i] is also zero. The value of the remaining entries of IMP is computed as $min(1,\ B/A)$ as described above.

3. The W (weight) matrix holds the *normalized* values of the source FD. Each entry of matrix W is a number between 0 and 1, and represents the percentage of the corresponding weight within its FD. There is one weight per feature in any given FD, therefore the size of matrix W is $f\ \times\ 1$, where f is the number of features in the FD.

4. The SAT (satisfaction) matrix combines matrices EQ and IMP, and is computed as EQ $\times$ IMP. SAT is *f-source* $\times$ *f-source* in size.

5. The SIM (similarity) matrix is the product of SAT $\times$ W. Entry SIM[j] of matrix SIM represents a weighted satisfaction index for feature j of the source FD with respect to the stored FD.

6. The final similarity value (also called *the confidence value*) is the result of the computation, and is obtained by summing up all the elements of SIM. This is a number between 0 and 1 inclusive, and represents the closeness of the source FD to the stored FD. The higher the similarity value, the closer the source FD is to the stored FD.

[2] A thesaurus is used to store synonyms (if any) and their degree of compatibility. In case of no synonyms, the compatibility degree is either 1 (if a match occurs) or 0 (if no match occurs).

Feature Number	*Source FD*	*Stored FD*
first feature	top-stack M	size-array M
second feature	pop-stack VH	size-queue L
third feature		top-stack VL

Table 3.1: A source FD and a stored FD

3.2.1 A Similarity Computation Worked Example

Consider two software components whose FDs are shown in Table 3.1 below.

According to the above, the sizes of the matrices will be: EQ=[2x3], IMP=[3x2], W=[2x1], SAT=[2x2], and SIM=[2x1].

The source and stored FDs of Table 3.1 have the same keywords (top-stack) in their first and third features respectively. Therefore, EQ[0][2] = 1, and all other entries of matrix EQ are zero. Considering the importance matrix IMP, IMP[2][0] = 0.25, because the third feature of the stored FD is compatible with the first feature of the source FD, and min(1,B/A) = min(1,VL/M) = $min\{1, (\frac{1/16}{1/4})\}$ = 1/4 = 0.25. All other entries of IMP are zero since all other entries of EQ are zero. The weight matrix W holds the normalized values of the weights of the source FD. In the source FD, we have the weights M and VH. According to the mapping (1, 1/2, 1/4, 1/8, 1/16) = (VH, H, M, L, VL), M is 1/4 and VH is 1, i.e. VH is four times as much as M. Thus, if we assume that the weights of the source FD amount to 5 units, 4 of these units are occupied by VH and 1 unit is occupied by M, i.e., 80% by VH and 20% by M, and hence, W[0] = 0.2, W[1] = 0.8. Matrices EQ, IMP and W are shown below.

$$EQ = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}; \qquad IMP = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0.25 & 0 \end{pmatrix}; \qquad W = \begin{pmatrix} 0.2 \\ 0.8 \end{pmatrix}.$$

As a result, matrices SAT and SIM are:

$$SAT = EQ \times IMP = \begin{pmatrix} 0.25 & 0 \\ 0 & 0 \end{pmatrix}; \qquad SIM = SAT \times W = \begin{pmatrix} 0.05 \\ 0 \end{pmatrix}.$$

Therefore, the similarity value between the given source and stored FDs is SIM[0] + SIM[1] = 0.05. This implies that the two FDs are not very similar, and furthermore, if the stored FD was to replace the source FD, then it would not be a good candidate for it.

Chapter 4

The Clustering Schemes

In this chapter, we present two heuristic clustering schemes for organizing the software repository. A *cluster* is the term referring to a group of items that are similar enough with respect to some measure within the cluster, and different enough between other clusters. The term *heuristic clustering* is due to Salton [25], and differs from other clustering methods in that it produces clusters quickly at little cost, and needs no advance knowledge of the similarities of the objects to be clustered. The inspiration to use heuristic clustering for software reuse originated from [19].

The other known clustering method used to organize software repositories [15] is hierarchical clustering [25]. In this method, a complete list of all pairwise similarities between the items to be clustered is first obtained, followed by a grouping mechanism which groups items that are similar enough, by some measure, into clusters. The pre-computed similarities are then compared during each iteration of the sorting of the pairwise similarities. This is to obtain the most similar (closest) item to be clustered first, followed by the next most similar, and so on. The clustering can be carried out divisively, or agglomeratively, the difference being that in the first case, one complete cluster is subsequently divided to form smaller clusters, while in the latter, several clusters are assimilated into a larger cluster in a hierarchical fashion.

With heuristic clustering methods, pairwise item similarities do not need to be computed in advance. Our first clustering scheme presented below is a simple one-pass clustering process. In both the schemes presented in this chapter, the components to be clustered can be taken one at a time in any arbitrary order. Note that we will use the term 'software component' to mean its functional description, since we are dealing with a software repository of functional descriptions.

4.1 The Sphere Packing Clustering Scheme (Scheme A)

4.1.1 Terminology

We use the symbol σ to represent the similarity value computed using the method described in Section 3.2. This is to differentiate between the similarity computed (σ) and the use of the word "similarity" for discussing how similar one component is to another. Also, we use the operator *sim* to represent the function that computes the similarity value σ as described in Section 3.2.

In this thesis, we sometimes use the term distance [1] to refer to how conceptually close (or far) is one component to another. In this context, a component is "close" to another if the similarity σ between them is high (the distance between them being small). If a component is "far" from another component, then the similarity σ between them is low (meaning that the distance between them is large).

4.1.2 The Scheme

This scheme centers around the idea that components inserted into the repository are organized into "spherical" clusters. The clusters (or "spheres") will have a center that is 'stationary'. By this, we mean that the center is fixed, and remains unchanged throughout the clustering process. Components inserted into a cluster have to be within a threshold value T from the center of the cluster. The first component to be inserted into the cluster automatically becomes the cluster's center. A more detailed description of the scheme is outlined below.

1. For every component c_i to be inserted into the repository
    2. If any clusters exist then
        3. compute the similarity of c_i with respect to
        the centers of all existing clusters
        4. Identify all clusters that can host c_i, and
        5. insert c_i into the cluster whose center is closest in terms of similarity
            6. If none of the existing clusters can host c_i then
                7. insert into a new cluster, making c_i its center

[1] Since distance has an "inverse" relationship with similarity, it is also non-symmetric, non-transitive, and does not exhibit the triangular inequality property.

8. else
    9. insert component into the first cluster and make it the cluster's center

Let c_1 be the first software component to be inserted into the repository. As yet, there are no existing clusters, so c_1 is inserted into a newly created cluster C_1 and becomes its center. Let c_2 be the second component to be inserted into the repository. The similarity σ of c_2 with respect to the center of cluster C_1 is computed. If $(1 - \sigma) \leq T$, then c_2 is inserted into cluster C_1. If $(1 - \sigma) > T$, then c_2 is inserted into a new cluster C_2, and also becomes its center. Recall that the greater the value of σ between two components, the more similar the components are. Since we would like to place similar components into the same cluster, we wish to have large similarity values between any two components of a cluster. Since $1 - \sigma$ becomes smaller as σ becomes larger (recall, $0 \leq \sigma \leq 1$), satisfaction of the condition "$1 - \sigma \leq T$" means that two components with similarity σ are "similar within T", and thus they should belong to the same cluster.

Consider an incoming component c_i. If there are m existing clusters C_1 to C_m, then for each incoming component to be clustered, we compute the similarity of the incoming component c_i with respect to the centers of all existing clusters.
$\sigma_{i1} = sim(c_i, C_1)$
$\sigma_{i2} = sim(c_i, C_2)$

$\vdots$
$\sigma_{im} = sim(c_i, C_m)$

The next step is to check if the similarities which are the result of the above computation fall within the threshold T.
$(1 - \sigma_{i1}) \leq T$
$(1 - \sigma_{i2}) \leq T$

$\vdots$
$(1 - \sigma_{im}) \leq T$

For those clusters whereby the σ value of the component to their centers is less than T, making the clusters ineligible to host c_i, an auxiliary array called *host_clusters* is used to record this information. For any cluster C_j that is not eligible to host component c_i, its corresponding entry in the *host_cluster* array is set to 0. If the cluster C_j is eligible to host a component, then the corresponding entry in the auxiliary array is set to 1.

Component	*Features*
c_1	push-element H pop-element H
c_2	size-array VL size-queue L
c_3	top-stack M pop-stack VH
c_4	push-element H
c_5	size-array L size-queue M

Table 4.1: The Components to be Clustered.

For those clusters that are able to host c_i, the component will be inserted into the closest cluster to c_i in terms of the similarity between c_i and that cluster. Thus, we sort all the similarity values of the component with respect to all clusters whose corresponding entry in the host_cluster array is 1 to obtain the greatest value among all possible hosting clusters. The component is then inserted into the cluster that has the center whose similarity value with respect to the component is greatest (i.e. it is closest to the component).

In the case that no cluster can host c_i, a new cluster is created, and c_i is then inserted into it and made its center. The above scheme is repeated for every incoming component, until all are clustered. Since the center of each cluster is rigid, and components are "packed" around the centers, we have aptly called it the "sphere-packing" scheme.

4.2 An Example of Clustering Scheme A

The five components $c_1, c_2, ..., c_5$ to be clustered are shown in Table 4.1.

Upon application of scheme A for clustering the components, the following takes place. c_1 is the first component to be clustered. Since there are no existing clusters yet in the repository, c_1 is inserted into the new cluster C_1 and made its center. Next, c_2 is to be clustered. The similarity of c_2 with respect to C_1's center is zero[2]. Since

[2]The similarity σ is computed as described in Section 3.2.

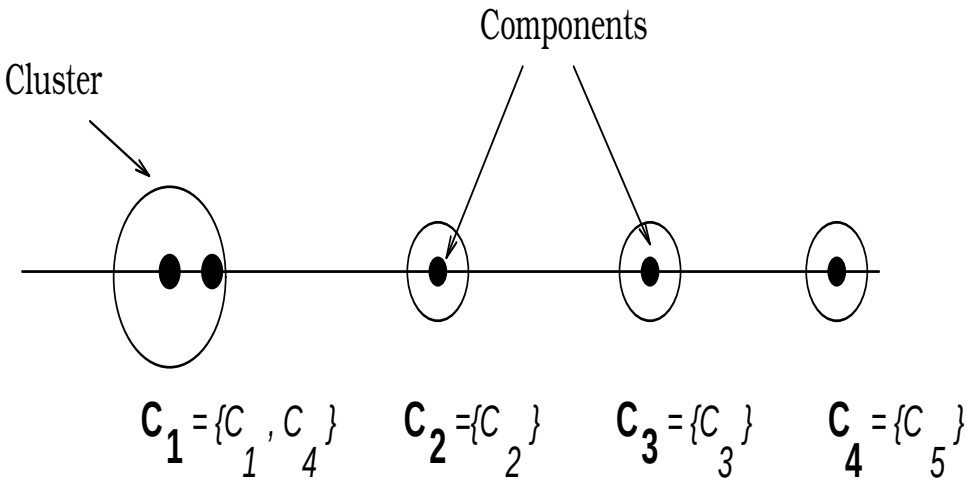

Figure 4.1: A Pictorial Representation of the Clustered Components for the Sphere-Packing Scheme

the threshold $T = 0.35$, then the difference (1-σ) where σ is the similarity value equals one and exceeds the threshold T. Thus, c_2 is inserted into a new cluster C_2 and also becomes its center.

The similarity of c_3 with respect to both the fixed centers of the two clusters is zero. Once again, (1-σ) (for c_3) is 1 and does not fall within the threshold T. This implies that c_3 goes into a new cluster C_3. Next, c_4's similarity with respect to the center of cluster C_1 is 1, and (1-σ) gives 0 which is less than T. Thus, c_4 is inserted into cluster C_1. For c_5, the same is true as for c_3 above, and c_5 ends up being in a new cluster by itself while also becoming that cluster's fixed center.

The final positions of the components are shown on a one-dimensional scale in Fig. 4.1. The components located in the centers of the circles are the fixed centers of the clusters.

4.3 The Multidimensional Clustering Scheme (Scheme B)

4.3.1 Motivation

The previous clustering scheme is remarkable in its simplicity, but has its drawbacks.

If the fixed center is a good representative of future components to be clustered that are similar to it, then scheme A will work very well. However, there is no way of exactly predicting how many, or what functionalities future components will have. Thus, a fixed center could lead to a poor representation of a cluster's overall composition in terms of the functionality of the software components in it.

The problem in having a dynamic center is that how do we "move" the center, or, equivalently, how do we vary the similarity value associated with the center that is representative of the cluster as more components are added to the cluster?

If we were to vary the center then how do we compute similarity for a dynamic cluster center? In scheme A, if we were to adjust the fixed center -to make it dynamic- by, let's say, modifying the center's similarity value each time another component is added to the cluster, then what would be the basis of the center's modification?

Answers to these questions, and thus solutions to the above problems, are outlined in the following section, where we propose another heuristic clustering scheme to be used for organizing software repositories.

4.3.2 Terminology

We use σ to represent the similarity computed between the the component to be clustered and the reference components. We also use *sim* as the function which returns the value σ, computed as described in Section 3.2.

4.3.3 The Scheme

The basic idea is to perceive the software repository as a k-dimensional space defined by k reference components (k is set by the software repository organizer). Incoming software components are inserted into the repository according to their "location" in the k-dimensional space, and clusters of similar components are formed. Components which are close to each other (according to their "location" in the k-dimensional space) are placed into the same cluster. Each cluster is associated with its "center". The center is a k-dimensional vector and it represents the average location of all components of the corresponding cluster within the k-dimensional space.

The location of a software component is a k-dimensional vector which indicates the relative position of the software component with respect to k reference com-

ponents. The reference components are constructed by the repository designer or administrator and they are fixed prior to initiating any component insertions into the repository. Also, they remain unchanged through the lifetime of the repository[3]. The reference components should be "well spread out" so that no two dissimilar software components end up having the same relative position with respect to the reference components. Later in this chapter (Section 5.2), we discuss how to make good choices of reference components. The clustering method is described below.

1. For every component c_{new} to be inserted into the repository
   2. compute $sim(c_{new}, R_i)$ for $i = 1$ *to* k
   3. If no clusters exist then
      4. insert c_i into a new cluster
      5. assign $sim(c_{new}, R_i)$ for $i = 1$ *to* k to the cluster's center $\vec{C} = (C_{j1}, C_{j2}, \cdots, C_{jk})$
   6. else
   7. compute $T^{(j)}_{new,i} = \sigma_{new,i} - C_{ji}$ for every cluster C_j $(for\ j = 1\ to\ m)$ and for every $R_i (for\ i = 1\ to\ k)$
      8. if $\mid T^{(j)}_{new,i} \mid \leq T$, matrix $[i][j] = \mid T^{(j)}_{new,i} \mid$ (= -1 otherwise)
         9. set array $NotInC[j] = 0$
         10. else $NotInC[j] = 1$
      11. if $C_1, C_2, ..., C_w$ (where $1 \leq w \leq m$) can host c_i
         12. insert c_i into cluster $C_h (1 \leq h \leq w)$ such that $\sum_{i=1}^{k} matrix[i][h] = min\left\{\sum_{i=1}^{k} matrix[i][1],\ \sum_{i=1}^{k} matrix[i][2], ...,\ \sum_{i=1}^{k} matrix[i][w]\right\}$
         13. modify center of the cluster to reflect addition of c_{new}
      14. else
      15. insert c_i into a new cluster
         16. assign $sim(c_i, R_j)$ for $j = 1$ *to* k to the cluster's center $\vec{C}$

In scheme B, each cluster center has a similarity value with respect to each of the k dimensions defined by the k reference components respectively. The similarity σ of each component is also computed with respect to each of the k reference components. To obtain the difference in similarity between each component to be clustered and the cluster centers, the difference between the similarity σ (in each dimension) of the component and the cluster center (also in each of the k dimensions respectively) is computed. Since both the values which are used to obtain the difference are with respect to the reference components, the difference in each dimension translates into

[3]If the reference components change, then the entire repository needs to be reorganized

the distance between the component and the cluster center. Thus, the direct correlation of difference with distance allows the use of the difference as distance between the component to be clustered and the cluster center. In scheme A, the center of each cluster does not have any similarity value(s) associated with it. When the similarity σ of a component to be clustered is computed, it is only with respect to the fixed cluster center (the cluster center is the first component to be inserted into the cluster).

If c_{new} is the component to be clustered (inserted into the repository), we compute:

$$\sigma_{new1} = sim(c_{new}, R_1) = C_{11} \pm T^{(1)}_{new1} = C_{21} \pm T^{(2)}_{new1}$$

$$\vdots$$

$$\sigma_{newk} = sim(c_{new}, R_k) = C_{1k} \pm T^{(1)}_{newk} = C_{2k} \pm T^{(2)}_{newk}.$$

Given that the threshold value T is the same as before, and two clusters C_1 and C_2 exist, then

$condition_1 = (T^{(1)}_{new1} \leq T \; and \; T^{(1)}_{new2} \leq T, ..and.., T^{(1)}_{newk} \leq T)$
$condition_2 = (T^{(2)}_{new1} \leq T \; and \; T^{(2)}_{new2} \leq T, ..and.., T^{(2)}_{newk} \leq T)$,
$condition_1$: check how similar c_{new} is from center of C_1
$condition_2$: check how similar c_{new} is from center of C_2.

Consider the cases for the various possibilities of truth values (where TRUE = 1 and FALSE = 0),
Case 1: ($condition_1 = 1 \; and \; condition_2 = 1$)
- insert c_3 into the (existing) cluster C_1 or C_2
- adjust the center of the hosting cluster, similar to the case for component c_2
Case 2: ($condition_1 = 1 \; and \; condition_2 = 0$)
- insert c_3 into the (existing) cluster C_1
- adjust the center of C_1 similar to the case for component c_2
Case 3: ($condition_1 = 0 \; and \; condition_2 = 1$)
- insert c_3 into the (existing) cluster C_2
- adjust the center of C_2 similar to the case for component c_2
Case 4: ($condition_1 = 0 \; and \; condition_2 = 0$)
- create new cluster C_3
- associate with C_3, the tuple $(C_{31}, ..., C_{3k}) = (\sigma_{31}, ..., \sigma 3k)$ as its center.

In the case that a component is to be inserted into an existing cluster already having at least one component in it, the weight of the center of that cluster prior to

the addition of the new component has to be taken into account. This implies that the center of the cluster resulting from the insertion of the new component is not "half-way" from the current center, but rather of the value

$$\vec{C} \mp \frac{\vec{T}_{new-component}}{|C|+1},$$

where $\vec{C} = (C_{j1}, C_{j2}, \cdots, C_{jk})$ for cluster C_j, $\vec{T} = (T_{z1}^{(j)}, T_{z2}^{(j)}, ..., T_{zk}^{(j)})$ for cluster C_j for component c_z, from the center of the existing cluster, and $\mid C \mid$ = the number of components in the cluster before the insertion of the new component.

There are two issues yet to be resolved for placing a component into a cluster. First, assuming that there are m clusters $C_1, C_2, ...C_m$ (where $m > 2$) created so far, how do we check if a component can be placed into any (and which) of these clusters? Second, having found all the clusters that are eligible to host a component, how do we determine which of them is the closest to this component?

A method that can be easily automated by implementing it in a software tool for the clustering, finding clusters that are eligible to host a component, as well as identifying the best (nearest) cluster for hosting the component is outlined below.

Let c_{new} be a component to be clustered, and $R_1, R_2, .., R_k$ be the reference components, and $C_1, C_2, ..., C_m$ be the clusters.

In order to check if any of these clusters (and which ones) can host component c_{new}, we can have a matrix whose rows represent the reference components and whose columns represent the clusters.

We compute $T_{new,i}^{(j)} = \sigma_{new,i} - C_{ji}$ for every cluster C_j ($for\ j = 1\ to\ m$) and for every reference component R_i ($for\ i = 1\ to\ k$). This is to compute the difference between the similarity value with respect to each reference component and each cluster's center. Then check if $\mid T_{new,i}^{(j)} \mid \leq T$.[4] This is to check if the difference falls within the threshold T. The absolute value of the difference is first taken before the comparison is made since in some cases, this difference can be less than T, while in others, it can be greater than T. However, the key criterion is that the difference should be within the "reach" of (fall within) the threshold T, since any difference greater than T is

[4]By using the absolute value $\mid T_{new,i}^{(j)} \mid$, we eliminate the need for the $\pm$ operations used during the insertions of c_1, c_2 and c_3 before.

considered as beyond the cluster's "radius" and thus, beyond reach for insertion into that particular cluster. If $\mid T^{(j)}_{new,i} \mid \leq T$, the value $\mid T^{(j)}_{new,i} \mid$ is stored into matrix $[i][j]$. Otherwise, $NotInC[j]$, an auxiliary array of the size equal to the number of existing clusters and used to keep track of which clusters are ineligible to host a new component, is set to 1 (a zero entry in $NotInC[j]$ would mean that cluster j can host the new component). If $NotInC[j] \neq 0$, then matrix $[i][j]$ is set to -1.

Thus, matrix $[i][j]$ is ≥ 0 if cluster C_j can host new component c_{new} with respect to R_i, or -1 otherwise. $NotInC[j] = 1$ if it is determined that cluster C_j cannot host component c_{new}, and the entry equals 0 otherwise. After $\mid Reference\ Components \mid \cdot \mid clusters \mid$ number of operations, we will have passed the entire table. If $C_1, C_2, ..., C_w$ (where $1 \leq w \leq m$) are the clusters selected from the algorithm that determines which clusters can host c_{new}, then component c_{new} is inserted into cluster $C_h (1 \leq h \leq w)$ such that
$\sum_{i=1}^{k} matrix[i][h] = min\left\{\sum_{i=1}^{k} matrix[i][1],\ \sum_{i=1}^{k} matrix[i][2], ..., \sum_{i=1}^{k} matrix[i][w]\right\}$.

Note that while the difference in similarity σ between two components gives us directly the difference in similarity between them, even if the difference in distance (where distance $= 1 - \sigma$) between them was used, this difference would be exactly the same in value as the difference obtained from their similarities. This is because

$$if \mid \sigma_1 - \sigma_2 \mid = difference, then \mid (1 - \sigma_1) - (1 - \sigma_2) \mid = \mid \sigma_2 - \sigma_1 \mid = difference.$$

4.4 An Example of Clustering Scheme B

Consider the 5 components $c_1, ..., c_5$ used in Section 4.2 to be clustered, shown in Table 4.2 again for the reader's convenience.

To demonstrate the clustering of our sample components using scheme B, k is set to 2, i.e. we have two reference components, as shown in Table 4.3.

We consider the reference components as the stored FDs, and the incoming components $c_1, ..., c_5$ as the source FDs. Assume a threshold T = 0.35. The similarities between the incoming components and the reference components are computed (as in Section 3.2), and their final values are shown in Table 4.4.

In the next three figures, we show the effect of the multidimensional clustering scheme on the dynamic centers. Following that, the resulting cluster formation of the repository and the center of each cluster, are shown.

Component	*Features*
c_1	push-element H pop-element H
c_2	size-array VL size-queue L
c_3	top-stack M pop-stack VH
c_4	push-element H
c_5	size-array L size-queue M

Table 4.2: The Five Components to be Clustered.

R_1	R_2
push-element M	size-array M
pop-element H	size-queue L
	top-stack VL

Table 4.3: The reference components (defining a 2-dimensional space)

component	*reference components*	
	$sim(c_i, R_1)$	$sim(c_i, R_2)$
c_1	0.75	0.0
c_2	0.0	1.0
c_3	0.0	0.05
c_4	0.5	0.0
c_5	0.0	0.67

Table 4.4: Similarities between c_i and R_j

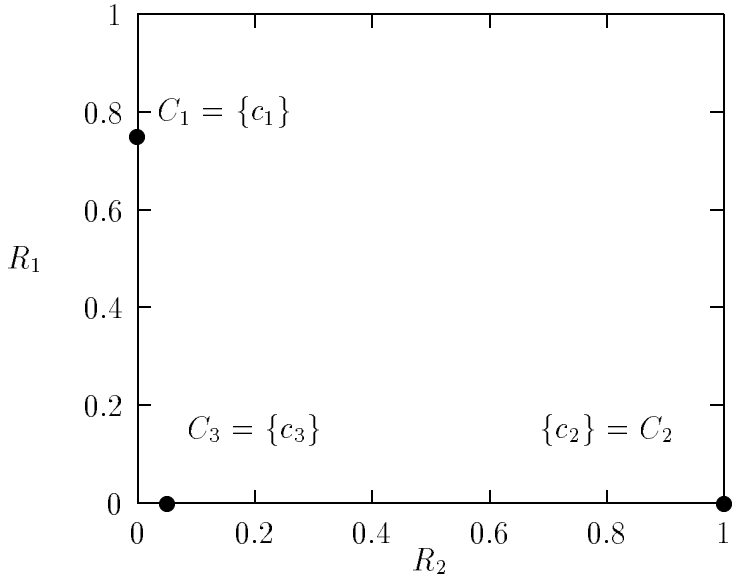

Figure 4.2: Cluster centers after insertion of c_1, c_2, c_3

Figure 4.2 is the first snapshot of the clusters after the first three components c_1, c_2 and c_3 are inserted into their respective clusters.

Next, upon insertion of component c_4 into cluster C_1, the cluster's center changes, and the shift in the 2-dimensional space (since we have two reference components) is shown by the arrow in Figure 4.3.

Component c_5 goes into cluster C_2 since its similarity only falls within the threshold with respect to C_2's center. The shift in the cluster's coordinates is due to the change in the center's coordinates and is shown in Figure 4.4.

The components' positions after insertion into the 2-dimensional space are shown in Fig. 4.5. The final positions of the clusters' centers are represented by a "X".

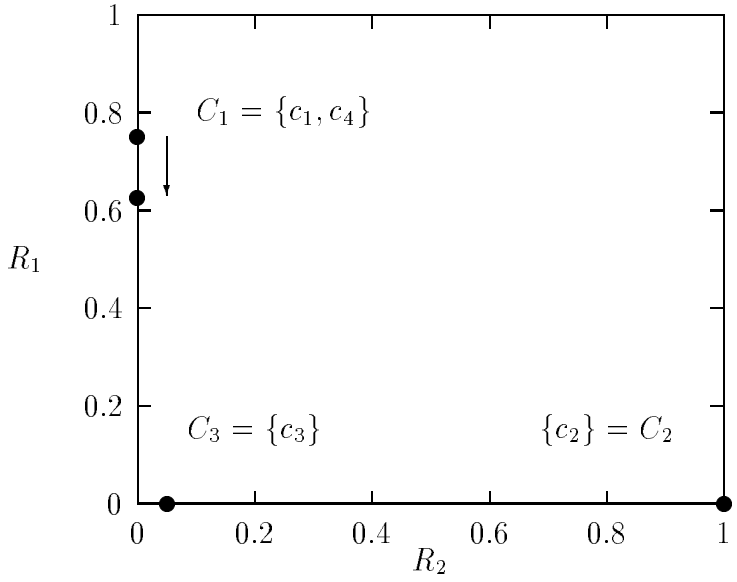

Figure 4.3: Cluster centers after insertion of c_4

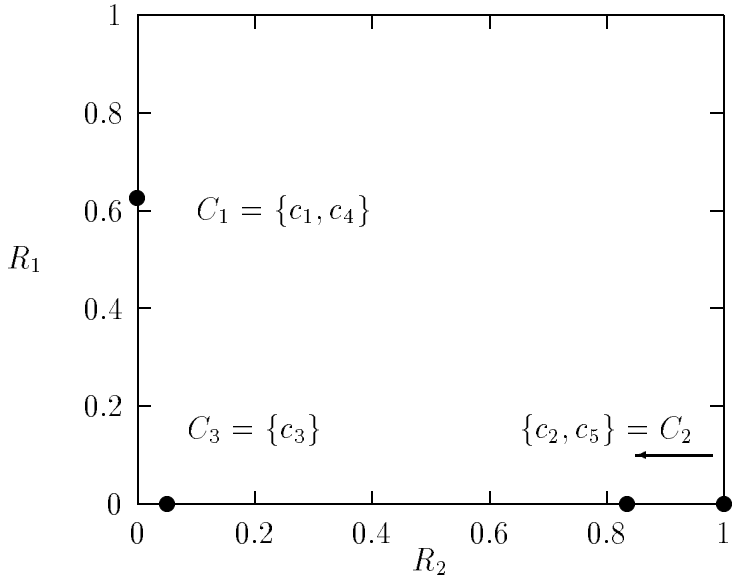

Figure 4.4: Cluster centers after insertion of c_5

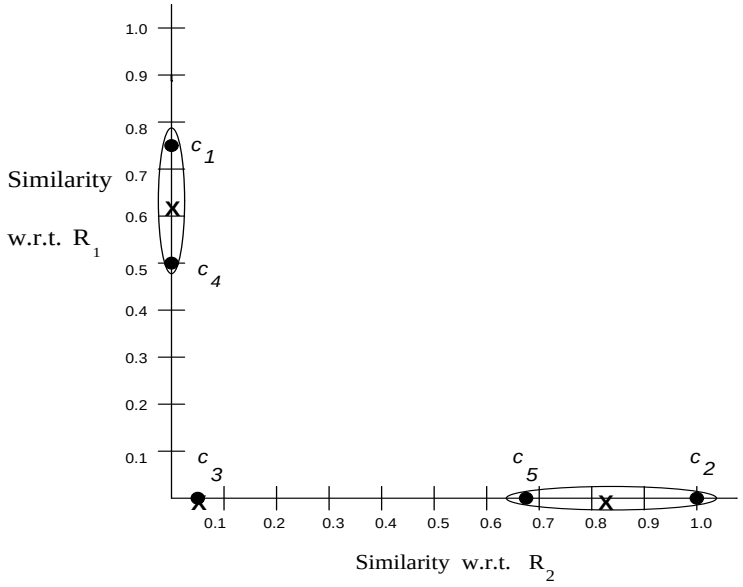

Figure 4.5: A Pictorial Representation of the Clustered Components With Respect to the Reference Components in terms of similarity σ.

Chapter 5

Performance Evaluation

5.1 Evaluation Methodology

We have implemented a prototype tool that organizes the repository using both of our clustering schemes, and allows the reuser to retrieve the conceptually closest component.

The components were queried after the repository was organized using the heuristic clustering schemes. A library of *data structures* component descriptions were used as our test data. This library consists of about 150 components that perform operations on the various data structures such as arrays, stacks, queues, B-trees and graphs used quite often in software development. All components used in our experiments are listed in Appendix A. The components in this library are based on a popular data structures text [13] used in data structures courses in universities. Our reason for using a library of data structures was due to the consensus among the software reuse community that there is a need to reuse "relatively small but very carefully designed collections of domain-specific software" [17]. The combination of features has been done using the software engineering principles of *high cohesion* and *low coupling*, including a few variations to make the data variant.

The following two query components are used to query the repository.

Q_1:	Q_2:
restore-btree M	swap-pointers H
size-array L	

The query is processed exactly as in the clustering mechanism, but after the closest cluster to the query is determined, all components of that cluster are retrieved.

The performance evaluation is done in terms of *recall*, *precision* and *cluster tightness*. Our results for the benchmarks recall and precision versus the threshold T are shown as graphs below for the two queries Q_1 and Q_2 submitted to the repository. Recall is defined as the ratio $\frac{r}{R}$. Precision is defined as the ratio $\frac{r}{c}$, where r = number of relevant components retrieved by query Q_i ($\bar{r}$ is the number of irrelevant components with respect to Q_i), R = number of relevant components in the repository ($\bar{R}$ is the number of irrelevant components in the repository) with respect to Q_i, c is the total number of components retrieved by Q_i, and C is the total number of components in the repository. We consider one component to be relevant to another if there is at least one common verb-noun pair between them. Since reuse of a component always requires some modification on the reuser's part, it follows that at least one common verb-noun pair between a source component and a stored component will allow reuse. Our choice of recall and precision as metrics for the evaluation of how well the clustering scheme functions by allowing the reuser to search for, and retrieve, the conceptually closest component is due to [26].

We also study the effect of the variation of cluster structure on the above benchmarks by first clustering the components inserted in one order, and then inserting them into the repository in the reverse order.

To evaluate the clusters formed from the two clustering schemes, we also investigate into how far away (in terms of similarity σ) are components that are inserted into clusters. To do this, we compute the distance of each component belonging to that cluster from it's center. We then compute the average distance of components in each cluster, and obtain the average over all clusters while varying T. This gives us an idea of how close to the center the components are, and thus how similar they are within a cluster (this is especially important in scheme B since the centers are varied accordingly when a new component is inserted into an existing cluster). Since T is the largest value that a cluster's radii can have (for each T value), the average radius of each cluster should certainly be no more than T, and if the clusters have a good number of functionally similar components in them, the radius should definitely be less than T.

5.2 Construction of Reference Components

Since the reference components provide the means to calculate the relative location of each component in our k-dimensional repository, for scheme B, we believe that they merit a discussion of how they are constructed.

The construction of reference components is a domain-intensive task. By this, we mean that a domain analysis has to be done, given the domain to which the components to be inserted into the repository belong. Todate, there is no automated means of performing a domain analysis, as [24] states, "Domain Analysis is a knowledge intensive activity for which no methodology or any kind of formalization is yet available." An important process in domain analysis is the "capture of the essential functionality required in that domain"[24]. In domain analysis, common characteristics from systems are captured and generalized. This implies that domain analysis is quite similar to knowledge acquisition, where knowledge has to be elicited from the domain, and encapsulated into the requirements of the system.

We have had to elicit the knowledge from the components to be inserted into the repository so that the reference components are representative of the domain. In our case, we have followed two principles for ensuring that the reference components truly represent the domain:

1. Construct functional descriptions of components that encompass the domain. (in our case, this is the data structures domain). This was done by extracting all possible components from among the data structures components and making them into reference components so that the domain would be completely represented; however, it was ensured that no feature appeared more than once among the reference components.

2. In constructing functional descriptions of components, ensure that every feature occurring in the components to be inserted into the repository occurs no more than once among the reference components. This implies that the reference components are distinct, i.e. no two reference components contain a common verb-noun pair. Note that both these steps are highly manual tasks.

It is important to point out that once the domain is well-understood and completely represented, there is no further need to construct new reference components. This is often the case when an organization becomes mature in its development of systems with respect to a particular domain. As [17] points out, corporations have

realized that reuse across domains is not done once the corporation's system development processes have matured. In [17], one researcher and software reuse practitioner points out that domain-specific collections is where reuse has had the most success, and therefore is where our future efforts should be directed. This supports our choice of using components limited to one domain. All the reference components used in our experiments are listed in Appendix B.

5.3 Test Results

The results of our experiments are plotted in Figs. 5.1 to 5.20. Appendix C shows the actual values used for these figures.

5.3.1 Recall and Precision

We label the graphs in the following manner:

$$< Scheme,\ Structure,\ Query,\ Number\ of\ Components >.$$

1. *Scheme* can be either S (for scheme A) or R (for scheme B)
2. *Structure* can be either n (for the initial investigation of components read in one order) or r (for components read in the reverse order)
3. *Query* is either Q_1 or Q_2
4. *Number of Components* can be 100 or 150.

The graphs for recall and precision for the clustering schemes A and B are shown below. The threshold T is increased by 0.1 each time and the recall and precision values for queries Q_1 and Q_2 are recorded.

Figs 5.1 and 5.2 show how recall varies for queries Q_1 and Q_2 for both clustering schemes. In Fig 5.1, we see that recall for Q_1 begins earlier (at T=0.1) for scheme A, while recall begins at T=0.3 for scheme B. However, from T=0.3 onwards, the recall for scheme B is much higher than for scheme A. This shows that scheme B has a much better recall performance than scheme A. In Fig. 5.2, recall -in the case

of Q_2- for scheme B starts much earlier (at T=0.2) than scheme A which produces recall at T=0.5. Furthermore, there is more of a gradation for recall in the case of scheme B than for scheme A. It is expected that recall would increase when the threshold is increased, since a higher T value means that there is a greater "tolerance" for components not so near to the cluster center in terms of similarity. This leads to the concept of a "wider" cluster, meaning that the closest components, as well as the similar but not so close components, will also be home to the same cluster. Our expectations are realized as the graphs demonstrate a general trend of increasing recall with increasing T.

In Figs. 5.3 and 5.4, the variation of precision is shown with respect to T. While both schemes maintain a high precision value for all values of T for which recall occurs, we see that because recall starts earlier (in Fig. 5.3) for scheme A, its precision values are extended over a longer T range (by two values of T). The opposite is true in the case of Q_2 (in Fig. 5.4) where the precision values of scheme B are extended over a longer range of T (from T= 0.2 to 0.4). For the graphs of precision versus T, we notice that whenever there is a non-zero recall value[1], the precision is at 1. This is indeed an encouraging result since this implies that the closest component(s) are always retrieved whenever retrieval occurs.

In Figures 5.5 to 5.8, we investigate into the variation of recall and precision with increase in repository size. After the number of components to be clustered is increased by 50%, the recall and precision is measured while varying T. It is clear from the graphs shown in Figs. 5.5 and 5.6 that scheme B performs better than scheme A in terms of recall for both the queries. In the case of query Q_1 in Fig. 5.7, precision for scheme A stays at 1, while for scheme B, it falls slightly by 10% to 0.9, which is still a very good precision value. For Q_2 (Fig. 5.8), precision for scheme B extends over more values of T than for scheme A, though both maintain a very high precision value of 1.

As the graphs demonstrate, recall values for scheme B do not vary by more than 10% over the recall values obtained before increasing the repository size. Also, the variation in precision for the same case is not more than 10%. For scheme A, recall values do vary by about 15% for some values of T (comparing Figs. 5.2 and 5.6 for Q_2) and for the value of T=0.7 (comparing Figures 5.1 and 5.5 for Q_1), by about 30%. Precision, however, stays a constant 1.0 for scheme A in all cases when a corresponding recall value is obtained. Thus, the performance of scheme B remains stable even when

[1]Note that when recall is zero, precision is also zero since when no items are retrieved, the value r in the precision ratio r/c is zero.

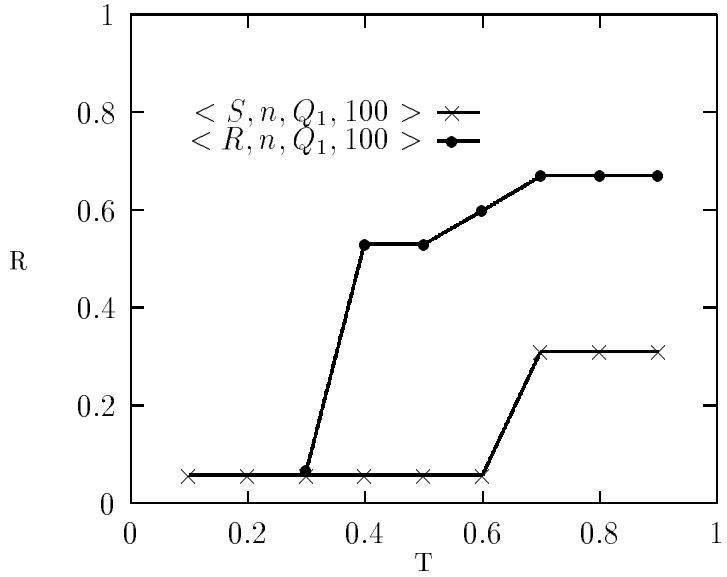

Figure 5.1: Recall (R) vs Threshold (T) for $< S, n, Q_1, 100 >$ and $< R, n, Q_1, 100 >$

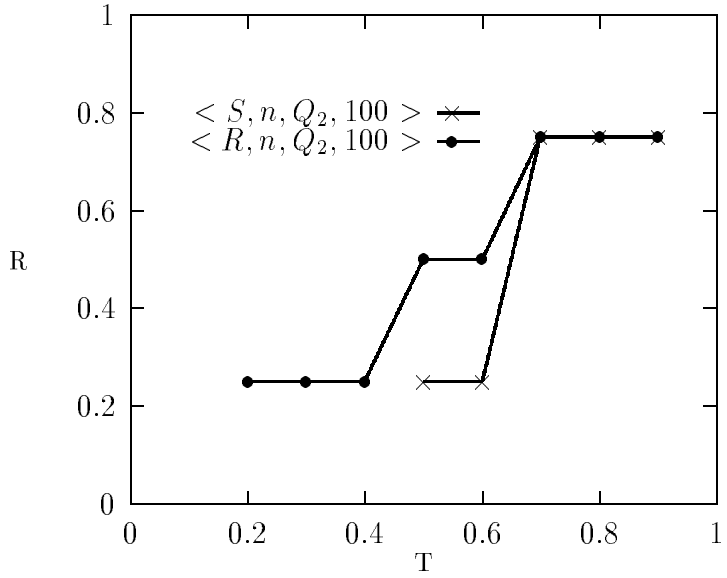

Figure 5.2: Recall (R) vs Threshold (T) for $< S, n, Q_2, 100 >$ and $< R, n, Q_2, 100 >$

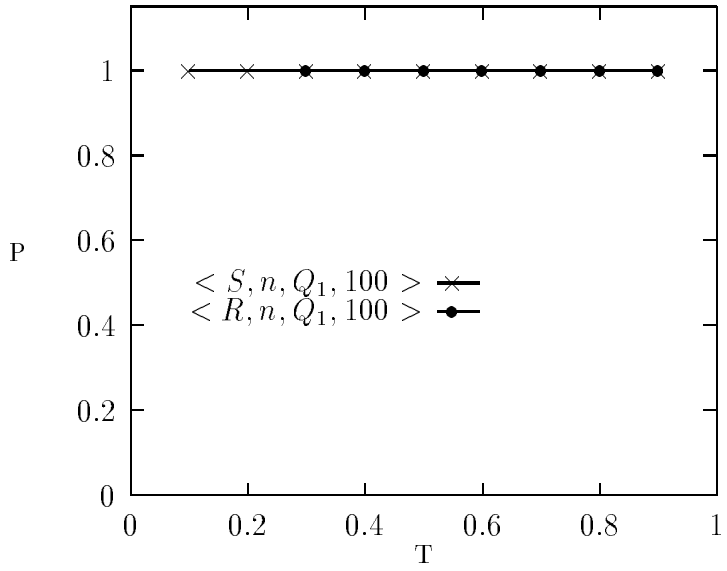

Figure 5.3: Precision (P) vs Threshold (T) for $< S, n, Q_1, 100 >$ and $< R, n, Q_1, 100 >$

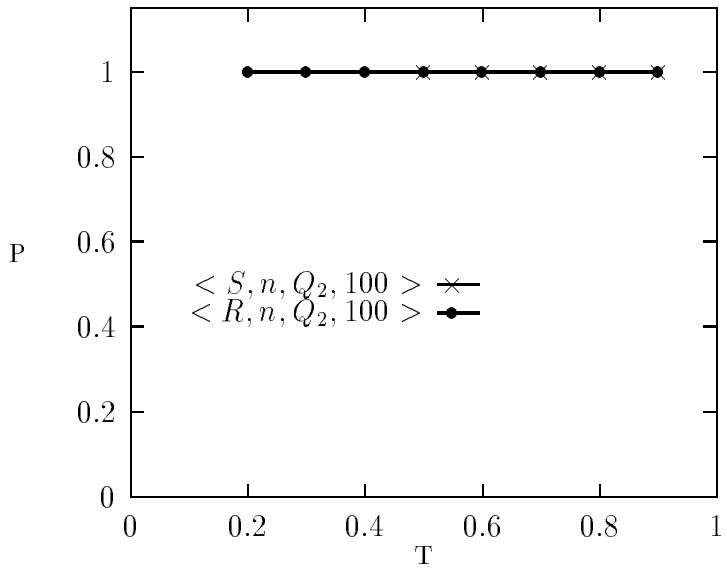

Figure 5.4: Precision (P) vs Threshold (T) for $< S, n, Q_2, 100 >$ and $< R, n, Q_2, 100 >$

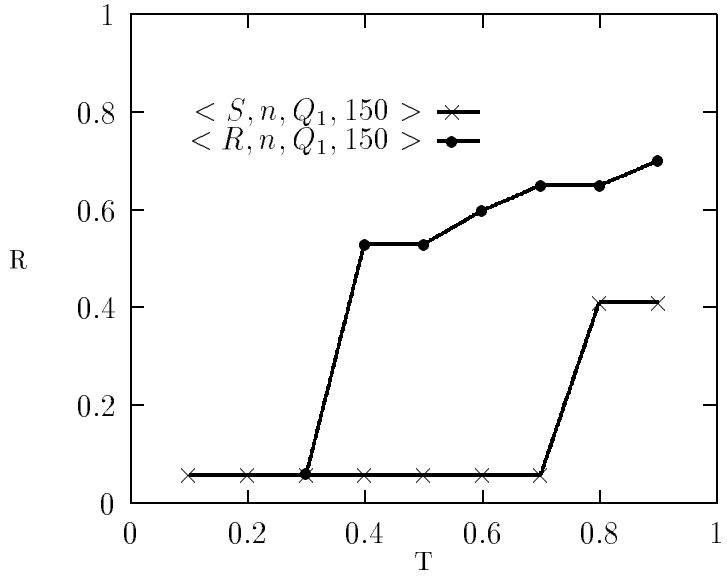

Figure 5.5: Recall (R) vs Threshold (T) for $< S, n, Q_1, 150 >$ and $< R, n, Q_1, 150 >$

the number of components to be clustered increases. Furthermore, the high precision values attained by both schemes demonstrates that the closest component is always retrieved, this being the key reason for organizing the repository.

In scheme A, the center of each cluster is fixed, and is also representative of the cluster's functionality. Because of the fixed center, this implies that perhaps there would be changes in the cluster structure formed when components are clustered (inserted into the repository) in a different order. Thus, we inserted the components into the repository in reverse order to the way they were inserted for the tests performed above. This would result in a different cluster structure formation from before. We then performed exactly the same queries Q_1 and Q_2 as before.

The variation of recall versus T is shown in Figs. 5.9 and 5.10. In the case of Q_1, scheme B greatly outperforms scheme A for all values of T. Compared to the results obtained when the components were inserted in the initial order, scheme A's results degrade by about 15%, in addition to having zero recall for a greater range of T. Scheme B however gives stable results, with the variation of recall being no greater than 10% from when the components were inserted into the repository in the initial order.

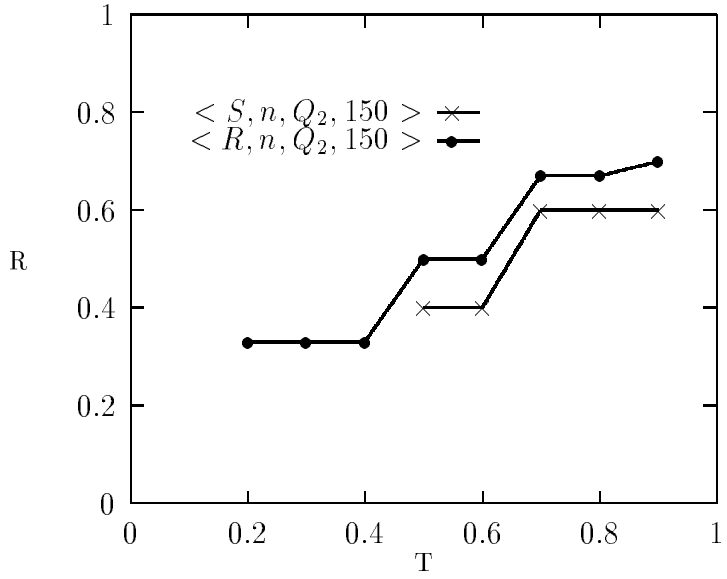

Figure 5.6: Recall (R) vs Threshold (T) for $< S, n, Q_2, 150 >$ and $< R, n, Q_2, 150 >$

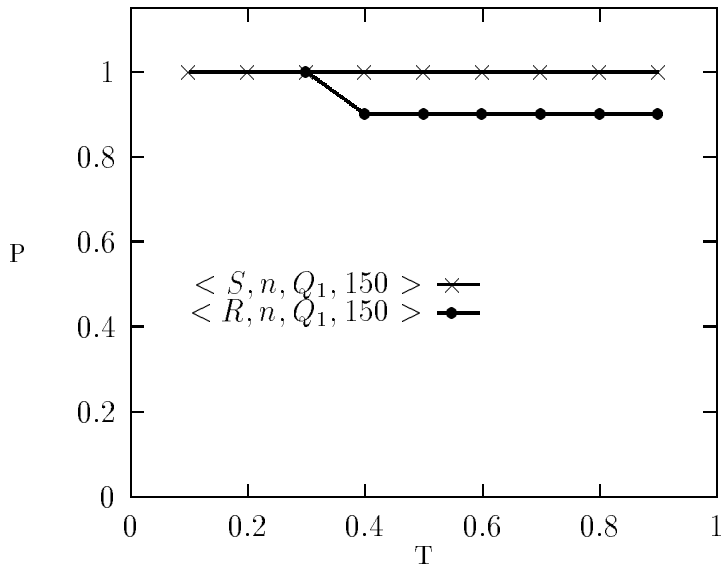

Figure 5.7: Precision (P) vs Threshold (T) for $< S, n, Q_1, 150 >$ and $< R, n, Q_1, 150 >$

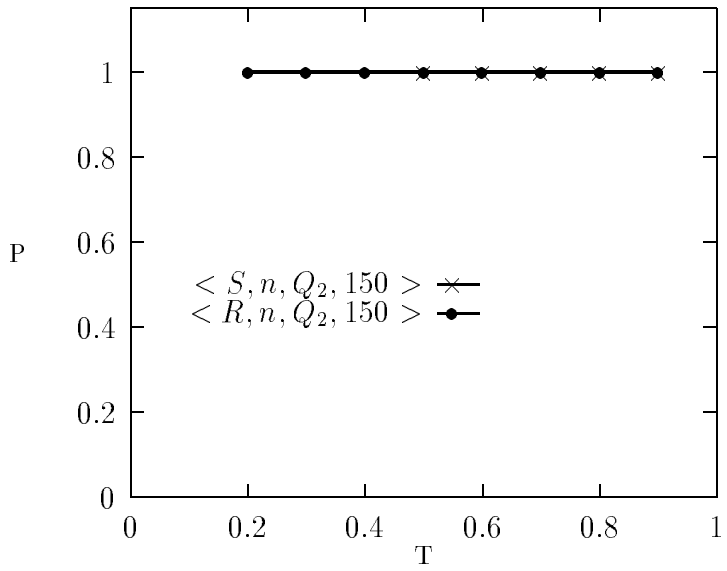

Figure 5.8: Precision (P) vs Threshold (T) for $< S, n, Q_2, 150 >$ and $< R, n, Q_2, 150 >$

Fig. 5.11 shows the precision for Q_1 for both schemes. The zero precision values for scheme A are highly noticeable, whereas the precision values for scheme B are unchanged from when the components were inserted into the repository in the initial order, and remain high at 0.9. In the case of query Q_2 (Fig. 5.12), the precision stays very high at 1.0 for scheme B, but for scheme A, we see a drop in precision of about 45% for values of T=0.8 and 0.9.

The two schemes were also tested when the components were inserted in the reverse order *and* after the repository size was increased by 50%. Fig. 5.13 shows that scheme B outperforms scheme A in recall (for Q_1) for all values of T greater than 0.3. The recall (for T=0.5 to 0.7) is less by no more than 10% compared to the recall before the repository size was increased. For Q_2 (Fig. 5.14), scheme A performs better than scheme B in terms of recall for all values of T despite the fact that scheme B's recall differs by no more than 10% for most values of T -except for T=0.4 when it differs by about 15%- before the repository size was increased (Fig. 5.10).

Precision values for Q_1 and Q_2 after increase in repository size are shown in Figures 5.15 and 5.16 respectively. For both queries, precision for scheme B is unchanged at 0.9 and 1.0 (from before increasing the repository size) for Q_1 and Q_2 respectively. In the case of scheme A, the precision is very high at 1.0 for Q_1 (Fig. 5.15) but falls once again by about 50% for Q_2 (Fig. 5.16).

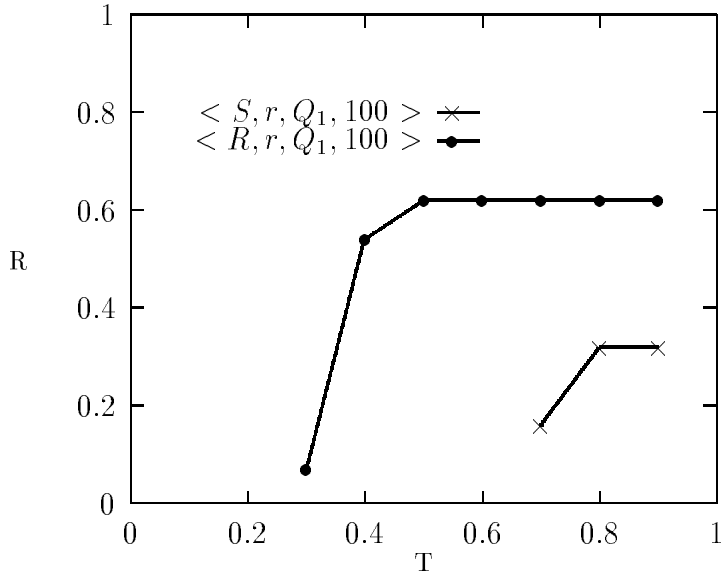

Figure 5.9: Recall (R) vs Threshold (T) for $< S, r, Q_1, 100 >$ and $< R, r, Q_1, 100 >$

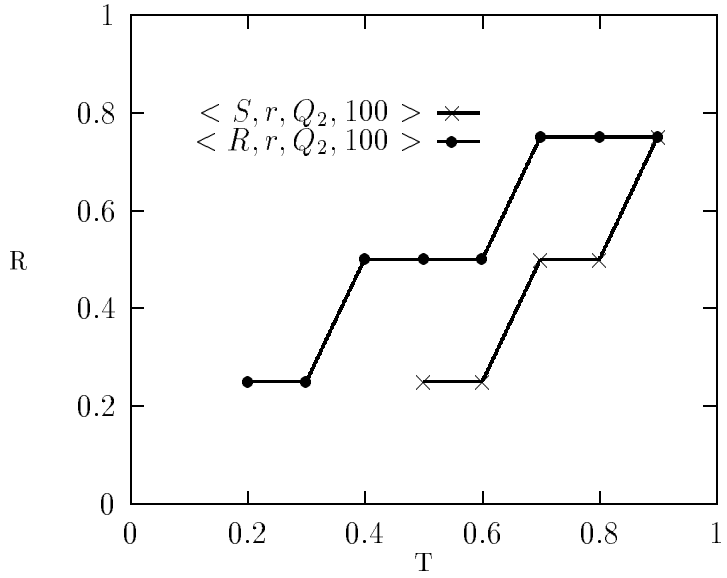

Figure 5.10: Recall (R) vs Threshold (T) for $< S, r, Q_2, 100 >$ and $< R, r, Q_2, 100 >$

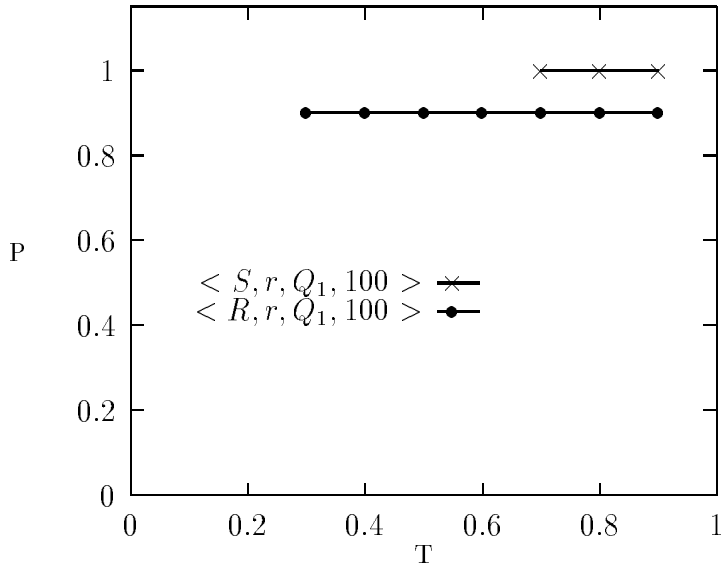

Figure 5.11: Precision (P) vs Threshold (T) for $< S, r, Q_1, 100 >$ and $< R, r, Q_1, 100 >$

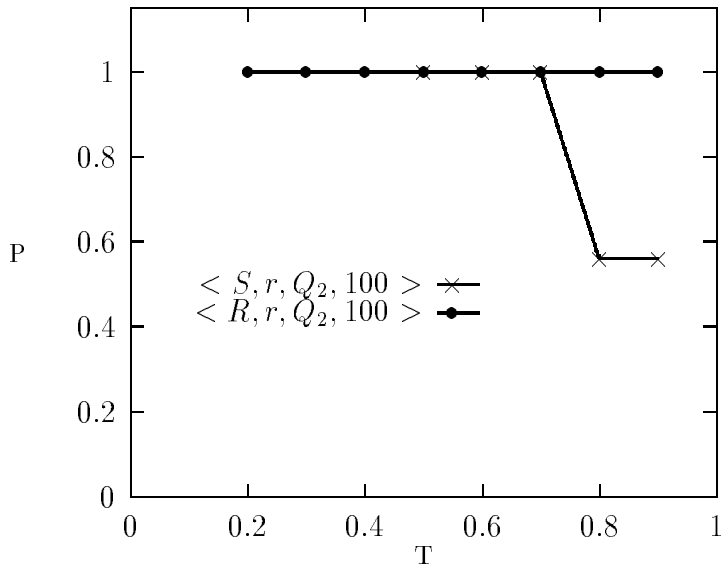

Figure 5.12: Precision (P) vs Threshold (T) for $< S, r, Q_2, 100 >$ and $< R, r, Q_2, 100 >$

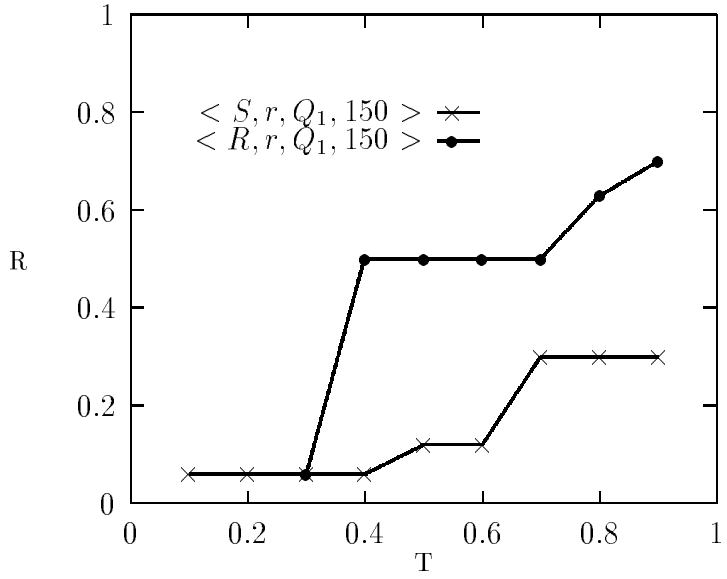

Figure 5.13: Recall (R) vs Threshold (T) for $< S, r, Q_1, 150 >$ and $< R, r, Q_1, 150 >$

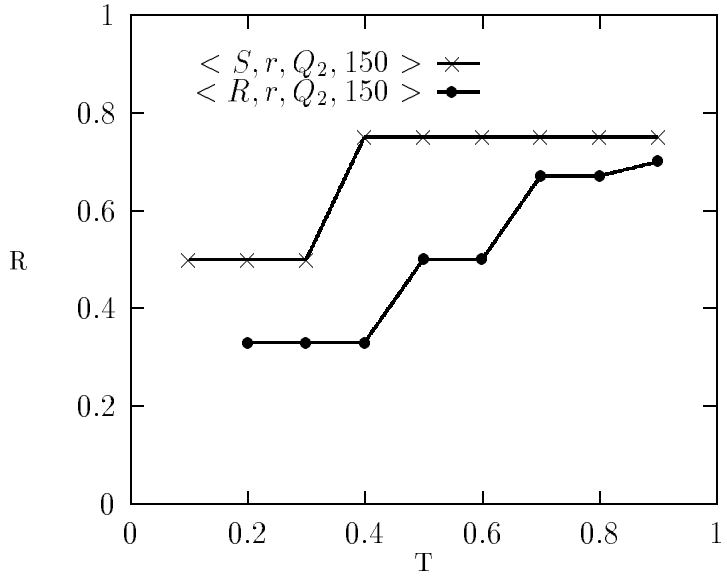

Figure 5.14: Recall (R) vs Threshold (T) for $< S, r, Q_2, 150 >$ and $< R, r, Q_2, 150 >$

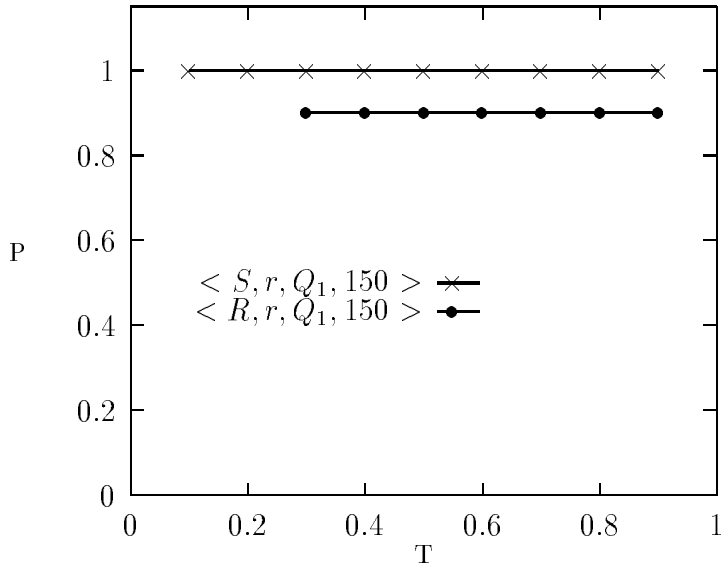

Figure 5.15: Precision (P) vs Threshold (T) for $< S, r, Q_1, 150 >$ and $< R, r, Q_1, 150 >$

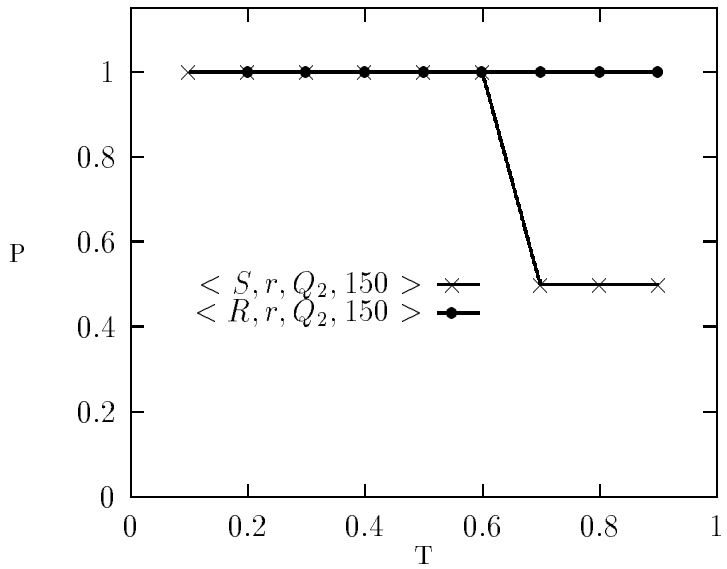

Figure 5.16: Precision (P) vs Threshold (T) for $< S, r, Q_2, 150 >$ and $< R, r, Q_2, 150 >$

The recall and precision results therefore indicate that scheme A is relatively stable when the repository size is increased once the initial components have been inserted, but comparisons with the insertion order of components indicate that the results of precision and recall can vary when the order of insertion of components into the library is changed. For the most part, the scheme's precision results are quite encouraging, and even a 50% drop in precision for a few values of T is a decent result since for most other values of T, precision stays at 1.

The cause of the degradation of some of the precision values, and an absence of recall over a wider range of T values for scheme A after the order of insertion of components was reversed (as compared to the results of the insertion of components in the initial order) can be attributed to having fixed centers as cluster representatives. If a component that becomes a cluster's center will not be a good representative of the other components that are to be inserted into the repository in the future, then this will cause the cluster either not to accept similar components in the future into the cluster unless the T value is high enough, or might allow components to be inserted into the cluster that would not be a close match to the query. This implies that for a query, recall might increase after a certain threshold is crossed, or that the cluster that is identified as having the most similar component will not contain all components that are similar to the query. These results certainly show that recall and precision can vary depending on the cluster structure formation, which in turn varies according to the order of incoming components to be clustered. However, it is an interesting result that for the most part, precision is high. This means that more often than not, the closest components are retrieved. This is significant in terms of how well clustering scheme A works, depicting that it mostly functions well.

For the case of scheme B, the very good precision values for all values of T tells us that the clusters formed for each value of T indeed contain components that are functionally very close to each other. When T is small (less than 0.5), the clusters formed are 'narrow', since no component that varies more than the low threshold from the center of the cluster is allowed into the cluster. Likewise, when the repository is organized with increasing values of T, the clusters become 'wider'. However, the precision value stays the same since the components that fall into the clusters are still functionally very similar. This also proves that the concept of a dynamic cluster center as explained in the clustering scheme works well. Furthermore, the clustering scheme is successful in keeping components that are dissimilar in separate clusters.

The above result is an important one, since the aim of the clustering scheme is to organize components in a manner whereby it should be possible for the reuser

to search for, and retrieve, the conceptually closest component to the query. Our proposed clustering scheme does exactly this.

The results clearly show that the idea of using a dynamic cluster center, as well as a multidimensional space model for the clustering of the repository consistently produce better results. The performance of scheme B over scheme A in terms of robustness is lucid from the results we have obtained through testing. Note that precision for scheme B never falls below 0.9, and for the most part, stays consistently at 1. This result is very encouraging since a key reason for organizing the repository is to be able to retrieve the most similar component to the queried one. Our recall values, especially for scheme B, are also quite good, peaking between 0.62 and 0.76 in all test cases.

5.3.2 Cluster Tightness

In this section, we present results of the cluster tightness tests as explained in the previous section for both schemes. The tests were run with the repository having 150 components (which is its current maximum size).

In Figures 5.17 and 5.18, we show the results of the variation of the average radii over all clusters with respect to the threshold T. The graph of average cluster radius versus T for scheme A in Fig. 5.17 (for components inserted into the repository in the initial order) depicts an increase in the radius as T is increased. This is expected since as T increases, there is greater leeway for the difference in similarity between the center and each component in a cluster. Thus, the average radius is expected to increase with increasing threshold T. Note that the average radius is always smaller than the T value. This clearly shows that the components are clustered around the center in close proximity (in terms of the similarity), at about $1/4$ the respective T value for the majority of T values. This provides us with a snapshot of the cluster formation within the repository.

In Fig. 5.18 for scheme A, when the order of insertion of components is reversed, the graph of the average radius versus T is shown. As is evident from the graph, the average radius increases with T as expected, but remains lower by more than $1/3$ the value of it's respective T value in general. This again shows that the clusters are tight, and that the cluster formation is quite good in terms of a cluster hosting components that are functionally similar to each other.

With respect to cluster tightness, it is important that the average shown for each

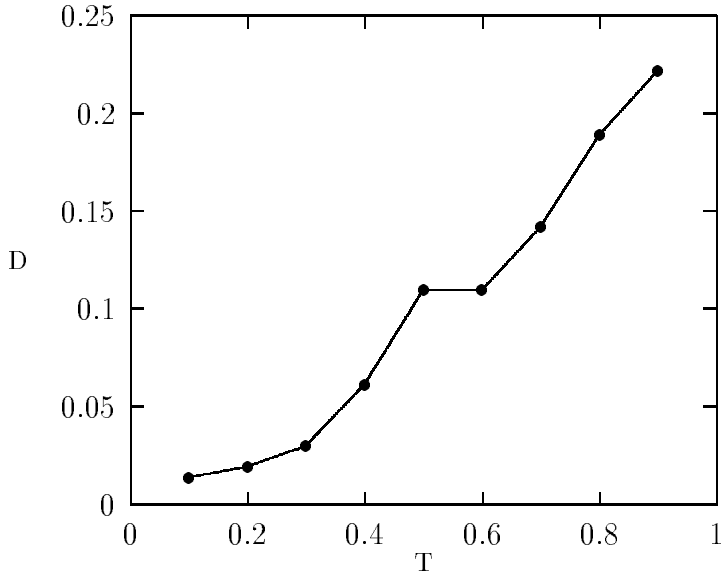

Figure 5.17: Average distance D of components over all clusters vs Threshold T for scheme A for components inserted in the initial order

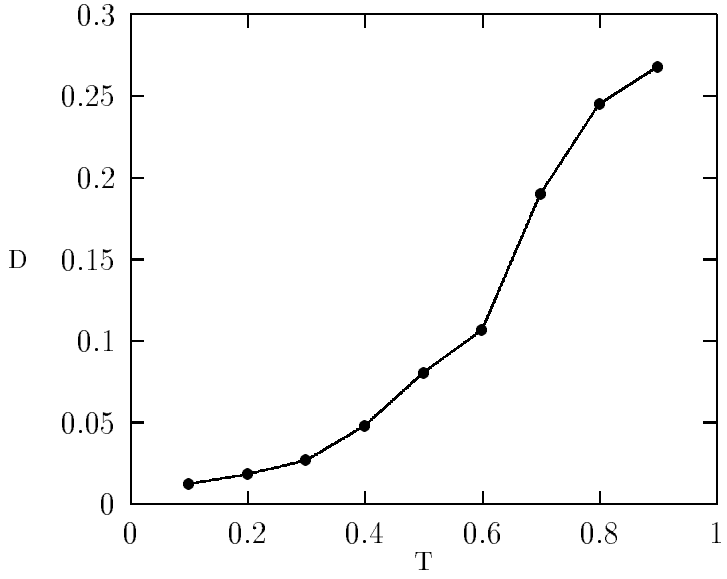

Figure 5.18: Average distance D of components over all clusters vs Threshold T for scheme A for components inserted in reverse order

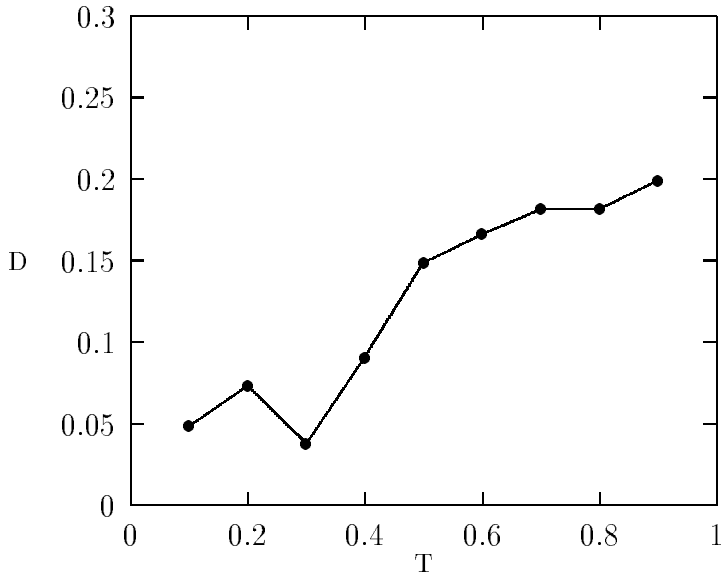

Figure 5.19: Average distance D of components over all clusters vs the Threshold T for scheme B for components inserted in the initial order

value of T does not vary by more than T. In fact, the lower the average distance, the better the clustering since low average distances means that the components are, in general, functionally similar to each other to a larger degree. Fig. 5.19 shows the cluster tightness results for scheme B when the components were inserted into the repository in the initial order. The graph shows that the average distance of components from their clusters is always lower then the respective T value. In fact, the average values are considerably lower than their respective T values, and in most cases, less than the T value by over $1/4T$. Also, the general trend of increasing average distance as T increases is expected, since the clusters formed due to higher T values are 'broader' than those formed when T is small. The dip in the average distance when T is 0.3 is not unnatural, since it is highly possible that for a certain T value, the clusters formed can be optimum in how close the components rally around their clusters' centers.

The results of evaluating the cluster tightness in terms of average radius values over all clusters for scheme B after components were inserted in the reverse order is shown in Fig. 5.20. The graph shows that in most cases, the ratio of T to the average radius over all clusters for that T value is less than $1/4$. This once again proves to us that the components in the clusters are indeed functionally quite similar.

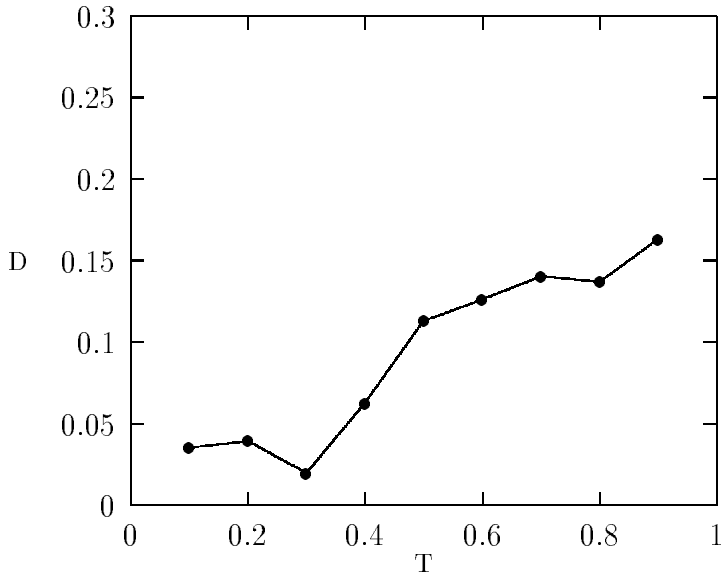

Figure 5.20: Average distance D of components over all clusters vs the Threshold T for scheme B for components inserted in reverse order

Note that Figs. 5.19 and 5.20 both show that at T=0.9, the average cluster radius is less in both Figures than those for scheme A (Figs. 5.17 and 5.18). Low average radii also complements the fact that using the concept of a dynamic cluster center pays off, resulting in superior performance over that of scheme A. Furthermore, it also enables scheme B to perform more robustly over scheme A. This, in terms of cluster structure formation, produces meagre changes in recall and precision when components are inserted into the repository in the reverse order over the results produced by inserting components into the repository in the initial order (as is evident from our experiments for scheme B, for some values of T, insertion of components in the initial order gives slightly better results over insertion in the reverse order, and vice-versa for other values of T).

5.4 Implementation

In this section, we outline the details of the implementation and the computing environment of the prototype tool we have developed that computes the similarity between software components, clusters the repository and retrieves components from the repository when queried by a reuser. The prototype has been implemented in the

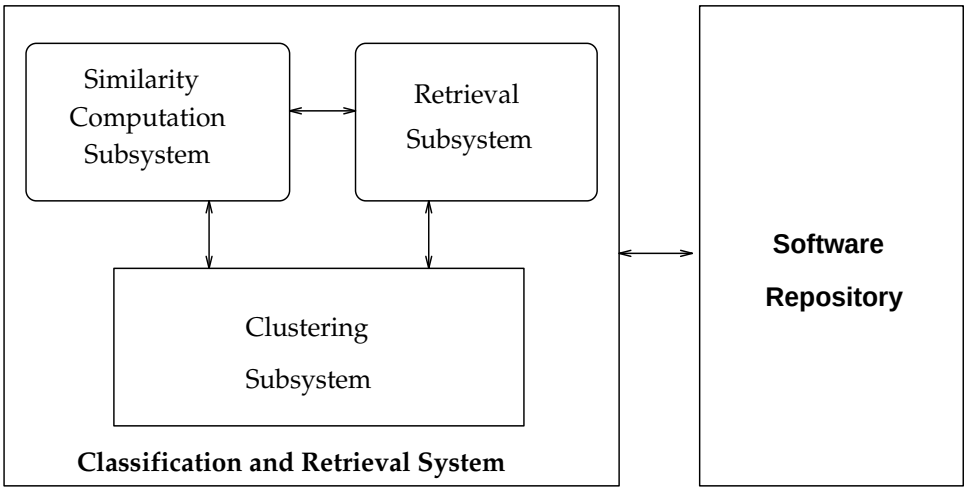

Figure 5.21: Logical View of the Prototype Tool

C language to run on Sun workstations in the Unix operating system environment.

5.4.1 Logical Design

The prototype system was used to obtain the results presented in Chapter 5. A logical view of the prototype is shown in Figure 5.21.

The Similarity Computation Subsystem

The similarity computation subsystem computes the similarity between software components. The method used for doing this is described in Chapter 3. In the case of scheme A, each time a component is to be inserted into the repository, the similarity with respect to the centers of existing clusters is computed. A file containing the centers of current clusters is kept. An index of the cluster numbers (their ids) is also maintained, and updated as new centers are added.

The software components' functional descriptions (FDs) are kept in a file to which components can easily be added, deleted, or modified. This is also the case for the reference components, supporting easy addition of new reference components if and when the domain needs to be expanded. These files are then read and the process of computing similarity then begins. Control is passed over to the similarity computation subsystem each time the similarity between the incoming component and the reference components (in the case of scheme B) or the "fixed" cluster centers (in the case of

scheme A) needs to be computed.

Since the components are inserted into the repository one at a time, the file of FDs that form the current clusters' centers (for scheme A) is updated when new clusters are formed.

The Clustering Subsystem

The clustering subsystem executes the heuristic clustering schemes. The user has the option to cluster the repository using either scheme A or scheme B. The clustering subsystem's modules receives similarity values computed by the similarity computation subsystem for each component and invokes the appropriate modules to cluster the component. In the case of scheme A, after the similarity computation subsystem has computed the similarity of a component with respect to all the centers of existing clusters, the clustering subsystem uses the computed similarities to insert the component into the closest cluster possible (according to scheme A outlined in Section 4.1). For clustering the components according to scheme B, the similarity computation subsystem makes available all similarity values of the components to be clustered with respect to all reference components. The clustering mechanism then inserts the component into the repository according to scheme B described in Section 4.3.2.

The Retrieval Subsystem

After all components are clustered, the reuser can submit a query expressed as a FD. The retrieval subsystem reads this query using the functionalities of the similarity computation subsystem to compute the similarity between the query and the centers formed as a result of clusters formed using scheme A. In the case of scheme B, the similarity with respect to the reference components is also computed using the similarity computation subsystem. Since retrieval follows the method of insertion into the repository by the respective clustering scheme, the retrieval subsystem uses part of the functionalities of the clustering subsystem to identify the best (closest) cluster for insertion of the component to be clustered. Once this is done, instead of inserting the component, the retrieval subsystem informs the reuser as to the cluster whose center is closest in terms of similarity between the components and the cluster center, and all the components of that cluster are retrieved.

Output

After clustering of the repository is completed, the system outputs a complete "picture" of the repository (this is done by writing the relevant information to a file for convenient accessibility to the reuser). This file contains all similarities of each component clustered. For scheme A, these similarities are with respect to the centers of each cluster, whereas for scheme B, it is with respect to each reference component.

Next, the cluster ids and the components hosted by each cluster follow. In the case of scheme B, the final similarity coordinates of each center are also shown. As in case of the clusters, each component is identified by a unique id which is a positive integer starting from zero to the number of total components in the repository.

Chapter 6

Discussion

Our clustering schemes address various shortcomings of previous solutions to the problem of classification and retrieval of software components. In the approaches outlined in [23] [21] [4], a form of the conceptual graph is used to provide a measure of similarity between facets (or features). The construction of a conceptual graph is expensive and tedious. Furthermore, the maintenance of such a graph as more terms or features are added can be cumbersome, since it will demand a restructuring of the whole graph. In our scheme the repository is organized automatically, and places no constraints in the event that more components are added to the repository. This is important since software development is a dynamic process, requiring many changes in the systems developed throughout the software lifecycle.

In [1] [9] [15], information retrieval techniques such as phonetic matching, automatic stemming, and analysis of natural language documentation are used. These information retrieval techniques are relatively hard to implement, and are expensive operations. [15] use a Hierarchical Agglomerative Clustering (HAC) method to organize the library, but their method is dependent on having documentation of software components in Unix-like man pages. Their results show that in the GURU system, when recall is 0.1, the precision value obtained is 0.85; when recall is 0.9, precision is 0.39. Their results cannot be directly compared with ours since their method of similarity computation is different, and furthermore, their test data is also different from ours. On the other hand, [6] points out that hierarchical agglomerative clustering methods often end up doing chaining, which would merely be an ordering of components rather than grouping them according to functional similarity. As is demonstrated by our tests, the second heuristic clustering method that we propose in this thesis is shown to be extremely robust and consistent in its results.

Important savings in using our technique of clustering the library are realized in the form of the complexity of the algorithm. In the HAC method discussed in [25], the pairwise comparison of n components to be clustered takes $n(n-1)/2$ operations. Furthermore, it takes $n^2 \log n^2$ operations to sort the similarity pairs and arrange them in decreasing order of similarity. For scheme A, the complete clustering process takes $n \times m$ steps, while for scheme B, the entire clustering process for n components takes no more than $2 \times n \times k \times m$ steps, where m is the number of resulting clusters, and k is the number of reference components used to define our k-dimensional space. The number k is a constant for any one domain (since the reference components are set as a subset of the component descriptions). Note that prior to the completion of the entire clustering process, the number of clusters is normally less than their final number m. After testing, we observed that on average, m is between $n/3$ to $n/2$, and our scheme B performs between $k.n^2/3$ and $k.n^2/2$ comparisons. Thus, our technique (scheme B) has considerable savings, while the savings are even greater in the case of scheme A. This would be greatly realized when the repository is to contain a large number of components, especially in an industrial setting.

Our clustering scheme uses a knowledge representation language structure for describing the functionality of components, and is independent of the documentation that might be provided with the software components. We also argue that the functionality of components is better represented by a knowledge representation language, especially one designed for representing information about information systems [18].

Our model for the software reuse environment has been documented extensively, and has been used widely in software reuse research targeted toward industry[29], [10], [16], [28], [7]. Thus, the librarian's task of constructing FDs would be highly intertwined with managing the library of reusable components. In the case of inter-company reuse -as is the aim of the REBOOT project- the librarian could have responsibility for maintaining libraries for various domains and as such, managing the repositories for the various domains would be a vital task.

When making queries to the repository, the reuser has to construct a query as a FD. We believe that it would be quite easy for the reuser to make a query since the FDs are simple and easy to construct, and the reuser need not know exact figures to insert as weights. The simple mapping of the weights into abbreviations such as H (for high) or L (low) to represent functionalities of various features within FDs would be easier than to express queries in terms of reuse metrics, where sometimes, more exactness is required in the absence of a simple and non-complicated mapping system.

A definite advantage with our clustering schemes is that they are implemented apart from the similarity computation method, that is, there is very low coupling between these two subsystems in their implementation. This implies that one could easily replace the current similarity computation method with another similarity computation method, and in this case, then our clustering scheme would still function with minor modifications required to the clustering subsystem.

Chapter 7

Conclusion

In this thesis, we presented two heuristic clustering schemes for organizing software libraries for software reuse.

The organization of the repository is done *automatically*, its structure is transparent to the software reuser, and places no constraint in the event of incremental changes. Our schemes use a knowledge representation language structure for describing the functionality of components, and therefore, do not rely on any special component description (as in [15]). The effectiveness of the proposed scheme was tested on a software collection in terms of recall and precision. Our experiments demonstrate that while our sphere-packing scheme performs at a good precision level for the most part, our multidimensional scheme attains a respectable recall value and a high precision value. While the cluster tightness experiments do not directly reflect on the precision values obtained through testing, they show that the clusters formed are indeed tight, and that components do cluster close to the centers. This implies that components are clustered with respect to the centers in terms of relatively high functional similarity (within the threshold set by the repository organizer). Results from our experiments demonstrate that the proposed multidimensional clustering scheme is stable and robust under conditions of variation in repository size, as well as the order of insertion of components into the repository. Our clustering schemes addresses various shortcomings of previous solutions to the problem of classification and retrieval of software components. Specifically, it does not require the manual construction of a conceptual graph as outlined in [23], [21], [4].

7.1 Future Work

There are many other issues yet to be dealt with in software reuse. We have compiled the following list which is by no means an exhaustive one, and outline some of the technical areas which are worth investigating.

1. Reuse software specifications as well as software designs.
2. Reuse of knowledge, especially knowledge relating to domain analysis.
3. Effective operation of multiple repositories.
4. Adaptation of software components.

We believe that the process of reusing specifications can follow the same approach as described in this thesis, since as long as the specifications are represented using a representation language, a repository comprising specification components can be organized using our clustering schemes. For the reuse of domain analysis knowledge, a key problem is that each domain analysis approach taken needs to be tailored for the particular domain. A formalization of the techniques used for domain analysis could be used as a tool for other domain analysis approaches. In the case of the interoperation of software repositories, we believe that representing knowledge using a knowledge representation language could form a representation standard, easing the problems due to varying component exchange standards between repositories. Lastly, if the classification and retrieval process of software components as described in this thesis could also be done for other software artifacts (where a repository would exist for each type of artifact), then the clustering schemes we describe in this thesis could be enhanced and adapted to combine similar software artifacts from the various repositories. This would provide the reuser with a complete software lifecycle of software artifacts. Functionally similar designs, architectures, code components, as well as test cases (all these being the artifacts for which multiple repositories could be maintained) would then be available for the development of a new system. This would further assist application developers in their modification and reuse of not only software components but all artifacts in general, while lowering the cost of the software development process.

Bibliography

[1] S. Arnold and S. Stepoway. The REUSE system: Cataloging and retrieval of reusable software. In *Proceedings of COMPCON '87*, pages 376–379, 1987.

[2] J. Capers. Software challenges: Economics of software reuse. *IEEE Software*, July 1994.

[3] E. Charniak, C. Riesbeck, D. McDermott, and J. Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum, 1987.

[4] W.G. Cho, Y.W. Kim, and J.H. Kim. CLIS: A software reuse library system with a knowledge based information retrieval model. In *Proceedings of the Pacific Rim International Conference on AI*, pages 402–407, 1990.

[5] M. D'Alessandro, P. Iachini, and A. Martelli. The generic reusable component: an approach to reuse hierarchical OO designs. In *Proceedings of the 2nd International Workshop on Software Reusability, March 24-26, Lucca, Italy*, pages 39–46, 1993.

[6] B. Everitt. *Cluster Analysis*. Halsted Press, 1993.

[7] J. Faget and J.-M. Morel. The REBOOT approach to the concept of a reusable component. In M. Griss and L. Latour, editors, *Proceedings of the 5th Annual Workshop on Software Reuse*. Dept of Computer Science, University of Maine, 1992.

[8] S. Faustle and M.G. Fugini. Retrieval of reusable components using functional similarity. Technical Report ITHACA.POLIMI.E.6.92, University of Pavia, 1992.

[9] W. Frakes and B. Nejmeh. An information system for software reuse. In W. Tracz, editor, *IEEE Tutorial: Software Reuse: Emerging Technology*. IEEE Computer Society Press, 1988.

[10] M. Fugini and S. Faustle. Retrieval of reusable components in a development information system. In *Proceedings of 3rd International Workshop on Software reusability IWSR-3*, pages 89–98, 1993.

[11] M. L. Griss. Software reuse at Hewlett-Packard. In W. Frakes, editor, *Proceedings of the 1st International Workshop on Software Reusability, Dortmund, July 3-5.* 1991.

[12] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2), June 1992.

[13] R. Kruse. *Data Structures and Program Design in C, 2nd Edition.* Prentice-Hall, 1991.

[14] Software Engineering Laboratory. *Proceedings of the Sixteenth Annual NASA/Goddard Software Engineering Technology*, December 1991.

[15] Yoelle Maarek, Daniel Berry, and Gail Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Software Engineering*, 17(8):800–813, August 1991.

[16] M.Fugini, O.Nierstrasz, and B.Pernici. Application development through reuse: the Ithaca tools enviroment. *ACM SIGOIS Bulletin*, August 1992.

[17] W. Myers. Workshop explores large-grained reuse. *IEEE Software*, pages 108–109, January 1994.

[18] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: representing knowledge about information systems. *ACM Trans. Information Systems*, October 1990.

[19] P. Nelson and A. Toptsis. Search space clustering in parallel bidirectional heuristic search. In *Proceedings of 4th UNB AI Symposium*, pages 563–573, 1991.

[20] United States General Accounting Office. *Issues Facing Software Reuse, Report GAO/IMTEC-93-16,B-251542.* Information Management and Technology Division, Washington, DC, 1993.

[21] E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun. Computing similarity in a reuse library system: an AI-based approach. *ACM Transactions on Software Engineering and Methodology*, pages 205–228, July 1992.

[22] R. Prieto-Diaz. *Communications of the ACM*, 34(5):88–97, May 1990.

[23] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[24] Ruben Prieto-Diaz. Domain analysis for reusability. In R. Prieto-Diaz and G. Arango, editors, *Domain analysis and software systems modeling*. IEEE Computer Society Press, 1991.

[25] G. Salton. *Automatic Text Processing*. Addison-Wesley, 1989.

[26] G. Salton, J. Allan, and C. Buckley. Automatic structuring and retrieval of large text files. *Communications of the ACM*, 37(2), February 1994.

[27] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[28] G. Sindre, E. Karlsson, and T. Staalhane. Organizing large libraries of reusable components: the REBOOT approach. *Journal of Software Engineering and Knowledge Engineering*, April 1992.

[29] G. Sindre, E. Karlsson, and T. Staalhane. A method for software reuse through large component libraries. In *Proceedings of the International Conference on Computing and Information*, pages 464–468, 1993.

[30] J. R. Tirso. The IBM reuse program. In L. Latour, editor, *Proceedings of the 4th Annual Workshop on Software Reuse*. Dept of Computer Science, University of Maine, 1991.

Appendix A

Component Functional Descriptions

```
insert-node L
delete-node VL
%
move_right-btree L
move_left-btree M
%
push-element M
pop-element M
%
split-btree M
restore-btree L
%
size-array VL
size-queue L
%
combine-btree VL
%
traverse-list M
%
top-stack M
pop-stack L
%
empty-queue L
```

```
full-queue VL
%
depthfirst-graph M
breadthfirst-graph VL
%
push_down-btree M
push_in-btree L
%
traverse-tree M
end-tree VL
%
merge-list L
divide-list M
%
dequeue-element VH
%
type_queue VL
insert-queue M
%
insert-heap L
build-heap H
%
initialize-queue M
initialize-stack M
%
insert-hashtable M
retrieve-hashtable L
%
swap-pointers M
type-queue H
%
sort-graph VH
distance-graph VL
%
insert-node VL
delete-node H
%
split-btree M
```

```
restore-btree VL
%
next-element H
first-element M
%
size-array H
push-element H
%
predecessor-element H
successor-element H
%
dereference-pointer L
swap-pointers L
%
top-stack L
pop-stack H
%
merge-list H
divide-list H
%
sort-list H
maxkey-list L
%
insert-hashtable H
hash-value L
%
traverse-list H
%
push_down-btree VH
push_in-btree VL
%
type-queue M
insert-queue L
%
traverse-tree M
%
delete-element H
insert-element VL
```

```
%
restore-btree H
type-queue H
%
initialize-queue L
initialize-stack H
%
insert-heap VL
build-heap M
%
sort-graph M
distance-graph M
%
check-null H
malloc-stack M
%
empty-queue L
full-queue L
%
split-btree M
swap-pointers L
%
combine-btree M
%
push-element VL
pop-element L
%
merge-list H
divide-list L
%
size-array L
restore-btree H
%
insert-hashtable M
hash-value M
%
insert-node M
delete-node M
```

```
%
depthfirst-graph L
breadthfirst-graph L
%
size-array M
size-queue M
%
empty-queue VH
full-queue L
%
push_down-btree H
%
sort-list VH
maxkey-list VL
%
free-memory VL
malloc-stack M
%
type_queue H
swap-pointers M
%
initialize-stack M
push-stack H
%
insert-heap VL
build-heap VH
%
next-element VH
first-element M
%
merge-list L
divide-list L
%
split-btree M
restore-btree VL
%
sort-graph VH
%
```

```
type_queue M
insert-queue M
%
check-null L
%
make-node H
delete-node M
%
traverse-tree M
end-tree L
%
sort-list H
malloc-stack L
maxkey-list VL
%
move_right-btree M
move_left-btree M
%
hash-value M
%
empty-queue VH
full-queue VL
%
delete-element VH
insert-element VL
%
split-btree M
push_in-btree M
%
retrieve-hashtable M
insert-hashtable VH
%
top-stack M
pop-stack M
%
split-btree M
restore-btree L
%
```

```
insert-node L
delete-node M
%
initialize-queue L
initialize-stack L
push-stack H
%
predecessor-element L
successor-element H
%
free-memory L
malloc-stack M
%
sort-graph M
distance-graph M
%
type_queue M
insert-queue L
%
size-array L
size-queue L
%
delete-element L
insert-element L
dequeue-element M
%
push-element H
pop-element M
%
depthfirst-graph VL
breadthfirst-graph VL
%
insert-hashtable M
hash-value L
%
traverse-tree H
end-tree L
%
```

```
insert-heap VL
build-heap M
%
move_right-btree M
move_left-btree VL
%
split-btree M
restore-btree L
%
empty-queue VH
full-queue L
%
check-null L
malloc-stack M
%
split-btree L
push_in-btree H
%
make-node H
delete-node L
%
bin_search-tree M
%
merge-list H
divide-list L
%
insert-hashtable H
retrieve-hashtable L
%
predecessor-element VH
successor-element H
%
combine-btree VH
size-stack L
%
dereference-pointer L
swap-pointers M
%
```

```
sort-graph H
distance-graph M
%
hash-value VH
%
size-array H
size-queue H
%
insert-node L
delete-node H
%
type-queue VH
insert-queue H
%
split-btree M
restore-btree L
%
push-element M
pop-element L
%
top-stack H
pop-stack H
%
next-element L
first-element M
%
bin_search-tree VH
%
traverse-tree L
%
check-null H
malloc-stack H
%
empty-queue L
full-queue L
%
traverse-tree H
size-queue H
```

```
%
free-memory VL
malloc-stack M
%
make-node VH
delete-node L
%
swap-pointers M
%
type-queue VH
insert-queue VH
%
predecessor-element M
successor-element H
%
depthfirst-graph M
breadthfirst-graph L
%
insert-node VL
delete-node M
%
type_queue VL
insert-queue VL
%
sort-graph M
distance-graph H
%
split-btree L
restore-btree L
%
insert-node M
delete-node VL
%
retrieve-hashtable H
insert-hashtable VH
%
top-stack M
pop-stack L
```

```
%
push_in-btree H
%
combine-btree L
%
check-null H
malloc-stack M
%
empty-queue M
full-queue L
%
next-element L
first-element L
%
traverse-tree M
end-tree L
%
bin_search-tree H
%
make-node M
delete-node L
%
move_right-btree M
move_left-btree M
%
empty-queue M
full-queue M
%
free-memory M
malloc-stack M
%
insert-node L
delete-node M
%
split-btree VL
restore-btree VL
%
type-queue H
```

```
insert-queue L
%
sort-graph VH
distance-graph L
%
move_right-btree M
delete-node L
%
full-queue M
next-element L
%
next-element L
first-element L
traverse-list H
%
top-stack L
size-stack M
%
insert-queue H
size-queue L
%
push_down-btree H
push_in-btree L
%
bin_search-tree H
%
sort-list M
maxkey-list M
%
merge-list M
divide-list M
%
```

Appendix B

The Reference Components

```
depthfirst-graph H
breadthfirst-graph H
%
insert-node M
delete-node M
%
move_right-btree VH
move_left-btree M
%
delete-element VH
insert-element VL
dequeue-element VH
%
free-memory M
malloc-stack VH
%
push-element VL
pop-element M
%
combine-btree H
size-stack M
%
size-array VH
size-queue VH
%
```

```
dereference-pointer H
swap-pointers M
%
empty-queue VH
full-queue H
%
next-element L
first-element M
%
predecessor-element VH
successor-element H
%
split-btree M
restore-btree M
%
top-stack L
pop-stack VL
%
check-null H
%
traverse-tree H
end-tree M
%
type-queue M
insert-queue M
%
retrieve-hashtable M
insert-hashtable M
%
insert-heap L
build-heap H
%
push_down-btree L
push_in-btree L
%
sort-graph H
distance-graph L
%
```

```
initialize-queue L
initialize-stack M
push-stack H
%
traverse-list H
%
bin_search-tree H
%
hash-value H
%
sort-list H
maxkey-list M
%
merge-list M
divide-list M
%
```

Appendix C

Values for Recall and Precision

$< S, n, Q_1, 100 >$

T	R
0.1	0.058
0.2	0.058
0.3	0.058
0.4	0.058
0.5	0.058
0.6	0.058
0.7	0.31
0.8	0.31
0.9	0.31

$< S, n, Q_2, 100 >$

T	R
0.5	0.25
0.6	0.25
0.7	0.75
0.8	0.75
0.9	0.75

$< S, n, Q_1, 100 >$

T	P
0.1	1
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< S, n, Q_2, 100 >$

T	P
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< S, n, Q_1, 150 >$

T	R
0.1	0.058
0.2	0.058
0.3	0.058
0.4	0.058
0.5	0.058
0.6	0.058
0.7	0.058
0.8	0.411
0.9	0.411

$< S, n, Q_2, 150 >$

T	R
0.5	0.4
0.6	0.4
0.7	0.6
0.8	0.6
0.9	0.6

$< S, n, Q_1, 150 >$

T	R
0.1	1
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< S, n, Q_2, 150 >$

T	R
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< S, r, Q_1, 100 >$

T	R
0.7	0.16
0.8	0.32
0.9	0.32

$< S, r, Q_2, 100 >$

T	R
0.5	0.25
0.6	0.25
0.7	0.5
0.8	0.5
0.9	0.75

$< S, r, Q_1, 100 >$

T	P
0.7	1
0.8	1
0.9	1

$< S, r, Q_2, 100 >$

T	P
0.5	1
0.6	1
0.7	1
0.8	0.56
0.9	0.56

$< S, r, Q_1, 150 >$

T	R
0.1	0.06
0.2	0.06
0.3	0.06
0.4	0.06
0.5	0.12
0.6	0.12
0.7	0.3
0.8	0.3
0.9	0.3

$< S, r, Q_2, 150 >$

T	R
0.1	0.5
0.2	0.5
0.3	0.5
0.4	0.75
0.5	0.75
0.6	0.75
0.7	0.75
0.8	0.75
0.9	0.75

$< S, r, Q_1, 150 >$

T	P
0.1	1
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< S, r, Q_2, 150 >$

T	P
0.1	1
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	0.5
0.8	0.5
0.9	0.5

$< R, n, Q_1, 100 >$

T	R
0.3	0.067
0.4	0.53
0.5	0.53
0.6	0.6
0.7	0.67
0.8	0.67
0.9	0.67

$< R, n, Q_2, 100 >$

T	R
0.2	0.25
0.3	0.25
0.4	0.25
0.5	0.5
0.6	0.5
0.7	0.5
0.8	0.75
0.9	0.75

$< R, n, Q_1, 100 >$

T	P
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< R, n, Q_2, 100 >$

T	P
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< R, n, Q_1, 150 >$

T	R
0.3	0.06
0.4	0.53
0.5	0.53
0.6	0.6
0.7	0.65
0.8	0.65
0.9	0.7

$< R, n, Q_2, 150 >$

T	R
0.2	0.33
0.3	0.33
0.4	0.33
0.5	0.5
0.6	0.5
0.7	0.67
0.8	0.67
0.9	0.7

$< R, n, Q_1, 150 >$

T	P
0.3	1
0.4	0.9
0.5	0.9
0.6	0.9
0.7	0.9
0.8	0.9
0.9	0.9

$< R, n, Q_2, 150 >$

T	P
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< R, r, Q_1, 100 >$

T	R
0.3	0.07
0.4	0.54
0.5	0.62
0.6	0.62
0.7	0.62
0.8	0.62
0.9	0.62

$< R, r, Q_2, 100 >$

T	R
0.2	0.25
0.3	0.25
0.4	0.5
0.5	0.5
0.6	0.5
0.7	0.75
0.8	0.75
0.9	0.75

$< R, r, Q_1, 100 >$

T	P
0.3	0.9
0.4	0.9
0.5	0.9
0.6	0.9
0.7	0.9
0.8	0.9
0.9	0.9

$< R, r, Q_2, 100 >$

T	P
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

$< R, r, Q_1, 150 >$

T	R
0.3	0.06
0.4	0.5
0.5	0.5
0.6	0.5
0.7	0.5
0.8	0.63
0.9	0.7

$< R, r, Q_2, 150 >$

T	R
0.2	0.33
0.3	0.33
0.4	0.33
0.5	0.5
0.6	0.5
0.7	0.67
0.8	0.67
0.9	0.7

$< R, r, Q_1, 150 >$

T	P
0.3	0.9
0.4	0.9
0.5	0.9
0.6	0.9
0.7	0.9
0.8	0.9
0.9	0.9

$< R, r, Q_2, 150 >$

T	P
0.2	1
0.3	1
0.4	1
0.5	1
0.6	1
0.7	1
0.8	1
0.9	1

Values for Fig. 5.17

T	D
0.1	0.013
0.2	0.019
0.3	0.03
0.4	0.06
0.5	0.11
0.6	0.11
0.7	0.141
0.8	0.189
0.9	0.22

Values for Fig. 5.18

T	D
0.1	0.012
0.2	0.018
0.3	0.026
0.4	0.047
0.5	0.08
0.6	0.106
0.7	0.19
0.8	0.245
0.9	0.268

Values for Fig. 5.19

T	D
0.1	0.048
0.2	0.073
0.3	0.037
0.4	0.09
0.5	0.148
0.6	0.166
0.7	0.181
0.8	0.181
0.9	0.199

Values for Fig. 5.20

T	D
0.1	0.035
0.2	0.039
0.3	0.019
0.4	0.062
0.5	0.113
0.6	0.126
0.7	0.14
0.8	0.137
0.9	0.163