NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

Program Restructuring Using Clustering Techniques

by

Xia Xu, M.Eng.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements of the degree of Master of Science
in Information and Systems Science

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
K1S 5B6 Canada

May 2004

@Copyright 2004, Xia Xu

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-93993-6

Our file Notre référence

ISBN: 0-612-93993-6

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

As software evolves, its structure tends to be degraded and it becomes hard to maintain. Program restructuring is a key method for improving an ill-structured program, thereby increasing understandability and reducing maintenance cost. This thesis presents an approach to program restructuring at the function level, based on the clustering technique with cohesion as a major criterion. The approach focuses on automated support for identifying ill-structured low-cohesive functions written in C, and providing heuristic restructuring advice in both the development and evolution phases. A new similarity measure is defined and studied intensively. In the case study, the proposed approach is used to restructure a real industrial program. The result shows that the clustering analysis based on the new similarity measure defined in this thesis can pick out high-cohesive functions from low-cohesive functions, and reveal existing problems in the code. The comparison before and after restructuring exhibits measurable improvement.

Acknowledgements

I would like to thank my co-supervisors, Professors Chung-Horng Lung and Dorina C. Petriu, for their valuable guidance, thoughtful advice and continuous support.

I would like to acknowledge the support of NCIT and Eion Inc. for providing funding and a real industrial environment for this research. I would also like to thank Marzia Zaman at Cistel Technology for providing me the initial tool and comments during the study.

Finally, I am very grateful to my friends and my family for their direct or indirect contribution to this thesis. Special thanks go to my husband, Yuanming, and our son, Gordon, for their endless support and encouragement.

Table of Contents

Chapter 1 Introduction.....	1
1.1 Problems in software development and maintenance	1
1.2 Motivation.....	2
1.3 Goals and contributions of the thesis	4
1.4 Thesis organization	5
Chapter 2 Background	7
2.1 Overview of software lifecycle	7
2.1.1 Forward engineering	8
2.1.2 Reverse engineering	9
2.1.3 Reengineering.....	9
2.1.4 Restructuring	9
2.2 Software design quality and maintainability	10
2.2.1 Component-level design metrics.....	10
2.2.1.1 Cohesion and coupling	10
2.2.1.2 Complexity	11
2.2.2 Maintainability	12
2.3 Overview of clustering techniques.....	12
2.3.1 Entities and attributes	13
2.3.2 Similarity measures	14
2.3.2.1 Distance measures	14
2.3.2.2 Association coefficients.....	15
2.3.2.3 Correlation coefficients	16
2.3.2.4 Probabilistic similarity measures.....	17
2.3.3 Clustering algorithms	17
2.3.3.1 Hierarchical algorithms	17
2.3.3.2 Non-hierarchical algorithms	19
2.4 Related work	21
2.4.1 Related work on program restructuring.....	21

2.4.2 Related work on software clustering	24
2.5 Scope of the thesis.....	27
Chapter 3 An Approach to Program Restructuring Using Clustering Techniques ..30	
3.1 Restructuring objective	30
3.2 Program restructuring approach.....	31
3.3 Data collection and processing	33
3.3.1 Entities.....	33
3.3.2 Attributes.....	36
3.3.3 Scale of attributes	39
3.4 Similarity measure	41
3.4.1 Issues with weighting of attributes.....	42
3.4.2 Issues with weighting of matches.....	44
3.4.3 Resemblance coefficient.....	46
3.5 Visualization and analysis.....	48
3.5.1 Clustering tree	48
3.5.2 Tree analysis.....	50
3.6 Restructuring process.....	52
Chapter 4 Experiments on Similarity Measure55	
4.1 Jaccard coefficient and Sorenson coefficient.....	55
4.2 Experiment with weight ratio of 2:1	58
4.3 Experiment with weight ratio of 3:1	61
4.4 Experiment with weight ratio of 9:4	65
4.5 Experiment with weight ratio of 7:3	66
4.6 Experiment with weight ratio of 5:2	67
4.7 Experiment with weight ratio of 8:3	69
4.8 Experiment with data attributes in control entities	71
Chapter 5 Case Study75	
5.1 Background of case study	75
5.2 Restructuring process.....	77
5.3 Results of restructuring	78
5.4 Empirical observations.....	82

5.4.1 Functional clusters.....	82
5.4.2 Duplicated code.....	83
5.4.3 Interleaved code	84
5.4.4 Cut-point.....	84
5.4.5 Comparison between different clustering algorithms	85
5.4.6 Limitations	88
Chapter 6 Conclusions.....	92
6.1 Summary	92
6.2 Future work	94
References	95

List of Figures

Figure 2-1 Relationship between terms [Chi90]	7
Figure 3-1. An approach to program restructuring	31
Figure 3-2. Sample code 1: Sale, Pay and Profit [Lak99]	35
Figure 3-3. Example of a clustering tree.....	49
Figure 3-4. Restructuring result from <i>cut2</i> for sample code 1 in Figure 3-2	53
Figure 3-5. Restructuring result from <i>cut3</i> for sample code 1 in Figure 3-2	54
Figure 4-1. Tree obtained with Jaccard coefficient for sample code 1 in Figure 3-2	57
Figure 4-2. Tree obtained with Sorenson coefficient for sample code 1 in Figure 3-2	57
Figure 4-3. Tree obtained with a 2:1 weight ratio for sample code 1 in Figure 3-2	59
Figure 4-4. Sample code 2: Sum and Product [Bie98]	60
Figure 4-5. Tree obtained with a 2:1 weight ratio for sample code 2 in Figure 4-4	60
Figure 4-6. Tree obtained with a 3:1 weight ratio for sample code 1 in Figure 3-2	62
Figure 4-7. Tree obtained with a 3:1 weight ratio for sample code 2 in Figure 4-4	62
Figure 4-8. Sample code 3: Process Body	64
Figure 4-9. Tree obtained with a 3:1 weight ratio for sample code 3 in Figure 4-8	65
Figure 4-10. Tree obtained with a 9:4 weight ratio for sample code 1 in Figure 3-2	65
Figure 4-11. Tree obtained with a 9:4 weight ratio for sample code 2 in Figure 4-4	66
Figure 4-12. Tree obtained with a 7:3 weight ratio for sample code 1 in Figure 3-2	66
Figure 4-13. Tree obtained with a 7:3 weight ratio for sample code 2 in Figure 4-4	67
Figure 4-14. Tree obtained with a 5:2 weight ratio for sample code 1 in Figure 3-2	67
Figure 4-15. Tree obtained with a 5:2 weight ratio for sample code 2 in Figure 4-4	68
Figure 4-16. Tree obtained with a 5:2 weight ratio for sample code 3 in Figure 4-8	69
Figure 4-17. Tree obtained with an 8:3 weight ratio for sample code 1 in Figure 3-2	69
Figure 4-18. Tree obtained with an 8:3 weight ratio for sample code 2 in Figure 4-4	70
Figure 4-19. Tree obtained with an 8:3 weight ratio in sample code 3 in Figure 4-8.....	70
Figure 4-20. Tree obtained with a state value of 2 for data attributes in control entities in sample code 1 in Figure 3-2.....	72
Figure 4-21. Tree obtained with a state value of 2 for data attributes in control entities in sample code 2 in Figure 4-4.....	73

Figure 5-1. Partial tree of <i>rsvpTeRxRErr</i>	83
Figure 5-2. Partial tree of <i>rsvpTeProcessERO</i>	83
Figure 5-3. Partial tree of <i>rsvpTeRxPTear</i>	84
Figure 5-4. Comparison of three algorithms for <i>rsvpTeRxResv</i>	86
Figure 5-5. Example of non-functional clusters	88
Figure 5-6. Example of singleton clusters	89
Figure 5-7. Example of big data structure	90
Figure 5-8. Example of one variable related to more functionalities	91

List of Tables

Table 3-1. Entities and their attributes of sample code 1 in Figure 3-2.....	39
Table 3-2. Entity-attribute matrix of sample code 1 in Figure 3-2	41
Table 4-1. Entity-attribute matrix of sample code 1 in Figure 3-2 in case where data attributes and control attributes are treated equally	56
Table 4-2. Entity-attribute matrix of sample code 2 in Figure 4-4	60
Table 4-3. Entity-attribute matrix of sample code 1 in Figure 3-2 with a state value of 2 for data attributes in control entities	72
Table 4-4. Entity-attribute matrix of sample code 2 in Figure 4-4 with a state value of 2 for data attributes in control entities	73
Table 5-1. Selected functions for restructuring analysis.....	76
Table 5-2. Size, complexity and cohesion comparisons before and after restructuring	80
Table 5-3. Comparison summary before and after restructuring	82
Table 5-4. Comparison between three clustering algorithms	87

Chapter 1 Introduction

1.1 Problems in software development and maintenance

In the lifecycle of software, software evolution usually accounts for more than 60% of total software costs [Som96]. Müller et al. [Mül95] indicate that 50-90% of software evolution work focuses on program comprehension. Ill-structured programs usually contain some functions involving multiple activities, and are therefore hard to understand. A poorly designed program has a bad starting point with regard to its evolution, and the evolution process is made even more difficult. Even when a program is well-designed, its structure also tends to be degraded as it evolves.

In real-world software development, software products are usually driven by tight schedules. Hence software engineers often emphasize more the functionality provided by the products than the non-functionality quality attributes. In addition, in order to respond to changes in markets and technologies during the development process, flexible product development has become popular in industries where the need for responsiveness is critical. In the flexible approach, the specification process and the development process overlap. As new requirements are added, some functions may become larger and their cohesion becomes lower. The program tends to be less and less structured. It is difficult to maintain good design quality throughout the development process.

Even though a software product might be well-designed during the software development process, over time the code will be modified in response to the changing needs of customers and technologies. Its original structure gradually fades and degrades. The original high-cohesive functions, which involved a single activity, tend to become

low-cohesive with multiple activities, and the size of those functions becomes larger. As a result, the program becomes difficult to understand and to change, and therefore it is more costly to maintain.

Program restructuring is an important option in software evolution in order to improve structures that have deteriorated and to keep software maintenance costs under control. It is also used in software development to turn a poorly designed program into well-designed one [Arn89]. Restructuring at the development process can help produce a high-quality product for customers and reduce maintenance costs during evolution.

In industry, programs are usually large and complex, and they are difficult to restructure manually based on the software engineer's intuition. In addition, "the cost of manual restructuring is, in practice, proportional to the complexity of the software" [Gri91]. Automated support for large and complex program restructuring is needed. A simple but effective tool is necessary to help software engineers identify the degraded structure and give restructuring advice, during both the development process and the evolution process.

1.2 Motivation

Since program structure tends to degrade during both software development and software evolution processes, restructuring poorly designed programs is needed to improve software quality and to keep or increase the software's value. The early days of restructuring focused on making a program's control flow easier to follow, and this category is quite mature [Arn89]. With regard to functional structure, however, the challenge of restructuring is how to group meaningfully related code together from a large code fragment to form small and cohesive functions, because in real-world code it

is common that unrelated code interleaves with related code fragments.

The previous research on restructuring low-cohesive programs at the function level has primarily used program slicing [Wei84] and input / output dependence techniques to restructure modules with cohesion as the main criterion [Kim94, Kan98, Kan99, Lak98, Lak99]. Conceptually, their works are very similar. The methods presented in both [Kim94] and [Kan98, Kan99] use data and control dependence information between input and output variables to measure cohesion and make restructuring decisions, which does not reflect the code fragments that are not directly related to the output variables. The tuck transformation provided by Lakhota and Deprez [Lak98, Lak99] complements [Kim94] and [Kan98, Kan99] by computing pairwise cohesion, which considers data and control dependence between all variables, but its process is complicated and difficult to use in industry. In addition, methods in [Kan98, Kan99, Lak98, Lak99] extract computations related to output variables. A function, which has a single output variable, cannot be decomposed further.

In practice, especially with regard to telecommunication programs, it is common that there are code fragments, such as the code related to error processing, which may be not related to output variables. In such cases, the slices of output variables cannot reflect the code fragment related to the error processing. In addition, it is also common that in a large function there is only one output variable (a global variable), but the function involves multiple functional activities. Therefore the previous approaches to restructuring low-cohesive functions have some limitations.

Cohesion, as an important measure in restructuring, measures how tightly related the elements are in a component. The goal of clustering is to group similar or related

elements together. It is possible to use clustering analysis to measure the strength of the relationship between elements in the component. Tzerpos and Holt [Tze98] provide a survey of approaches to the clustering problem in software engineering, and demonstrate that there is research potential in the software-clustering field. Wiggerts [Wig97] provides a general overview of clustering techniques and clustering approaches used in software system remodularization, and concludes, “Clustering methods seem a very good starting point for the remodularization of software.” The previous research in the software-clustering field has mainly been concerned with software remodularization at the architecture level.

In order to overcome the problems that have been demonstrated in program restructuring by the abovementioned previous research, this thesis presents an approach to program restructuring via clustering techniques at the functional level. It focuses on using automated support for identifying low-cohesive functions and making restructuring decisions, instead of the automated restructuring process. The purpose is to help software engineers identify ill-structured functions and give them heuristic advice for restructuring programs.

1.3 Goals and contributions of the thesis

This thesis proposes an approach to program restructuring using clustering techniques. The primary goal of the thesis is to develop a simple but effective approach to give automated support for identifying ill-structured low-cohesive functions and providing heuristic restructuring advice in both the development and evolution phases. Specifically the goals include:

- Define entities and attributes.

- Define a resemblance coefficient as a similarity measure that is used to group similar entities together.
- Conduct extensive experiments on the weights of different attributes.
- Compare different agglomerative hierarchical clustering algorithms: the single linkage algorithm (SLINK), complete linkage algorithm (CLINK) and unweighted pair-group method using arithmetic averages (UPGMA).
- Apply an industrial program as a case study to demonstrate and validate the proposed approach.

The main contributions of this thesis are as follows:

- A simple but effective approach to program restructuring using clustering techniques is proposed.
- A new similarity measure for program restructuring at the function level is defined and studied in depth.
- A comparative study on three clustering algorithms – SLINK, CLINK and UPGMA – is performed on a real telecommunication program.
- The empirical observations show that the heuristic advice provided by the approach can help software engineers make decisions as to why and how to restructure a program.
- The tools developed in this thesis are adopted by industry, Cistel Technology, for further evaluation and development.

1.4 Thesis organization

The structure of the rest of this thesis is as follows. Chapter 2 provides background information on program restructuring and software clustering. It also describes the related

work in both areas and the scope of this thesis. Chapter 3 proposes an approach for program restructuring using clustering techniques and discusses all the issues involved in the approach. It defines entities, attributes and a new resemblance coefficient as a similarity measure. Chapter 4 provides an extensive study on the similarity measure by weighting attributes differently. Chapter 5 contains the case study. The proposed approach is applied to restructure a real telecommunication program with different clustering algorithms. Empirical observations are also summarized. Chapter 6 presents the conclusions and future work.

Chapter 2 Background

This chapter gives background information related to this thesis and then defines the scope of the thesis. First, the chapter presents an overview of the software lifecycle and the software design quality and maintenance. Next, an overview of clustering techniques is provided. After that, previous related work is reviewed. Finally the chapter outlines the scope of the thesis.

2.1 Overview of software lifecycle

A lifecycle model presented by [Chi90] (Figure 2-1) clearly shows the key stages and concepts involved in the software lifecycle, and the relationship between them. In practice, iterations within the stages and overlaps between the stages are common.

Figure 2-1 Relationship between terms [Chi90]

When a system is designed, the first thing that needs to be understood is the problem to be solved, what rules should be followed, and what the limitations are. Those activities

belong to the first stage in the lifecycle model – the requirements stage. At the end of this stage, the specifications of the problem to be solved will be formed, including objectives, constraints and business rules. Next is the design stage, where possible solutions are studied and compared from a technical, economical and risk point of view, in order to decide the final solution. After that, the design is followed to implement the product, which includes coding, unit testing, integration and system testing, and delivery.

After delivery, the system may be modified to reflect changes in the customer's needs or the system's environment. Bugs that were not found during system validation may appear and need to be fixed. The process of changing a system after delivery is referred to as software maintenance, and it belongs to the software evolution process [Som96]. Although software engineering practice often emphasizes the design and implementation stages, it is widely acknowledged that the cost of maintaining a software product is between 60% and 80% of the total cost [Lun04].

The lifecycle phases consist of four activities: forward engineering, reverse engineering, reengineering and restructuring, which are described in the following sections.

2.1.1 Forward engineering

Forward engineering represents the traditional process in the software lifecycle, which moves “from high-level representations and logical implementation independent designs to the physical implementation of a system” [Chi90]. Figure 2-1 shows that this process passes through the requirements, design and implementation stages of the lifecycle. It includes software specification, development, and validation activities. It is mainly used to distinguish the forward engineering process from the reverse engineering

process when necessary.

2.1.2 Reverse engineering

Reverse engineering refers to “the process of analyzing software with the objective of recovering its design and specification” [Som96]. Figure 2-1 shows that this process covers the same three lifecycle stages as forward engineering, but in reverse order. The major goal of reverse engineering is to gain an understanding of the existing system in order to support both maintenance and new development. It forms part of the maintenance phase. The challenge of reverse engineering is to understand program code that has unsuitable documentation or is without documentation.

2.1.3 Reengineering

Reengineering is the process of taking an existing system and re-implementing it to produce a new and more maintainable system [Som96]. It is also associated with business process reengineering, such as improving the system performance. Usually, additional functionality will be added during this process [Chi90]. As shown in Figure 2-1, the reengineering process includes forward engineering and reverse engineering. It starts with the existing system, which is the main aspect that differentiates it from new system development. Reverse engineering, as part of the reengineering process, plays a role in developing a better understanding of the existing system.

2.1.4 Restructuring

Software restructuring is the process of modifying the internal structure of the software to make it easier to understand and to change when future requirements are added [Arn89]. It changes the internal structure of the software, but does not change its

external behavior. The goal of restructuring is to increase the understandability, maintainability, reusability and extensibility of the software. It is most often applied in the software maintenance phase, but it is also applicable to the software development phase. Figure 2-1 shows that restructuring can occur at any stage of the software lifecycle. In this thesis, restructuring is concerned with code-to-code restructuring, such as modifying code from poorly structured programs to well-structured programs (the implementation stage in Figure 2-1).

2.2 Software design quality and maintainability

In this thesis, the main purpose of program restructuring is to improve the quality of the program and make it more maintainable. Design quality directly affects maintainability, and good design can facilitate the maintenance phase. Design quality metrics are used to assess whether a design is good or not. Component-level design metrics, which are used in this thesis, focus on the internal characteristics of software components and include the following measures [Pre97]:

- Cohesion: this measures how closely the parts inside a component are related;
- Coupling: this measures how independent a component is;
- Complexity: this measures how reliable and maintainable a component is.

2.2.1 Component-level design metrics

2.2.1.1 Cohesion and coupling

The cohesion of a component is a measure of the tightness of the relationships among the elements inside the same component. A component should implement a single functionality, which represents a single part of the problem solution. All the elements that

contribute to this implementation should be encapsulated in this component [Som96 and Bri96]. If all the elements in a component are related to the implementation of a single functionality, the component is high-cohesive. Therefore the component is easy to understand, changes can be local in effect, and the system tends to be more reliable and maintainable. If the component contains unrelated operations or the component implements two or more functionalities, it has a low cohesion.

While cohesion measures the intra-component strength, coupling measures the inter-component strength. It captures the amount of relationships between the elements in the different components of a system [Bri96]. It reflects the degree of relationships between components [Mun03]. A system should have low coupling, so changes in a component will not affect many other components. The system is also less error-prone and easy to maintain. Low coupling also makes the system easy to understand.

Therefore, cohesion and coupling are closely associated with the understandability and maintainability of the system. A good design should make the components of a system with high cohesion and low coupling. However, there is a compromise between cohesion and coupling. If components are designed to be very small in order to gain higher cohesion, the relationship among the components will increase and the coupling will also be high.

2.2.1.2 Complexity

Complexity metrics are used to determine the complexity of program control flow. The cyclomatic complexity is the most widely used complexity metric, providing a quantitative measure of the logical complexity of a program [Pre97]. The value of cyclomatic complexity gives the number of independent paths in the program. The higher

the value of cyclomatic complexity, the more complex a program is, and therefore the program is less reliable and maintainable. Complexity metrics also provide feedback information during development activity to help control design quality.

2.2.2 Maintainability

During the software evolution process, the major activity is maintenance. There are three types of software maintenance [Som96]:

- Corrective maintenance: this means fixing reported errors, including coding errors, design errors and requirement errors;
- Adaptive maintenance: this refers to changing software to a new environment;
- Perfective maintenance: this is concerned with implementing new functional or non-functional system requirements.

Maintainability is a metric used to predict the relative amount of effort needed to maintain a software system. Considering these different types of maintenance activities, Pressman [Pre97] gives a definition of software maintainability as “the ease with which software can be understood, corrected, adapted and/or enhanced.” It is clear that software maintainability is directly related to software understandability. It is difficult to make any change to software whose functionality is not understood.

2.3 Overview of clustering techniques

Clustering techniques are used in a wide range of fields, from science, planning and management to engineering. People use clustering techniques to discover knowledge of nature or historical circumstances. Clustering can also be used to obtain new knowledge.

Clustering refers to activities that separate a set of data into groups or clusters. A

cluster is a group of entities. The entities within a cluster are more similar to each other than the entities from different clusters [Eve74]. There are many methods for measuring the similarity between entities based on selected attributes. There are also a lot of clustering algorithms. Different methods or different clustering algorithms may result in different clusters. Therefore, when performing a clustering analysis, three questions have to be answered:

- What are the entities to be clustered and what are the attributes used to describe them?
- How can the similarities between entities be measured?
- What algorithm is used to execute the clustering?

2.3.1 Entities and attributes

An entity is “an object or event in the real world” and an attribute is “a feature or property of an entity” [Fen97]. In this thesis, the term “entity” is used to refer to an item to be clustered, and the term “attribute” is used to represent an entity’s characteristic or feature, which is important in distinguishing one entity from another.

In practice, attributes can be measured using two categories: quantitative scale and qualitative scale. Quantitative scale includes interval scale and ratio scale. Both interval scale and ratio scale are continuous; the difference is in the magnitude of the data. Take a tank of water as an example. The weight of the water in kilograms can be measured in interval scale, such as $\text{weight} = (0, 100)$, and the level of the water can be measured in ratio scale, such as $\text{level} = (0, 1)$. Qualitative scale includes nominal scale and ordinal scale. A nominal scale measures unordered classes and multi-state classes; for example, the classes of direction (1 = east, 2 = south, 3 = west, 4 = north). An ordinal scale

measures ordered classes; for example, medals at the Olympic Games (1 = gold, 2 = silver, 3 = bronze).

Attributes that can be measured through the quantitative scale are called quantitative attributes. Similarly, qualitative attributes are attributes that can be measured by qualitative scale.

2.3.2 Similarity measures

The similarity between two entities is usually estimated by a resemblance coefficient or similarity coefficient. In this thesis the term, resemblance coefficient, is used. A resemblance coefficient is computed based on the scale of attributes and its value is between 0 and 1. Two entities with a higher resemblance coefficient are more similar than two entities with a lower resemblance coefficient. There are four categories of similarity measures [Sne73]: distance measures, association coefficients, correlation coefficients and probabilistic similarity measures, which are described briefly in the following sections.

2.3.2.1 Distance measures

Entities with p attributes are viewed as points in p -dimensional space, each dimension corresponding to one attribute. More disparity between two entities contributes to greater distance between them; that is, the less the distance, the more similar the two entities. There are different distance measures; the most commonly used is Euclidean distance.

Euclidean distance between entity j and entity k :
$$\left[\sum_{i=1}^p (X_{ij} - X_{ik})^2 \right]^{1/2}$$

where: X_{ij} is the value of i^{th} attribute of entity j

X_{ik} is the value of i^{th} attribute of entity k

2.3.2.2 Association coefficients

Association coefficients deal with qualitative attributes whose value is either 1 (representing the presence of the attribute) or 0 (representing the absence of the attribute). When attribute states are compared between two entities (entity j and entity k), the outcome can be summarized as follows [Sne73]:

- number of attributes that are present for both entities (1-1 match), represented by a ;
- number of attributes that are present for entity j but absent for entity k (1-0 match or mismatch), represented by b ;
- number of attributes that are absent for entity j but present for entity k (0-1 match or mismatch), represented by c ;
- number of attributes that are absent for both entities (0-0 match), represented by d .

Various association coefficients are presented in the literature to handle different matches. For instance, 1-1 matches are given more weight, 0-0 matches are ignored, and so on. The most frequently used association coefficients [Sne73] are:

- Jaccard coefficient: $\frac{a}{a+b+c}$

which gives the 1-1 matches and mismatches the same weight, and ignores 0-0 matches.

- Sorenson coefficient: $\frac{2a}{2a+b+c}$

which gives the 1-1 matches twice the weight of mismatches and ignores 0-0 matches.

- Simple Matching coefficient: $\frac{a+d}{a+b+c+d}$

which gives all matches the same weight and treats both 1-1 matches and 0-0 matches as having the same contribution to similarity.

2.3.2.3 Correlation coefficients

Correlation coefficients are used to measure proportionality and independence between pairs of entities. The value of the correlation coefficient ranges from -1 to 1 . A 1 value means maximum similarity; a -1 value means maximum dissimilarity; and a 0 value indicates that two entities are not related. The popular correlation coefficient is Pearson product-moment correlation coefficient. The correlation coefficient between entity j and entity k is [Sne73]:

$$\frac{\sum_{i=1}^p (X_{ij} - \bar{X}_j)(X_{ik} - \bar{X}_k)}{\sqrt{\sum_{i=1}^p (X_{ij} - \bar{X}_j)^2 \sum_{i=1}^p (X_{ik} - \bar{X}_k)^2}}$$

where: X_{ij} is the value of i^{th} attribute of entity j

X_{ik} is the value of i^{th} attribute of entity k

$\bar{X}_j$ is the mean value of all attribute values of entity j

$\bar{X}_k$ is the mean value of all attribute values of entity k

p is the number of attributes

2.3.2.4 Probabilistic similarity measures

Probabilistic similarity coefficients consider the distribution of the frequencies of attribute states over the set of entities whose attributes are qualitative attributes. The idea is that rare attribute states should be given more weight, but it seems that this idea is not generally accepted [Sne73].

2.3.3 Clustering algorithms

After entities and their attributes and the similarity measure have been selected, the next step is to choose a clustering algorithm. Clustering algorithms are procedures that are used for grouping a set of data into clusters based on similarity measures. Over the years, many clustering algorithms have been presented in different areas. Basically, however, clustering algorithms can be divided into two categories: hierarchical and non-hierarchical [Tse01].

2.3.3.1 Hierarchical algorithms

Hierarchical algorithms provide efficient steps for building a hierarchy of clusters without the knowledge of the number of clusters. The results can be presented using a dendrogram, which is a two-dimensional diagram that illustrates the clusters that have been made at each successive level (see section 3.5.1). Hierarchical algorithms are subdivided into agglomerative algorithms and divisive algorithms.

Agglomerative algorithms start at the bottom of the hierarchy by viewing each individual entity as a cluster, then group them into larger clusters at each successive step and finally form a single big cluster that contains all the entities. Divisive algorithms work in the reverse order to agglomerative algorithms. They begin with the top of the

hierarchy, where there is only one cluster containing all the entities. The cluster is then split into two clusters. Clusters with two or more entities will be split at each successive step, until all the clusters contain only one entity. For a set of data with n entities, there are $2^{(n-1)}-1$ possible divisions at the first step in division algorithms, where the computational complexity is high and it is not suitable for large sets of data. Therefore division algorithms are not commonly used. On the other hand, agglomerative algorithms are easier and more efficient than divisive algorithms and can be used to process large sets of data, so they are widely used in the real world.

In agglomerative algorithms, there are many available methods. The most commonly used are SLINK, CLINK and UPGMA.

- SLINK: also called the nearest neighbor method. It defines the similarity measure between two clusters as the maximum resemblance coefficient among all pair entities in the two clusters.
- CLINK: also called the furthest neighbor method. It defines the similarity measure between two clusters as the minimum resemblance coefficient among all pair entities in the two clusters.
- UPGMA: this defines the similarity measure between two clusters as the arithmetic average of resemblance coefficients among all pair entities in the two clusters.

SLINK tends to build a small number of loose clusters and CLINK tends to form a large number of compact clusters. UPGMA provides a compromised method between those two. In general, agglomerative algorithms are the most frequently used strategies in finding clusters, and it is feasible to use them with a larger data set. The main

disadvantage of hierarchical algorithms is that they cannot reallocate entities that may have been poorly classified at an early stage in the clustering process. The major difficulty with these techniques lies in the choice of a method and a similarity measure because there are many available methods in this category and many similarity measures could be used.

2.3.3.2 Non-hierarchical algorithms

Non-hierarchical algorithms, also called partitioning techniques or optimization algorithms, are different from hierarchical algorithms in that they reallocate entities to improve previous poor partitions at a later stage. They partition the set of entities by optimizing a predefined criterion. There are two distinct procedures:

- Initiate clusters. Select initial cluster centers and allocate entities to those clusters based on a similarity measure.
- Reallocate some or all of the entities to other clusters to improve a predefined criterion. This procedure is repeated until no single entity needs to be reallocated for further improvement of the criterion.

Most methods of initiating clusters begin with finding k points in the p -dimensional space, which are used as initial estimated cluster centers. Different initial clusters may produce different results. Mostly the user has to decide the value of k ; prior knowledge plays an important role, especially in the case of a large number of clusters. Unfortunately, a clustering problem usually refers to unsupervised learning without any *priori* knowledge. Users often need to try different values of k and repeat the whole process. This is very tedious and time-consuming. There are other methods for deciding initial clusters. One possibility is a random partition, which is used to generate initial

clusters. Then entities are reallocated until no further improvement can be made and the final clusters are formed. The initial clusters may also be generated from other clustering algorithms, such as hierarchical algorithms.

Another problem in this category is that the result is often a sub-optimal solution. In order to gain a global optimal solution, every possible partition of the data should be taken into consideration. Everitt [Eve74] gives an example, partitioning a set of 19 entities into 8 clusters. There are 1,709,751,003,480 distinct partitions. The number of partitions is huge, and it is impossible to consider every one in practice. Hence, the algorithms are designed to search for a local optimum of the criterion. In each iteration, entities are reallocated, the new partitions that give an improvement of the criteria are kept, and the old partitions are discarded. These algorithms are also called hill-climbing algorithms.

The k -means algorithm is a popular algorithm in this category. It treats entities with p attributes as points in p -dimensional space. It selects k points to be the centers of k clusters, finds the distance from each point to each center and assign the point to the closest center. Then it finds a new center for each cluster and repeats the process until the centers do not change.

With well-structured data, different initial clusters usually lead to the same final result. However, we do not know if the result is global optimal or local optimal (usually it is local optimal). The main problem with non-hierarchical algorithms is that they consume a large amount of computing time and are not suitable for use with very large data sets.

2.4 Related work

Previous research work related to this thesis includes program restructuring and software clustering analysis, which are described in the following two sections.

2.4.1 Related work on program restructuring

This thesis is mainly concerned with program restructuring at the function level, hence only related work in this area is discussed. The early days of restructuring efforts concentrated on making a program's control flow easier to follow, and this category is now quite mature [Arn89]. Restructuring that focuses on decomposing low-cohesive functions into high-cohesive functions still has some unsolved problems. The challenge is how to group meaningfully related code together to form high-cohesive functions, because in real-world code it is common that unrelated code segments interleave.

The closely related work on restructuring low-cohesive programs at the function level makes use of program slicing and input/output dependence techniques with cohesion as a criterion [Kim94, Kan98, Kan99, Lak98, Lak99]. Their ideas are similar, but they use different cohesion measures and different restructuring processes.

Kim and Kwon [Kim94] present a method of restructuring an ill-structured module at the software maintenance phase. The method applies program slicing to extract tightly coupled sub-modules (processing blocks), and uses module strength as a criterion to identify multi-function modules and to decide how to restructure such modules. Based on code implementation, module strength is defined in terms of level of sharing between processing blocks, which includes data sharing due to data interaction and control sharing due to control interaction. However, the method does not give the information that is not related to output variables.

Kang and Beiman [Kan98] have introduced a method to restructure poorly designed modules during the design and maintenance phases. The authors define the input/output dependence graph (IODG) of a module, similar to the variable dependence graph (VDG) in [Lak93], to model data dependence and control dependence relationship between input and output components of a module. They also define an association-based design-level cohesion (DLC) measure as a criterion for comparing alternative design structures. The cohesion measures presented in this paper only consider dependence information between input and output components, and do not reflect code fragments that are not related to the output components.

Kang and Beiman [Kan99] provide an extension to [Kan98]. The authors propose a software restructuring framework that uses two design-level models: IODG and a module interconnection graph (MIG). MIG is used to model the relationship between the modules of a system. They also define another design-level measure, design-level coupling (DCP), by examining coupling shown in MIG and IODG. This may be a good idea for high-level software restructuring, but it does not solve the problem existing in [Kan98], as discussed above, and may not work well for low-level program restructuring.

Lakhotia and Deprez [Lak98] use tuck transformation to restructure programs by breaking large functions into small functions. Tuck includes three transformations: wedge, split and fold. A wedge is a sub-set of statements in a slice, which contains related computations. After a wedge is formed, it is split from the rest of the code and folded into a new function. The paper uses a rule-based approach proposed in [Lak93] to compute pairwise cohesion between variables in the function as a criterion of restructuring. The rule-based approach defines cohesion levels that are related to output

variables in terms of data and control dependence. This paper provides the theoretical basis of program restructuring using tuck transformation.

Lakhotia and Deprez [Lak99] extend the work of [Lak98]. Their paper proposes a method for automatically identifying seed statements, which are inputs for program restructuring by tuck transformation. They define any variable modified by the function as an output variable. To restructure a program, first output variables are identified and pairwise cohesions between them are computed. The pairwise cohesions are then used to identify low-cohesive functions and select seed statements. Finally the function is restructured using tuck transformation. The empirical study in this paper shows that the cohesion rules do not work well for data structure, such as struct, because it treats each independent member in the data structure as a single variable. The empirical study also shows that it takes a whole day and uses up 40 MB of swap space when running a program with 30,000 lines. Although the paper argues that this is because the tool is poorly designed, it also indicates that the proposed method is quite complicated.

The pairwise cohesion mentioned by [Lak98, Lak99] complements the cohesion measures provided by [Kim94] and [Kan98, Kan99] because it considers dependence information between all variables in the function, not just input and output variables. However, the proposed method in [Lak98, Lak99] seems quite complicated and not suitable for industrial application. In addition, the methods in both [Kan98, Kan99] and [Lak98, Lak99] treat a function that has only one output variable with the highest cohesion, so there is no need to restructure such a function. However, in practice, a function with one output variable (such as a global variable) may involve multiple functional activities.

2.4.2 Related work on software clustering

Research on software clustering has been done at the design or architectural level. Tzerpos and Holt [Tze98] present a survey on the research potential of clustering approaches to software engineering, where they indicate that classic clustering techniques can be used in the software context and there is research potential in the software-clustering field. Wiggerts [Wig97] provides a general overview of clustering techniques and their applications to system remodularization, where the benefit from the general theory of clustering analysis is highlighted. Lakhotia [Lak97] gives a survey on subsystem classification techniques and provides a unified framework for entity description and clustering methods in order to facilitate comparison between various subsystem classification techniques.

Hutchens and Basili [Hut85] use data bindings as features or attributes to measure the similarity between entities, and hierarchical clustering algorithms to perform clustering analysis for system modularization. They also compare different clustering algorithms, but the study does not provide sufficient evidence to conclude which algorithm is the best.

Schwanke and Platoff [Sch89, Sch91] provide a tool called Arch, which is used to group related procedures into modules. The proposed approach in [Sch89] defines the similarity measure based on shared neighbors. Schwanke [Sch91] uses information sharing as the similarity measure, which is based on the non-local names used by procedures. It also uses maverick analysis to identify individual procedures that appear in the wrong module. Clustering is done by a SLINK algorithm in the hierarchical clustering technique.

Choi and Scacchi [Cho90] use the module interconnection language (NuMIL) to transform a resource flow diagram (RFD) into a resource structure diagram (RSD), which shows the hierarchical structure of a system. A module could be a single function or a collection of functions, such as a file. The restructuring measure is the minimum-coupling and minimum-alteration-distance.

Müller et al. [Mül93] use interconnection strength and client/supplier measures to construct hierarchical subsystem structures. The resulting compositions are represented by $(k, 2)$ -partite graphs, which is a combination of resource flow graphs and composition dependency graphs. The construction process is manipulated by the Rigi tool.

Lung [Lun98] proposes a method that uses the inter-connectivity of components as features. The author adopts the Sorenson similarity measure and uses the UPGMA clustering algorithm to recover software architecture. The author also combines use cases and clustering techniques together to reduce system complexity. Lung et al. [Lun04] introduce applications of clustering techniques into the software lifecycle, including software partition in the design phase, architecture recovery in the reverse engineering process, software restructuring at the maintenance stage, and source code improvement. The main difference between [Lun98, Lun04] and others is that the individual scenario is applied to the clustering analysis at the architecture level.

Mancoridis et al. [Man98] introduce a clustering tool called Bunch, which treats automatic modularization as an optimization problem. The objective is maximum modularization quality, which is defined by subtracting the average inter-connectivity from the average intra-connectivity to demonstrate the tradeoff between intra-module cohesion and inter-module coupling. A module dependency graph (MDG) is used to

capture the intra-connectivity and inter-connectivity. The tool uses hill-climbing algorithms to produce sub-optimal results. Mancoridis et al. [Man99] have extended Bunch and integrated the designer knowledge about the system structure into the automatic clustering process in order to obtain better results. Mitchell and Mancoridis [Mit01] propose a new similarity measure by considering edge weight in MDGs and have integrated it into the Bunch tool.

Anquetil and Lethbridge [Anq03] present an analysis of entity description, similarity metrics comparison and a comparison of three different hierarchical clustering algorithms (SLINK, CLINK and UPGMA). They choose files as entities to conduct experiments on three public domain systems: gcc, Linux and Mosaic. The method uses formal features (having direct influence on the system behavior) and non-formal features (having no direct influence on the system behavior) to describe entities, and concludes that the non-formal feature kinds provide more information than the formal feature kinds. The comparison among different taxonomic similarity metrics shows that the Jaccard and Sorensen coefficients give the best results. The comparison among different clustering algorithms shows that the CLINK algorithm gives the best results, especially regarding cohesion. The UPGMA algorithm is a good choice when more coupling is concerned, and the SLINK algorithm tends to form a black hole configuration.

Previous software clustering approaches have concentrated on software system modularization or remodularization at the architectural or design level. They follow software engineering principles, high intra-module cohesion or low inter-module coupling, or both. The entities are functions or files. Their similarity measures are either based on relationships between entities [Hut85, Cho90, Mül93, Lun98, Lun04, Man98,

Mit01], or based on shared features [Sch89, Sch91, Anq99, Anq03], with or without giving weight to the relationships or features. Researchers have used different information to measure the similarity based on different points of view.

The clustering algorithms used in previous work fall into three categories: hierarchical algorithms [Hut85, Sch89, Sch91, Lun98, Lun04, Anq99, Anq03], optimization algorithms [Man 98, Man99, Mit01], and graph theoretic algorithms [Cho90, Mül93]. The hierarchical algorithms are the most used. In hierarchical algorithms, the survey in [Lak97] shows that most researchers prefer the SLINK algorithm, but the CLINK algorithm is suggested by [Anq03], based on their experiments. It seems that different algorithms are suitable for different applications.

Clustering analysis in software engineering is a sophisticated research domain. It is the researchers' job to decide how to choose entities and the entity's attributes, how to compute the measure of similarity, and which algorithm to use for a particular clustering problem. More research and experimental study, especially in large systems, are needed.

2.5 Scope of the thesis

Software restructuring is a very important activity in the software lifecycle. Many efforts have been made to support restructuring automatically. Previous restructuring work at the function level has focused on program slicing and input/output dependence techniques, which have shown some limitations. The dependence information between input and output variables in [Kim94, Kan98, Kan99] does not reflect the complete information on the function. The pairwise cohesion mentioned by [Lak98, Lak99] seems quite complicated and not suitable for industrial applications. On the other hand, a lot of work has been reported on restructuring at the architectural level using clustering

techniques, and has shown that software clustering is a potential field of research. However, not much has been applied to program restructuring at the function level.

This thesis concentrates on program restructuring in enhancing structure at the function level, using clustering techniques. It emphasizes automated support for identifying ill-structured code fragments that are candidates for restructuring and facilitates restructuring decision. As discussed in section 2.3, clustering analysis involves three issues: entities and entity description, the similarity measure and the clustering algorithm. This thesis covers all three of these issues. In detail, the thesis includes:

- An approach to program restructuring: This thesis proposes an approach as a general guideline for program restructuring using clustering techniques. Some new functionality has been added to the existing tools [Lun04] to provide automated support for program restructuring with the proposed approach.
- Entity and entity description: An entity can be described by using different attributes from different points of view. This thesis discusses the selection of entities and how to select attributes that are important to distinguish two different entities, from a cohesion point of view.
- Similarity measure: Different kinds of attributes may have different contributions to the similarity measure. This thesis conducts extensive experiments on weight adjustments between different attributes and defines a new resemblance coefficient as a similarity measure for program restructuring at the function level.
- Clustering algorithm: Non-hierarchical algorithms are complex and often give sub-optimal results. They are not an option in this thesis. In the hierarchical category, the agglomerative approach is more effective than the divisive approach

and therefore has been chosen for this thesis. Among the many different agglomerative algorithms, SLINK, CLINK and UPGMA are primarily used, and different researchers have their own preference. This thesis compares these three algorithms during the similarity measure experiments and the case study.

- Case study: This thesis applies the proposed approach to restructure an industrial program, analyzes empirical observations, and validates the proposed similarity measure.

Chapter 3 An Approach to Program Restructuring Using Clustering Techniques

This chapter presents an approach to program restructuring using clustering techniques at the function level. The restructuring objective is stated in section 3.1 and an overview of the approach is given in section 3.2. Based on the objective, section 3.3 identifies the entities and their attributes that are used in the clustering analysis. The objective also leads to the similarity measure, which is discussed in section 3.4. Section 3.5 describes how to explain the clustering analysis results, obtain heuristic advice from them and make restructuring decisions. After a restructuring decision has been made, the real restructuring activities are discussed in section 3.6.

3.1 Restructuring objective

The research in this thesis is concerned with restructuring ill-structured programs at the function level by decomposing low-cohesive functions into high-cohesive functions. The structural properties of a program are characterized by internal attributes, in which two commonly used are cohesion and coupling. Cohesion refers to the relative functional strength of components. A high-cohesive function means that statements in a function are highly related and the function performs a single task. Such a function is considered to be well-structured. An ill-structured function means that the statements in the function are less related, and more than one functional activity may be involved. Such a function has low cohesion. Coupling refers to the degree of connection of one component to other components. It is commonly accepted that good design should compose components with high cohesion and low coupling.

The objective of program restructuring is to improve the structure or internal strength of a function. Cohesion is used as the main criterion in this thesis. The criterion of coupling is only considered when software engineers make final restructuring decisions.

3.2 Program restructuring approach

The program restructuring approach proposed in this thesis is based on clustering analysis, with cohesion as the main criterion, and is supported by a set of tools. The existing structure of a program with a quantitative measure is shown in a clustering tree after a clustering analysis has been performed. The approach provides information about the existing structure of a function, the quantitative structure measure, and heuristic advice for improving the existing code. It can be used to help software engineers make a decision - why and how to restructure an existing program. Figure 3-1 shows the approach for program restructuring using clustering techniques. Currently the study is used for restructuring C programs; however, the technique can be applied to other languages as well.

Figure 3-1. An approach to program restructuring

The restructuring approach using clustering techniques has four key phases, as shown in Figure 3-1. Phase one is data collection and processing. In this phase, the Parser tool parses source code automatically and produces raw data of entity-attribute matrix. The raw data may contain some “noise” (unwanted data). For example, a loop counter increment statement is an unwanted entity, and a constant variable is an unwanted attribute. There is no guarantee for getting the exact data that the user wants through the Parser. The “noise” is removed during data refining. The entity-attribute matrix after data refining is the input data for the next phase – clustering.

Phase two is clustering. The most important and fundamental step in clustering analysis is the similarity measure. When entities and their attributes are defined, a metric called a resemblance coefficient is adopted to measure the similarity between entities. Because different attributes may have different contributions to the similarity between two entities, the resemblance coefficient has to be concerned with the weighting of the attributes. This is a complicated issue and the experiments are conducted to decide the attributes’ weight. After the resemblance coefficient has been clearly defined, the similarity between two entities can be measured and clusters can be constructed using a clustering algorithm. Currently, the approach supports three agglomerative hierarchical algorithms: SLINK, CLINK and UPGMA. The Clustering tool performs this phase automatically.

Phase three is visualization and analysis. After the clustering phase, the clustering result is displayed as a clustering tree (dendrogram), which shows the existing structure of a function being analyzed. Closely related entities are grouped into a cluster. The degree of relatedness of entities in a cluster is represented by the resemblance coefficient,

which is also shown in the tree. By examining the tree, ill-structured code fragments may be identified; these are the candidates for restructuring. The clusters in the tree also provide heuristic advice on how to restructure the function. However, the number of clusters depends on the tree cut-point. In addition to the program size and complexity, other factors, such as performance, are also involved in the tree cut-point. Therefore, software engineers must participate in making the final decision for restructuring based on the clustering result and their experience and insight.

Phase four is restructuring, which is the real action produced by the decision. Identified low-cohesive functions are decomposed into several code fragments, and some of them are composed into new functions. This phase is processed manually.

3.3 Data collection and processing

In the restructuring approach shown in Figure 3-1, the entity-attribute matrix is the input data for the clustering analysis. An entity can be described using different attributes from different points of view. In clustering analysis, only the attributes, which are important to describe the entity's properties that are related to the objective, are considered. Clustering analysis is meaningful only when it is based on a valid entity-attribute input matrix. Therefore choosing relevant entities and their attributes is a key issue in clustering analysis, which is described below.

3.3.1 Entities

The cohesion of a function refers to the degree of relatedness of statements in the function. For program restructuring at the function level, statements are chosen as entities because they are directly related to the restructuring objective. There are two types of

statements: executable statements and non-executable statements. Non-executable statements, such as comment statements and declaration statements, have no real effect on the implementation of the functionality provided by the function. So they are not selected as entities. Executable statements include assignment statements, predicate statements, iteration statements, function call statements, end statements, and so on. However, only the executable statements that can be described by their attributes are considered to be entities (attributes are discussed in the next section). A statement, such as a *break* statement or *continue* statement, which does not contain any attribute described in section 3.3.2, is not considered as an entity. In addition, an output statement, such as the *printf()* statement in C written for debugging, is not counted as an entity because it is not related to the functional activity.

Therefore, in the restructuring approach, an entity is an executable statement that is related to a functional activity and can be described by its attributes. Entities can be subdivided into control entities and non-control entities. A control entity refers to an entity that is either a predicate statement (such as an *if* or *switch* statement) or iteration statement (such as a *for* or *while* statement). If an entity is not a control entity, it is a non-control entity. Each entity is represented by the line number, which corresponds to a statement in the source code. In other words, each number refers to one entity, which usually represents one executable statement. However, an entity may also represent more than one executable statement in the case of a control entity, such as a control statement *for(i=0; i<10; i++)*.

Figure 3-2 is a sample code from [Lak99], which is an example of low-cohesive code with interleaved computations. This function uses the same input data *sale* to compute

```

1  procedure sale_pay_profit(days:integer;
                                cost: float;
                                var sale: int_array;
                                var pay: float;
                                var profit: float;
                                process: boolean);
2  var i: integer; total_sale, total_pay: float;
3  begin
4      i := 0;
5      while i < days do begin
6          i := i + 1;
7          readln(sale[i]);
8      end;
9      if process = True then begin
10         total_sale := 0;
11         total_pay := 0;
12         for i:=1 to days do begin
13             total_sale := total_sale + sale[i];
14             total_pay := total_pay + 0.1*sale[i];
15             if sale[i] > 1000 then
16                 total_pay := total_pay + 50;
17             end;
18             pay := total_pay/days + 100;
19             profit := 0.9*total_sale - cost;
20         end;
21 end;

```

Figure 3-2. Sample code 1: Sale, Pay and Profit [Lak99]

two outputs *pay* and *profit*. In the code, entities are statements in line 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 18, and 19, and represented by (5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19). Statements in line 4 and 6 are not entities because the variable *i* is a loop counter that is not an attribute (see section 3.3.2); that is, there is no attribute used to describe them. Entities (5, 12) and entities (9, 15) are control entities, which correspond to iteration statements and predicate statements, respectively. All the other entities are non-control entities; entity 7 is a function call statement and the others are assignment statements. Each entity represents one executable statement, except for entity 12, which includes three executable statements:

1. $i = 1;$
2. $i < days;$
3. $i++.$

3.3.2 Attributes

An attribute is a feature of an entity. An entity may have many attributes. Different properties of an entity can be described by different attributes. However, selected attributes must contribute to the understanding of a predefined objective criterion. In a function written in C, a statement consists of variables, constants, operators, key words, brackets, function names (in function call statements) and semicolon. In the context of cohesion, a statement is evaluated to see if it is related to a functional activity. Different variables and function names may be related to different functional activities and therefore are used as attributes. Constants and operators only affect the result of a functional activity, but they are not related to a specific activity. Take the following two statements as an example:

$pay = total_pay/days + 100;$

...

$pay = total_pay/days - 50;$

where the variables pay , $total_pay$ and $days$ are related to a functional activity – the computation of pay . The constants ('100' and '50') and operators ('+', '-', and '=') only affect the value of pay . As another example, two statements

$tax = income * 0.3;$

$food = expense * 0.3;$

are not related, although both have the same operators ('*' and '=') and the same constant

'0.3'. For the key words, such as *return*, see the following two statements:

```
return (income * 0.3);
```

```
return (expense * 0.3).
```

Although both share the key word *return*, they are distinct statements without any relationship. Hence constants, operators and key words are not used to distinguish two different statements from a cohesion point of view, and they are not chosen as attributes. Based on the data dependence and control dependence relationship, variables can be divided into data variables and control variables, which are described below:

Data variable. A data variable refers to the variable that is directly used in a statement. Data variables as a type of attribute in a function reveal the data dependence relationship between entities. They include local variables, global variables, and parameters passed to the function. They can be divided into two types of variables: variables with a primitive type and variables with a composite type, or a user-defined type. A composite variable, such as an array, a linked list, or a user-defined data structure (struct), is treated as one variable, because the program restructuring focuses on functional structure, not on data structure. In addition, a function name in a function call statement is also treated as a data variable.

Loop counter variable. A loop counter variable is a kind of data variable and is used to count the number of times that a loop is repeated. Because the restructuring focuses on static functional structure, no matter how many times a loop is repeated, the loop body is treated as having the same relatedness to one or more functional activities. In addition, the loop counter is usually used to indicate the element of a composite variable that is treated as one variable. Therefore, the loop counter variable is not counted as an attribute

of an entity. For example, in the sample code 1 in Figure 3-2, variable *i* in line 4 to line 7 is a loop counter variable. The variable *i* is used to indicate the *i*th element of an array variable *sale*. In this case, the variable *i* is not treated as an attribute.

Control variable. Data variables, as a type of attribute, describe the data dependence relationship between entities. However, they do not show control dependence relationships. In order to reveal control dependence, control variables are postulated as a type of attribute in the restructuring approach. A control variable is one that is artificially added to an entity in a control block. It is a logical variable used to describe control dependence relationship between entities. Entities with the same control variable mean that they belong to the same control block in the source code. Take the sample code 1 in Figure3-2 as an example: entities 13 and 14 are located in two control blocks: the *if* control block and the *for* loop control block. Two control variables – for example, *if1* and *loop2* – are added to describe these two entities. Thus the attributes used to describe entity 13 are *total_sale*, *sale*, *if1* and *loop2*, and the attributes for entity 14 are *total_pay*, *sale*, *if1* and *loop2*.

In the restructuring approach, data variables (not including loop counter variables) and control variables are chosen as attributes for describing entities from the cohesion point of view. They are also called data attributes and control attributes. Take the sample code 1 in Figure 3-2 as an example. The entities and attributes are shown in Table 3-1. From this table, we can see that entity 5 is described by two attributes – *days* and *loop1* – and entity 7 is described by three attributes – *readln*, *sale* and *loop1*. Both entities share the same control attribute *loop1*, which indicates that both are located in the same loop control block.

Table 3-1. Entities and their attributes of sample code 1 in Figure 3-2

Entity	Attribute												
	Data attribute									Control attribute			
	<i>days</i>	<i>readln</i>	<i>sale</i>	<i>process</i>	<i>total_</i> <i>sale</i>	<i>total_</i> <i>pay</i>	<i>pay</i>	<i>profit</i>	<i>cost</i>	<i>loop1</i>	<i>if1</i>	<i>loop2</i>	<i>if2</i>
5	√									√			
7		√	√							√			
9				√							√		
10					√						√		
11						√					√		
12	√										√	√	
13			√		√						√	√	
14			√			√					√	√	
15			√								√	√	√
16						√					√	√	√
18	√					√	√				√		
19					√			√	√		√		

3.3.3 Scale of attributes

As mentioned in section 2.3.1, an attribute can be measured with a quantitative scale or a qualitative scale. It is hard to measure the attributes defined in section 3.3.2 with a quantitative scale, so in this thesis each attribute is measured on a qualitative scale as a binary representation. Thus each attribute has two states, either presence or absence. The absence states for both data and control attributes are represented by the same value 0. However, the presence states for data and control attributes are represented by different values. They are described as follows:

0 – absence state of a control or data attribute

1 – presence state of a control attribute

2 – presence state of a data attribute

There is a special case for control entities. The data attributes in control entities are

treated as control attributes, which are discussed in section 3.4.1. Between any two entities, there are six different types of matches, as follows:

- 1-1 match: a control attribute is present in both entities; that is, both entities share a common control attribute.
- 2-2 match: a data attribute is present in both entities in case neither of them is a control entity. That is, two non-control entities share a common data attribute.
- 0-0 match: an attribute (a control attribute or data attribute) is absent in both entities.
- 1-0 or 0-1 match (mismatch): a control attribute is present in one entity but absent in the other.
- 2-0 or 0-2 match (mismatch): a data attribute is present in one entity but absent in the other.
- 2-1 or 1-2 match: a data attribute is present in both entities in case one of them is a control entity and the other is a non-control entity.

Now, entity-attribute matrix can be represented by these states. Table 3-2 gives the entity-attribute matrix of the sample code 1 in Figure 3-2. There are 12 entities and 13 attributes. Entities 5, 9, 12 and 15 are control entities. The variables in those statements are used for control; hence they are treated as control attributes with a state value of 1. For example, the data attribute, *days*, has a value of 1 in entities 5 and 12, but it is assigned to 2 for entity 18 because entity 18 is a non-control entity. For entities 5 and 7, there are one 1-1 match, nine 0-0 matches, one 1-0 match and two 0-2 matches between them. These two entities are in the same control block but do not share any data attribute. For entities 5 and 12, there is only one 1-1 match on data attribute *days*, but they are in

different control blocks because they do not share any control attribute. For entities 13 and 14, they share two control attributes, *if1* and *loop2*, and one data attribute *sale*, so there exist two 1-1 matches and one 2-2 match between them and they are in two common control blocks. There are a lot of 0-0 matches between any two entities, since usually only a few attributes are present in each entity.

Table 3-2. Entity-attribute matrix of sample code 1 in Figure 3-2

Entity	Attribute												
	Data attribute									Control attribute			
	<i>days</i>	<i>readln</i>	<i>sale</i>	<i>process</i>	<i>total_sale</i>	<i>total_pay</i>	<i>pay</i>	<i>profit</i>	<i>cost</i>	<i>loop1</i>	<i>if1</i>	<i>loop2</i>	<i>if2</i>
5	1	0	0	0	0	0	0	0	0	1	0	0	0
7	0	2	2	0	0	0	0	0	0	1	0	0	0
9	0	0	0	1	0	0	0	0	0	0	1	0	0
10	0	0	0	0	2	0	0	0	0	0	1	0	0
11	0	0	0	0	0	2	0	0	0	0	1	0	0
12	1	0	0	0	0	0	0	0	0	0	1	1	0
13	0	0	2	0	2	0	0	0	0	0	1	1	0
14	0	0	2	0	0	2	0	0	0	0	1	1	0
15	0	0	1	0	0	0	0	0	0	0	1	1	1
16	0	0	0	0	0	2	0	0	0	0	1	1	1
18	2	0	0	0	0	2	2	0	0	0	1	0	0
19	0	0	0	0	2	0	0	2	2	0	1	0	0

3.4 Similarity measure

As stated earlier, the restructuring objective criterion is cohesion. At the function level, cohesion refers to the degree of relatedness of statements in a function. The similarity measure is used to evaluate cohesion and is represented by a resemblance coefficient. Generally, the more attributes two entities share, the closer they are related and the more similar they are. For qualitative attributes, association coefficients are most

commonly used for a similarity measure. There are many different association coefficients, as discussed in section 2.3.2.2. Among them, the Jaccard coefficient and the Sorenson coefficient are popular and have been used for software clustering analysis in previous research [Lun04, Lun98, Anq03 and Anq99]. The Jaccard coefficient treats all the attributes equally, with the same weight. The Sorenson coefficient gives more weight to 1-1 matches, but basically it still treats all the attributes equally.

The approach outlined in this thesis distinguishes attributes into two types: data attributes and control attributes. From a cohesion point of view, these two types of attributes contribute to different degrees of cohesion, because they describe different dependence relationships between entities. Lakhoria [Lak93] defines cohesion levels based on data and control dependence relationship. The author indicates that two variables, which have a data dependence relationship, are more high-cohesive than two variables that have a control dependence relationship. Therefore weighting attributes is necessary to gain a solid similarity measure. Both Jaccard and Sorenson coefficients do not work well in the restructuring approach, because they give both data attributes and control attributes the same weight, as described in section 4.1. Before defining a new resemblance coefficient to measure similarity between entities, some weighting issues should be discussed first.

3.4.1 Issues with weighting of attributes

As discussed above, data attributes and control attributes should be weighted differently. However, data attributes can be different types. As mentioned in section 3.3.2, a data attribute may be a local variable, a global variable, a parameter passed to a function or a function name in a function call statement. Should they all be treated the

same way? In addition, a data attribute may appear in a non-control entity or a control entity. Should a data attribute be measured equally when it is used to describe different types of entities? If not, how should it be dealt with? These questions are discussed below.

Variable scope. A global variable can be referenced by multiple functions in a program. It may be related to many different functional activities. A local variable is referenced inside a function and it is only related to the functional activities provided by the particular function. At the function level, however, from the functional activity point of view, there is no difference between a global and a local variable. In a low-cohesive function, a global variable may be referenced by several different activities. But a local variable or a parameter passed to the function may also have the same situation. The parameter *sale* in the sample code 1 in Figure 3-2 is involved in two different activities (computation of pay and computation of profit). A global variable and a local variable in a function play the same important role in function cohesion and are therefore treated equally.

Function call. A function name in a function call statement is treated as a data variable. In the restructuring approach, a function call statement is treated as a non-control entity. Different functions usually perform different tasks or activities. A function name is used to distinguish different function calls that correspond to different functionalities. Therefore, a function name in a function call is measured in the same way as a local variable.

Data attributes in control entities. A control entity refers to a predicate statement or an iteration statement. A control entity is different from a non-control entity in that it has

an indirect contribution to a functional activity. In addition, one control entity may be shared by more than one activity. For example, entity 12 in the sample code 1 in Figure 3-2 is shared by two activities – computation of total sale and computation of total pay – but it does not directly participate in either activity. When a variable or a data attribute appears in a control entity, it has no direct relatedness to an activity. However, when it is used in a non-control entity, it is directly related to an activity. Therefore, data attributes in control statements or entities should be treated differently from those in non-control entities. In this approach, data attributes used in control entities are simply treated as control attributes. The experimental results presented in section 4.8 support the rationale.

Therefore, all the data attributes in the non-control entities are considered to have equal importance for functional cohesion. That is, all the data attributes in the non-control entities have the same weight. Data attributes in control entities are treated as control attributes. Thus the problem of weighting attributes boils down to the problem of the weighting between control attributes and data attributes. A data attribute has a direct effect on a functional activity and a control attribute only affects a functional activity indirectly. So data attributes should be weighted more than control attributes.

3.4.2 Issues with weighting of matches

Another issue is to determine the weight for different matches. Different matches play different roles in the similarity measure. They are discussed as follows.

0-0 match: 0-0 match means that an attribute is not used in either of the two entities. In an entity-attribute matrix, there are usually many attributes that are used in a function. But for each individual entity, it is only related to a few of them, and most attributes are valued with 0. There are a lot of 0-0 matches in the matrix. Lung et al. [Lun04] argue that

counting 0-0 matches will generate distortion and result in dissimilarity. The study presented by Anquetil and Lethbridge [Anq03] also shows that better results are obtained without considering 0-0 matches. In program restructuring, the similarity of two entities is not affected by adding unrelated attributes to the function. Therefore, 0-0 matches are ignored.

1-2 / 2-1 match: This kind of match occurs between one control entity and one non-control entity when they share a common data attribute. When these two entities are in the same control block, they share a common control variable and there is already a 1-1 match that counts the control dependence. Thus there is no need to use this match to describe control dependence again. When these two entities are not in the same control block, they do not have dependence relationship. Take the sample code 1 in Figure 3-2 as an example. Entity 12 is a control entity belonging to a *for* loop block, and entity 18 is a non-control entity outside the *for* loop block. Although both entities share a data variable *days*, they are not in the same *for* loop block (*loop2*) and do not have dependence relationship between them. So 1-2 / 2-1 matches are ignored.

In addition to 0-0 matches and 1-2 / 2-1 matches, there are four other types of matches, which are 1-1 matches, 2-2 matches, 1-0 / 0-1 matches and 2-0 / 0-2 matches. 1-1 matches and 2-2 matches mean that two entities share common attributes, which have a positive contribution to the similarity of entities. 1-0 / 0-1 matches and 2-0 / 0-2 matches are mismatches, which have a negative contribution to the similarity of entities. All of them have an important effect on the similarity measure.

1-1 match and 2-2 match. A 1-1 match indicates that two entities have a control dependence relationship, or two control entities share a common data variable. It reflects

the control structure of a function. A 2-2 match shows that two entities have a data dependence relationship. Because data dependence is more cohesive than control dependence from the cohesion perspective, a 2-2 match should have more weight than that of a 1-1 match.

1-0 / 0-1 match and 2-0 / 0-2 match. A 1-0 / 0-1 match is a mismatch on a control attribute and shows the dissimilarity on control dependence or control structure. A 2-0 / 0-2 match is a mismatch on a data attribute and describes the dissimilarity for data dependence. Both contribute to the dissimilarity between entities. If matches on common data attributes (2-2 matches) play a more important role in similarities between entities than matches on common control attributes (1-1 matches), then mismatches on data attributes (2-0 / 2-0 matches), on the other hand, should also have more importance for dissimilarity than mismatches on control attributes (1-0 / 0-1 matches). Hence, 2-0 / 0-2 matches should be weighted more than 1-0 / 0-1 matches.

In summary, 0-0 matches and 1-2 / 2-1 matches are ignored; 2-2 matches contribute more to the similarity than 1-1 matches; 2-0 / 0-2 matches contribute more to the dissimilarity than 1-0 / 0-1 matches. The matches on data attributes are more important than on control attributes. The weighting of matches is consistent with the weighting of attributes.

3.4.3 Resemblance coefficient

According to the discussion above, 0-0 matches and 1-2 / 2-1 matches are ignored; 1-1 matches and 2-2 matches contribute to the similarity of entities; 1-0 / 0-1 matches and 2-0 / 0-2 matches contribute to the dissimilarity of entities. Different weights should be given to different types of attributes, but the same type of attributes is given the same

weight no matter whether it contributes to the similarity or dissimilarity. That is, 1-1 matches and 1-0 / 0-1 matches share the same weight, and 2-2 matches and 2-0 / 0-2 matches share the same weight. A new resemblance coefficient between two entities is defined as follows:

$$coeff = \frac{w_d a_d + w_c a_c}{w_d a_d + w_c a_c + w_d b_d + w_c b_c} \quad (3-1)$$

where: *coeff* - resemblance coefficient

a_d - number of 2-2 matches between two entities

a_c - number of 1-1 matches between two entities

b_d - number of 2-0 and 0-2 matches between two entities

b_c - number of 1-0 and 0-1 matches between two entities

w_d - weight of data attributes

w_c - weight of control attributes

$w_d > w_c > 0$.

Here, the weight of an attribute represents its importance compared to other attributes. Attributes of the same type are weighted the same. Because data attributes play a more important role in function cohesion than control attributes, the weight of data attributes is heavier than control attributes. If there is no common attribute shared by two entities, they are unrelated and $coeff = 0$. If all attributes used to describe two entities are shared by the two entities, that is, $b_d = 0$ and $b_c = 0$, the two entities achieve their maximum similarity with $coeff = 1$. The value of resemblance coefficient is between 0 and 1. The similarity between entities also depends on the total number of attributes, the number of

data attributes and the number of control attributes used in the entities.

This resemblance coefficient is a modification of the Jaccard coefficient, taking the weighting of attributes into consideration. When $w_d = w_c$, it is equivalent to the Jaccard coefficient. The weights are decided through the experiments, which are discussed in Chapter 4. The final result is $w_d = 8$ and $w_c = 3$. Thus the resemblance coefficient can also be described by:

$$coeff = \frac{8a_d + 3a_c}{8a_d + 3a_c + 8b_d + 3b_c} \quad (3-2)$$

3.5 Visualization and analysis

After the clustering phase, the next phase in the restructuring approach is visualization and analysis, which consists of generating a clustering tree and its subsequent analysis. These are discussed below.

3.5.1 Clustering tree

The result of the clustering analysis is depicted by a dendrogram. A dendrogram is a two-dimensional diagram, in which there is a vertical scale of a resemblance coefficient from 1 to 0, and the entities are indicated in a horizontal direction. The numbers along the horizontal direction correspond to the line numbers in source code. The diagram illustrates the hierarchical structure of the functional relatedness of the entities. Entities that are more closely related are grouped in the lower layer with the higher resemblance coefficients.

Figure 3-3 shows a clustering tree for the sample code 1 in Figure 3-2, where the numbers on the horizontal direction represent entities. For example, number 5 refers to

entity 5, which is the statement at line 5 in the source code. The numbers are not contiguous and there is no fixed starting number, because they only represent the statements that are considered as entities. In the sample code 1, there are 21 lines of code and only 12 of them are selected as entities. Figure 3-3 also shows the hierarchical structure of the tree. There are four branches or clusters, C1, C2, C3 and C4. C2 and C3 are grouped to form cluster C5, then C5 and C4 are grouped to form cluster C6, and finally C1 and C6 form the biggest cluster, C7. The resemblance coefficient between two clusters shows the relatedness between them. For example, the resemblance coefficient between C1 and C6 is 0.03, which means that these two clusters have only a very low relationship.

Figure 3-3. Example of a clustering tree

Within each cluster, entities are also clustered in a hierarchical structure. Take cluster C3 as an example. Entities 11 and 16 are grouped in the lowest layer with the highest resemblance coefficient of 0.64; entity 14 is grouped with the cluster of entities (11, 16) with a resemblance coefficient of 0.53; and entity 18 is grouped with the cluster of

entities (11, 16, 14) with a resemblance coefficient of 0.32. The resemblance coefficient reflects the level of relatedness or similarity between those entities.

3.5.2 Tree analysis

The next step is to analyze the clustering tree. Three issues are discussed regarding the analysis.

Control entities. In the clustering tree, there is no overlap among entities; that is, one entity can only belong to one cluster. In the real code, however, a control entity may be shared by different non-control entities that participate in different activities within different clusters. Control entities that are nested or share the same data attributes tend to be grouped together. However, control entities do not have a direct contribution to a functional activity. In most cases a cluster that contains only control entities cannot be extracted as a new function. In Figure 3-3, cluster C2 includes entities 9, 12, and 15, all of which are control entities. These entities are nested control entities and therefore they are grouped together, but they cannot form a new function. However, the control entities generally need to be used for those non-control entities underneath. In the case that control entities are separated from non-control entities, it is the software engineer's responsibility to decide which control entity should go to which cluster or clusters.

Clusters. In the clustering tree, related or similar entities are grouped together to form a cluster. The resemblance coefficient shows the level of relatedness or similarity. However, the number of clusters is determined by the cut-point. In Figure 3-3, cut-point *cut1* gives two clusters, C1 and C6. Cut-point *cut2* gives four clusters, C1, C2, C3 and C4. For cut-point *cut3*, there are eight clusters. Different cut-points produce different numbers of clusters. Generally, the more clusters there are, the more new functions with

high cohesion may be composed. But at the same time, the coupling among functions increases.

In Figure 3-3, if *cut2* is chosen as the cut-point, without considering cluster C2 because it consists only of control entities, there are three clusters, C1, C3 and C4. Cluster C1 is related to the activity of reading the input data *sale*. Cluster C3 is related to the activity of computing *pay*, and cluster C4 is related to the activity of computing *profit*. These three clusters can be translated into three new cohesive functions. In such a case, control entity 15 should be grouped to C3, since it is tightly related to the computing *pay*. Similarly, control entity 12 should be used for both C3 and C4, because both C3 and C4 belong to the same *for* loop. Control entity 9 is kept in the original function. The outcome is demonstrated in Figure 3-4 in the next section.

On the other hand, if *cut3* is selected as the cut-point, without considering all the control entities, there are five clusters. Cluster 1 contains entity 7, which is related to the activity of reading the input data *sale*. Cluster 2 includes entities (11, 14, 16), which are related to the activity of computing *total_pay*. Cluster 3 is a cluster with a single entity 18, which is concerned with the activity of computing *pay* (average pay). Cluster 4 contains entities (10, 13), which are related to the activity of computing *total_sale*. Cluster 5 is also a single entity (entity 19) cluster, related to the activity of computing *profit*. All five clusters can be extracted to five new high-cohesive functions, as shown in Figure 3-5 in the next section. The result from *cut3* is the same as the result in the paper by [Lak99]. Cut-point *cut3* generates more functions with higher cohesion than *cut2* does, but it also results in high coupling and more overhead from a performance aspect. Therefore designer interactions are recommended.

Interleaved entities. As mentioned above, a cluster contains related entities. If there is no interleaved code, the cluster corresponds to a contiguous fragment of code. In Figure 3-3, cluster C1 contains entities 5 and 7. Because there is no entity 6 in the analysis, C1 represents a contiguous fragment of code from line 5 to line 7 in the source code. Cluster C3 contains entities from line 11 to line 18 in the source code, and cluster C4 includes entities from line 10 to line 19. It is apparent that there is interleaved code between these two clusters, which also means that two different activities are interleaved in the source code.

In a word, there is no strict rule used for analyzing a clustering tree. The tree only provides heuristic advice, and the final decision on restructuring should be made by software engineers, based on the restructuring objective and their skills, experience and intuitions.

3.6 Restructuring process

After the restructuring candidates have been decided, the real restructuring process begins with decomposing the original low-cohesive function, based on the clustering tree. Some clusters in the tree can be composed into new cohesive functions. This process includes selecting new meaningful function names, and input and output parameters. Some code fragments in the original function are replaced with the new function call statements. This process is done manually.

Based on the clustering result shown in Figure 3-3, suppose *cut2* is chosen as the cut-point, three clusters C1, C3, and C4 are extracted from the original code and composed into three new functions. Cluster C1 forms a new function *read_input()*. Clusters C3 and C4 form two new functions, *compute_average_pay()* and *compute_profit()*, respectively.

The result is shown in Figure 3-4. After restructuring, the function *sale_pay_profit()* clearly shows that three functional activities are involved, each of them is implemented in one cohesive function. The program becomes easy to understand and maintain. If *cut3* is selected as the cut-point, two more new functions are composed, *compute_total_pay()* and *compute_total_sale()*. The result is shown in Figure 3-5.

```

1  procedure sale_pay_profit(days:integer; cost: float;
                                var sale: int_array; var pay: float;
                                var profit: float; process: boolean);
2  begin
3      read_input(days, sale);
4      if process = True then begin
5          compute_average_pay(days, sale, pay);
6          compute_profit(days, cost, sale, profit);
7      end;
8  end;

9  procedure read_input(days:integer; var sale: int_array);
10 var i: integer;
11 begin
12     i := 0;
13     while i < days do begin
14         i := i + 1;
15         readln(sale[i]);
16     end;
17 end;

18 procedure compute_average_pay(days:integer; sale: int_array;
                                var pay: float);
19 var i: integer; total_pay: float;
20 begin
21     total_pay := 0;
22     for i:=1 to days do begin
23         total_pay := total_pay + 0.1*sale[i];
24         if sale[i] > 1000 then
25             total_pay := total_pay + 50;
26         end;
27     pay := total_pay/days + 100;
28 end;

29 procedure compute_profit(days:integer; cost: float;
                                sale: int_array; var profit: float);
30 var i: integer; total_sale: float;
31 begin
32     total_sale := 0;
33     for i:=1 to days do begin
34         total_sale := total_sale + sale[i];
35     end;
36     profit := 0.9*total_sale - cost;
37 end;

```

Figure 3-4. Restructuring result from *cut2* for sample code 1 in Figure 3-2

```

1  procedure sale_pay_profit(days:integer; cost: float;
                               var sale: int_array; var pay: float;
                               var profit: float; process: boolean);
2  begin
3      read_input(days, sale);
4      if process = True then begin
5          compute_average_pay(days, sale, pay);
6          compute_profit(days, cost, sale, profit);
7      end;
8  end;

9  procedure read_input(days:integer; var sale: int_array);
10 var i: integer;
11 begin
12     i := 0;
13     while i < days do begin
14         i := i + 1;
15         readln(sale[i]);
16     end;
17 end;

18 procedure compute_average_pay(days:integer; sale: int_array;
                                   var pay: float);
19 var total_pay: float;
20 begin
21     compute_total_pay(days, sale, total_pay);
22     pay := total_pay/days + 100;
23 end;

24 procedure compute_profit(days:integer; cost: float;
                               sale: int_array; var profit: float);
25 var total_sale: float;
26 begin
27     compute_total_sale(days, sale, total_sale);
28     profit := 0.9*total_sale - cost;
29 end;

30 procedure compute_total_pay(days:integer; sale: int_array;
                                   var total_pay: float);
31 var i: integer;
32 begin
33     total_pay := 0;
34     for i:=1 to days do begin
35         total_pay := total_pay + 0.1*sale[i];
36         if sale[i] > 1000 then
37             total_pay := total_pay + 50;
38         end;
39 end;

40 procedure compute_total_sale(days:integer; sale: int_array;
                                   var total_sale: float);
41 var i: integer;
42 begin
43     total_sale := 0;
44     for i:=1 to days do begin
45         total_sale := total_sale + sale[i];
46     end;
47 end;

```

Figure 3-5. Restructuring result from *cut3* for sample code 1 in Figure 3-2

Chapter 4 Experiments on Similarity Measure

Section 3.4 defines a new resemblance coefficient, but the weights of the attributes have not been given. No systematic study has been carried out on this issue in the existing research. Dhama [Dha95] uses a heuristic estimate to give the data parameters twice as much weight as the control parameters. Schwanke [Sch91] estimates the significance of a feature using Shannon information content, which gives rarely-used identifiers higher weight than frequently-used identifiers. This chapter demonstrates an extensive empirical study of the impact of different weight ratios on the clustering results.

In the equation (3-1), the weights of the attributes are considered to be positive integers and decided by means of experiments. The experiments focus on the different weight ratios between the data attributes and the control attributes. Over 40 functions appearing in papers by [Bie94, Bie98, Kim94, Lak99], student assignments, and real industrial programs are used in the experiments. The size of each function ranges from 8 lines of code (LOCs) to 182 LOCs (not including comments and white spaces). Because the 1:1 weight ratio is equivalent to the Jaccard coefficient, the experiments begin with the Jaccard coefficient. The Sorenson coefficient is also validated together with the Jaccard coefficient.

During the experiment, three clustering algorithms, SLINK, CLINK and UPGMA, are used. CLINK tends to form large number of compact clusters and only yields good results for small examples. SLINK and UPGMA give similar results during the experiments. The results shown in this chapter are generated by the UPGMA algorithm.

4.1 Jaccard coefficient and Sorenson coefficient

The Jaccard coefficient and the Sorenson coefficient are described in section 2.3.2.2.

Both of them treat all the attributes equally; that is, the data attributes and the control attributes are treated the same way. In such cases, there are only 1-1 matches, 0-0 matches, 1-0 and 0-1 matches. A 1-1 match means that two entities share a common attribute, no matter whether it is a data attribute or a control attribute. A 1-0 / 0-1 match means that an attribute is present in one entity but absent in the other. Take the sample code 1 in Figure 3-2 as an example: the entity-attribute matrix in Table 3-2 is modified to form a new matrix, as shown in Table 4-1. Taking the matrix in Table 4-1 as the input data matrix, the clustering trees obtained by the Jaccard coefficient and the Sorenson coefficient are illustrated in Figure 4-1 and 4-2, respectively.

Table 4-1. Entity-attribute matrix of sample code 1 in Figure 3-2 in case where data attributes and control attributes are treated equally

Entity	Attribute												
	<i>days</i>	<i>readln</i>	<i>sale</i>	<i>process</i>	<i>total_ sale</i>	<i>total_ pay</i>	<i>pay</i>	<i>profit</i>	<i>cost</i>	<i>loop1</i>	<i>if1</i>	<i>loop2</i>	<i>if2</i>
5	1	0	0	0	0	0	0	0	0	1	0	0	0
7	0	1	1	0	0	0	0	0	0	1	0	0	0
9	0	0	0	1	0	0	0	0	0	0	1	0	0
10	0	0	0	0	1	0	0	0	0	0	1	0	0
11	0	0	0	0	0	1	0	0	0	0	1	0	0
12	1	0	0	0	0	0	0	0	0	0	1	1	0
13	0	0	1	0	1	0	0	0	0	0	1	1	0
14	0	0	1	0	0	1	0	0	0	0	1	1	0
15	0	0	1	0	0	0	0	0	0	0	1	1	1
16	0	0	0	0	0	1	0	0	0	0	1	1	1
18	1	0	0	0	0	1	1	0	0	0	1	0	0
19	0	0	0	0	1	0	0	1	1	0	1	0	0

Figures 4-1 and 4-2 show two very similar trees. The only difference between them is the resemblance coefficients. In the Jaccard coefficient, the same weight is given to the 1-1 match as to the 1-0 / 0-1 match. In the Sorenson coefficient, the 1-1 match is weighted twice as much as the 1-0 / 0-1 match. Therefore, the Sorenson coefficient has higher

Figure 4-1. Tree obtained with Jaccard coefficient for sample code 1 in Figure 3-2

Figure 4-2. Tree obtained with Sorenson coefficient for sample code 1 in Figure 3-2

values than the Jaccard coefficient. However, in this sample code, the same tree structure is shown. Both of them form four clusters: C1, C2, C3 and C4. In cluster C1, entities 5 and 7 are grouped together. This group is correct because they are related to the activity of reading the input data *sale*. In cluster C4, entities 13 and 14 are grouped together, but in fact they are not related, because entity 13 is involved with computing *total_sale*, while entity 14 computes *total_pay*. They are clustered together incorrectly because they share three common attributes with one data attribute and two control attributes, and both the Jaccard coefficient and the Sorenson coefficient treat data attributes and control attributes equally. In the source code, entities 10 and 13 are closely related to the

computation of *total_sale*, which is also related to entity 19, because entity 19 uses *total_sale* to compute *profit*. Entities 11, 14 and 16 are closely related to the computation of *total_pay*, which is also related to entity 18, because entity 18 uses *total_pay* to compute average *pay*. However, the trees in Figures 4-1 and 4-2 do not reflect the real functional structure of the code.

The main limitation of the Jaccard coefficient and the Sorenson coefficient in program restructuring at the function level is that they do not weight attributes differently. The Sorenson coefficient gives different weights to different matches, which deals with the weighting of matches instead of the weighting of attributes. From the cohesion point of view, the two types of attributes contribute to cohesion differently because they describe different dependence relationships between entities. Therefore, the weighting of the attributes is important in program restructuring. In the following sections, different weight ratios are used to validate how the weights affect the similarity measure and to identify a suitable weight ratio from various experiments.

4.2 Experiment with weight ratio of 2:1

In this section, the data attributes are given twice as much weight as the control attributes; that is, in the equation (3-1), $w_d = 2$ and $w_c = 1$.

Figure 4-3 is the clustering tree, obtained by using a 2:1 weight ratio for the sample code 1 in Figure 3-2. It shows three clusters: C1, C2 and C3. Cluster C1 includes entities (5, 7), which are related to the activity of reading the input data *sale*. Cluster C2 contains entities (11, 14, 16, 18), which contribute to the functional activity of the computation of *pay*. Cluster C3 has entities (10, 13, 19), which are involved in the functional activity of the computation of *profit*. Entities 9 and 12 are control entities that do not directly

participate in a specific activity. They are shared by two activities: the computation of *pay* and the computation of *profit*. Control entity 15 is related to the activity of computing *pay*. The tree reveals the real functional structure, and the 2:1 weight ratio works well in this example.

Figure 4-3. Tree obtained with a 2:1 weight ratio for sample code 1 in Figure 3-2

Now let us take another example. The sample code 2 in Figure 4-4 is a communicational cohesion example presented in [Bie98]. It involves two activities – one is the computation of product and the other is the computation of sum and average. Because both output variables, *sum* and *prod*, use the same input variable *arr*, this function is considered to have communicational cohesion [Bie98]. In the function, there are six entities: five non-control entities (4, 5, 7, 8, 10) and one control entity 6. There are also six attributes: five data attributes (*n*, *arr*, *sum*, *prod*, *avg*) and one control attribute *for*. The entity-attribute matrix is shown in Table 4-2. Taking the matrix in Table 4-2 as the input data, the clustering tree obtained by the resemblance coefficient in the equation (3-1) with a 2:1 weight ratio is illustrated in Figure 4-5.

```

1 procedure sum_and_prod(n: integer;
                        arr: int_array;
                        var sum, prod: integer;
                        var avg: float);
2   var i: integer;
3   begin
4     sum := 0;
5     prod := 1;
6     for i:=1 to n do begin
7       sum := sum + arr[i];
8       prod := prod * arr[i];
9     end;
10    avg := sum/n;
11 end;

```

Figure 4-4. Sample code 2: Sum and Product [Bie98]

Table 4-2. Entity-attribute matrix of sample code 2 in Figure 4-4

Entity	Attribute					
	<i>n</i>	<i>arr</i>	<i>sum</i>	<i>prod</i>	<i>avg</i>	<i>for</i>
4	0	0	2	0	0	0
5	0	0	0	2	0	0
6	1	0	0	0	0	1
7	0	2	2	0	0	1
8	0	2	0	2	0	1
10	2	0	2	0	2	0

Figure 4-5. Tree obtained with a 2:1 weight ratio for sample code 2 in Figure 4-4

Figure 4-5 shows that entities 7 and 8 are grouped together, and entities 4 and 10 are grouped together, but in fact, entities 4 and 7 are related to the same functional activity – the computation of *sum*. Entity 10 uses the result of *sum* to compute the average *avg*. Entities 5 and 8 contribute to the same functional activity – the computation of product *prod*. The tree obtained with a weight ratio of 2:1 does not reveal the real functional structure in this example.

In order to find the real reason for this, let us examine the resemblance coefficients between those entities with a 2:1 weight ratio.

$$\text{Resemblance coefficient between entities 4 and 7: } coeff_{(4,7)} = \frac{2 \times 1}{(2 \times 1) + (2 \times 1) + (1 \times 1)} = 0.40 .$$

$$\text{Resemblance coefficient between entities 5 and 8: } coeff_{(5,8)} = \frac{2 \times 1}{(2 \times 1) + (2 \times 1) + (1 \times 1)} = 0.40 .$$

$$\text{Resemblance coefficient between entities 7 and 8: } coeff_{(7,8)} = \frac{(2 \times 1) + (1 \times 1)}{(2 \times 1) + (1 \times 1) + (2 \times 2)} = 0.43 .$$

Because $coeff_{(7,8)} > coeff_{(4,7)}$ and $coeff_{(7,8)} > coeff_{(5,8)}$, entities (7, 8) are clustered together instead of entities (4, 7) and entities (5, 8). Although the data attributes are weighted twice as heavily as the control attributes, it seems that the control attributes still play more of a role in the similarity measure than they should. In the next section, a larger weight ratio is used in the experiments.

4.3 Experiment with weight ratio of 3:1

In this section, the data attributes are weighted three times as much as the control attributes. In other words, in the equation (3-1), $w_d = 3$ and $w_c = 1$.

Figure 4-6 is the clustering tree obtained with a 3:1 weight ratio for the sample code 1 in Figure 3-2. It shows three clusters similar to those in Figure 4-3. Although the

resemblance coefficients and the positions of the clusters in the two trees are different, the clusters in Figure 4-6 also illustrate the real functional structure of the code. The 3:1 weight ratio also works well in this example.

Figure 4-6. Tree obtained with a 3:1 weight ratio for sample code 1 in Figure 3-2

Figure 4-7 is the clustering tree obtained with a 3:1 weight ratio for the sample code 2 in Figure 4-4. It shows a tree structure different to the tree illustrated in Figure 4-5. In Figure 4-7, there are two clusters: C1 and C2. Cluster C1 has three entities (4, 7, 10), which are related to the computation of *sum* and *avg*. Cluster C2 consists of two entities (5, 8), which are related to the computation of product *prod*. Entity 6 is a control entity that is shared by two computation activities. The tree shows the real functional structure. The 3:1 weight ratio works well in this example.

Figure 4-7. Tree obtained with a 3:1 weight ratio for sample code 2 in Figure 4-4

With a 3:1 weight ratio, the clustering result of the sample code 2 is totally different from the result obtained with a 2:1 weight ratio. The resemblance coefficients of the entity pairs (4, 7), (5, 8) and (7, 8) are as follows:

$$\text{Resemblance coefficient between entities 4 and 7: } coeff_{(4,7)} = \frac{3 \times 1}{(3 \times 1) + (3 \times 1) + (1 \times 1)} = 0.43 .$$

$$\text{Resemblance coefficient between entities 5 and 8: } coeff_{(5,8)} = \frac{3 \times 1}{(3 \times 1) + (3 \times 1) + (1 \times 1)} = 0.43 .$$

$$\text{Resemblance coefficient between entities 7 and 8: } coeff_{(7,8)} = \frac{(3 \times 1) + (1 \times 1)}{(3 \times 1) + (1 \times 1) + (3 \times 2)} = 0.40 .$$

Here, $coeff_{(4,7)} > coeff_{(7,8)}$ and $coeff_{(5,8)} > coeff_{(7,8)}$, so entities (4, 7) and entities (5, 8) are grouped together, respectively.

Now let us look at a more complex example. The sample code 3 in Figure 4-8 is an example taken from an industrial program. It is an implementation of processing the body of a function based on the token type in a C code parser program. The main functional activity is to process the token body with an unreserved token type, which is implemented in the source code from line 26 to line 57. In the function, there are 23 entities and 31 attributes, including 11 control attributes.

Figure 4-9 shows the clustering analysis result with a weight ratio of 3:1. The cluster C1 is related to the activity of processing the body with an unreserved token type, which should be involved with entities from entity 26 to entity 54, as mentioned earlier. Entity 26 is grouped with entities 18 and 20 because they share the same control attribute *token_type*. But entities 16 and 19 interleave cluster C1. Entity 16 merges with this cluster by sharing a common data attribute *token* with entities (30, 38, 43, 45). Entity 19 joins cluster C1 by sharing a common data attribute *cntl_flag* with entities (32, 36, 40, 50). This shows that data attributes with a 3:1 weight ratio may play more of a role in the

```

1 process_body (char[] token, int *token_type, int *cntl_flag,
                int *strcpy_flag, int equal_flag, int line_no)
2 {
3     int position;
4     int check_type_process_reserved ();
5     int search_local_list ();
6     int search_decl_keywords ();
7     int search_decl_user ();
8
9     //...
10
16    *token_type = check_type_process_reserved (token);
17
18    if (*token_type == CNTL_KEY)
19        *cntl_flag = TRUE;
20    else if (*token_type == LIBRARY_FUNC) {
21        if (strcmp (token, "strcpy") == 0)
22            *strcpy_flag = TRUE;
23        else
24            ; /* to avoid ambiguity of nested if */
25    }
26    else if (*token_type == IDENTIFIER) { /* not a reserved word */
27        if (! search_decl_keywords (token) && ! search_decl_user (token)){
28            //...
29
30            position = search_local_list (token);
31            if (position != -1) { /* found in the local list */
32                update_local_list (position, *cntl_flag, line_no);
33
34                //...
36                update_para_list (*strcpy_flag, equal_flag, *cntl_flag,
                                position);
37            } else {
38                position = search_global_list (token);
39                if (position != -1) {
40                    update_global_list (position, *cntl_flag, line_no);
41
42                    if (global_list [position].type == GLOBAL)
43                        put_token_into_local_list (token, GLOBAL);
44                    else if (global_list [position].type == FUNCTION)
45                        put_token_into_local_list (token, FUNCTION);
46
47                    //...
49                    position = local_count - 1;
50                    update_local_list (position, *cntl_flag, line_no);
51                }
52            } /* end of outer if (position != -1) */
53            if (*strcpy_flag)
54                *strcpy_flag = FALSE;
55
56        } /* end of if (!search_decl_keywords ...) */
57    } /* end of if (*token_type == IDENTIFIER) */
58 }

```

Figure 4-8. Sample code 3: Process Body

similarity measure than they should. In the next four sections, different weight ratios, between 2:1 and 3:1, are chosen for the experiments.

Figure 4-9. Tree obtained with a 3:1 weight ratio for sample code 3 in Figure 4-8

4.4 Experiment with weight ratio of 9:4

In this section, the data attributes are weighted at 9 and the control attributes are at 4; that is, in the equation (3-1), $w_d = 9$ and $w_c = 4$.

Figure 4-10 is the clustering tree obtained with a weight ratio of 9:4 for the sample code 1 in Figure 3-2. It shows three clusters identical to those displayed in Figure 4-6. The weight ratio of 9:4 also works well in the sample code 1.

Figure 4-10. Tree obtained with a 9:4 weight ratio for sample code 1 in Figure 3-2

Figure 4-11 is the clustering tree obtained with a weight ratio of 9:4 for the sample code 2 in Figure 4-4. It shows the same problem as in Figure 4-5. Two unrelated entities, 7 and 8, are grouped together. The weight ratio of 9:4 does not work well with the sample code 2.

Figure 4-11. Tree obtained with a 9:4 weight ratio for sample code 2 in Figure 4-4

4.5 Experiment with weight ratio of 7:3

In this section, the data attributes are weighted at 7 and the control attributes are at 3; that is, in the equation (3-1), $w_d = 7$ and $w_c = 3$.

Figure 4-12 is the clustering tree obtained with a weight ratio of 7:3 for the sample code 1 in Figure 3-2. It shows three clusters identical to those displayed in Figure 4-6. The weight ratio of 7:3 also works well with the sample code 1.

Figure 4-12. Tree obtained with a 7:3 weight ratio for sample code 1 in Figure 3-2

Figure 4-13 is the clustering tree obtained with the weight ratio of 7:3 for the sample code 2 in Figure 4-4. It shows the same problem as in Figure 4-5. Two unrelated entities, 7 and 8, are grouped together. The weight ratio of 7:3 does not work well with the sample code 2 either.

Figure 4-13. Tree obtained with a 7:3 weight ratio for sample code 2 in Figure 4-4

4.6 Experiment with weight ratio of 5:2

In this section, the data attributes are weighted at 5 and the control attributes are at 2; that is, in the equation (3-1), $w_d = 5$ and $w_c = 2$.

Figure 4-14 is the clustering tree obtained with a weight ratio of 5:2 for the sample code 1 in Figure 3-2. It shows three clusters identical to those displayed in Figure 4-6. The weight ratio of 5:2 also works well with the sample code 1.

Figure 4-14. Tree obtained with a 5:2 weight ratio for sample code 1 in Figure 3-2

Figure 4-15 is the clustering tree obtained with a weight ratio of 5:2 for the sample code 2 in Figure 4-4. It shows two clusters identical to those illustrated in Figure 4-7. The 5:2 weight ratio works well with the sample code 2.

Figure 4-15. Tree obtained with a 5:2 weight ratio for sample code 2 in Figure 4-4

Figure 4-16 is the clustering tree obtained with a weight ratio of 5:2 for the sample code 3 in Figure 4-8. It shows that cluster C1 contains the exact entities that are related to the activity of processing token body with an unreserved token type. No entities interleave in the cluster. Entities 16 and 19, which are inside cluster C1 in Figure 4-9 with a weight ratio of 3:1, are outside the cluster here. This is because the weight of the data attributes has been reduced. The relationship between entity 16 and entities (30, 38, 43, 45), due to sharing a common data attribute *token*, becomes weaker and entity 16 is separated from cluster C1. Similarly, the relationship between entity 19 and entities (32, 36, 40, 50) also becomes weaker, and entity 19 is pushed out of cluster C1. The clustering tree reveals the real functional structure of the sample code 3. The weight ratio of 5:2 works well in this example, and it also gives the expected results for all the functions selected in the experiments.

Figure 4-16. Tree obtained with a 5:2 weight ratio for sample code 3 in Figure 4-8

4.7 Experiment with weight ratio of 8:3

In this section, the data attributes are weighted at 8 and the control attributes are at 3; that is, in the equation (3-1), $w_d = 8$ and $w_c = 3$.

Figure 4-17 is the clustering tree obtained with a weight ratio of 8:3 for the sample code 1 in Figure 3-2. It shows three clusters identical to those displayed in Figure 4-6. The weight ratio of 8:3 also works well with the sample code 1.

Figure 4-17. Tree obtained with an 8:3 weight ratio for sample code 1 in Figure 3-2

Figure 4-18 is the clustering tree obtained with a weight ratio of 8:3 for the sample

code 2 in Figure 4-4. It shows two clusters identical to those illustrated in Figure 4-7. The 8:3 weight ratio works well with the sample code 2.

Figure 4-18. Tree obtained with an 8:3 weight ratio for sample code 2 in Figure 4-4

Figure 4-19 is the clustering tree obtained with a weight ratio of 8:3 for the sample code 3 in Figure 4-8. It shows a tree structure identical to that displayed in Figure 4-16 and the resemblance coefficients in the two trees are also very close. The tree illustrates the real functional structure of the sample code 3. Therefore, the weight ratio of 8:3 also works well in this example. The ratio is also applied to all selected examples and gives the expected results.

Figure 4-19. Tree obtained with an 8:3 weight ratio in sample code 3 in Figure 4-8

In summary, six different weight ratios were used in the experiments. The weight ratios of 2:1, 9:4 and 7:3 do not work with the sample code 2. The weight ratio of 3:1 works well with the sample code 2, but does not work well with the sample code 3. Both the weight ratios of 5:2 and 8:3 work very well with all three sample codes and with other examples, and give very close results. In the restructuring approach, therefore, the ratio of 8:3 is chosen to weight the data attributes and the control attributes in the similarity measure.

4.8 Experiment with data attributes in control entities

The data attributes in the control entities are treated as control attributes, which are discussed in section 3.4.1. This section uses some examples to show the effect on the similarity measure if the data attributes in the control entities are not treated as control attributes. The results support the rationale concerning the data attributes in the control entities treated as control attributes, as discussed in section 3.4.1.

First, take the sample code 1 in Figure 3-2 as an example. There is a data attribute *days* in control entities 5 and 12. When it is counted as a data attribute in both control entities, the input entity-attribute matrix in Table 3-2 is modified to the matrix in Table 4-3. The value of *days* in control entities 5 and 12 becomes 2.

The tree obtained from the input data in Table 4-3 with a weight ratio of 8:3 is shown in Figure 4-20. Entities 5 and 7 in the original cluster C1 are separated. Two control entities (5 and 12) are grouped together because they share the same variable *days*, which is treated as a data attribute with more weight. The original *while* loop block for reading the input data *sale* is not illustrated in the tree. Entity 5 is connected to cluster C2, but there is no relationship between them. The tree does not show the real functional

structure of the sample code 1.

Table 4-3. Entity-attribute matrix of sample code 1 in Figure 3-2 with a state value of 2 for data attributes in control entities

Entity	Attribute												
	Data attribute									Control attribute			
	<i>days</i>	<i>readln</i>	<i>sale</i>	<i>process</i>	<i>total_sale</i>	<i>total_pay</i>	<i>pay</i>	<i>profit</i>	<i>cost</i>	<i>loop1</i>	<i>if1</i>	<i>loop2</i>	<i>if2</i>
5	2	0	0	0	0	0	0	0	0	1	0	0	0
7	0	2	2	0	0	0	0	0	0	1	0	0	0
9	0	0	0	1	0	0	0	0	0	0	1	0	0
10	0	0	0	0	2	0	0	0	0	0	1	0	0
11	0	0	0	0	0	2	0	0	0	0	1	0	0
12	2	0	0	0	0	0	0	0	0	0	1	1	0
13	0	0	2	0	2	0	0	0	0	0	1	1	0
14	0	0	2	0	0	2	0	0	0	0	1	1	0
15	0	0	1	0	0	0	0	0	0	0	1	1	1
16	0	0	0	0	0	2	0	0	0	0	1	1	1
18	2	0	0	0	0	2	2	0	0	0	1	0	0
19	0	0	0	0	2	0	0	2	2	0	1	0	0

Figure 4-20. Tree obtained with a state value of 2 for data attributes in control entities in sample code 1 in Figure 3-2

Now take the sample code 2 in Figure 4-4 as an example. There is a data attribute *n* in

control entity 6. If it is treated as a data attribute in control entity 6, the input entity-attribute matrix in Table 4-2 is modified to the matrix in Table 4-4. The value of n in control entity 6 changes from 1 to 2. The tree obtained from this input data with a weight ratio of 8:3 is shown in Figure 4-21.

Table 4-4. Entity-attribute matrix of sample code 2 in Figure 4-4 with a state value of 2 for data attributes in control entities

Entity	Attribute					
	n	arr	sum	$prod$	avg	for
4	0	0	2	0	0	0
5	0	0	0	2	0	0
6	2	0	0	0	0	1
7	0	2	2	0	0	1
8	0	2	0	2	0	1
10	2	0	2	0	2	0

In Figure 4-21, entity 10 in the original cluster C1 is separated from the cluster and grouped together with entity 6. This is because entities 6 and 10 share the same variable n , which is treated as a data attribute with more weight. They are not in the same control block and there is no relationship, but they are connected together. Therefore, the tree does not reflect the real functional structure of the sample code 2.

Figure 4-21. Tree obtained with a state value of 2 for data attributes in control entities in sample code 2 in Figure 4-4

Both examples illustrate the problems when the data attributes in the control entities are also counted as data attributes with more weight. In such cases, control entities that share common data attributes have a more potential to be clustered together. A control entity also tends to be grouped together with unrelated non-control entities when they share the same data attributes. Therefore the data attributes in the control entities are treated as control attributes with less weight in the restructuring approach.

Chapter 5 Case Study

This chapter presents a case study of program restructuring for a real industrial program, using the proposed restructuring approach described in Chapter 3, which evaluates how effectively the approach can be applied to the real software industry.

5.1 Background of case study

The problem in the case study is a real network protocol RSVP-TE [Awd01] program in the telecommunication industry. RSVP [Bra97] is a resource reservation control protocol that enables Internet applications to obtain different qualities of service (QoS). RSVP-TE is a signaling protocol that extends the RSVP to support multiple protocol label-switching (MPLS) [Ros01] traffic engineering applications. RSVP-TE provides a mechanism to establish and maintain explicitly routed label switched paths (LSPs) [Ros01], with or without resource reservation. RSVP-TE has two fundamental messages: the Path message and the Resv (reservation request) message, which are used to set up LSPs and also used as refresh messages to maintain existing LSPs. In addition, there are also PathTear (path teardown), ResvTear (reservation teardown), PathErr (path error) and ResvErr (reservation error) messages. The PathTear and ResvTear messages are used to tear down existing LSPs and release reserved resources. The PathErr and ResvErr messages deal with the errors that occur during Path and Resv message-processing, respectively.

The original RSVP-TE program was completed on a schedule-driven basis. During design it was provided only with basic functionalities, and then more functionalities were added during conformance testing in order to satisfy the specifications of the protocol.

The emphasis of the software development is on functionality. The RSVP-TE program was written in C with about 6,500 LOCs (in this chapter, all lines of code do not include comments and white spaces) and 110 functions. Some functions are large and involve more than one functional activity. The understandability of the code was not good. Maintaining and extending the code for additional functionality was less than desirable. In order to improve the design quality, 24 functions with a total of 2,173 LOCs have been

Table 5-1. Selected functions for restructuring analysis

No.	Function Name	LOCs	Functionality
1	rsvpTeRx	74	Process all received RSVP-TE messages
2	rsvpTeDecodeMsg	283	Decode all received RSVP-TE messages
3	rsvpTeRxPath	104	Process Path message
4	rsvpTeProcessERO	118	Process explicit route object (ERO) in Path message
5	rsvpTeUpdateERO	42	Update ERO in path state block (PSB)
6	rsvpTeProcessRRO	51	Process recorded route object (RRO) in Path or Resv message
7	rsvpTeUpdateRRO	42	Update RRO in both PSB and reservation state block (RSB)
8	rsvpTeBuildPSB	110	Build a new PSB
9	rsvpTeUpdatePSB	76	Update an existing PSB
10	rsvpTeBuildPathMsg	91	Build Path message
11	rsvpTeRxResv	56	Process Resv message
12	rsvpTeReserve	100	Make resource reservation on an interface
13	rsvpTeResvRefresh	57	Refresh all upstream nodes in the session
14	rsvpTeProcessFlowDescriptor	98	Process flow descriptor in Resv message
15	rsvpTeBuildRSB	32	Build a new RSB
16	rsvpTeUpdateRSB	58	Update an existing RSB
17	rsvpTeRSBAddFilter	50	Add a filter block to an RSB
18	rsvpTeBuildResvMsg	164	Build a Resv message
19	rsvpTeRxPTear	70	Process PathTear message
20	rsvpTePSBTimeout	44	Delete an existing PSB
21	rsvpTeRxRTear	191	Process ResvTear message
22	rsvpTeResvTearFD	70	Tear down one filter block
23	rsvpTeRxPErr	52	Process PathErr message
24	rsvpTeRxRErr	140	Process ResvErr message
Total		2173	

selected as restructuring candidates, as shown in Table 5-1. The function sizes range from 32 LOCs to 283 LOCs, with an average of 90.54 LOCs. The source code of the selected functions, which is restricted by Eion Inc., is not shown in this thesis.

5.2 Restructuring process

For each selected function shown in Table 5-1, the restructuring process follows the steps of the program restructuring approach outlined in this thesis:

1. Generate input data matrix: the input entity-attribute data matrix is generated from source code using the Parser tool. The matrix is saved in an input file, which will be read by the Clustering tool during clustering analysis.
2. Perform clustering analysis: the Clustering tool computes the resemblance coefficients between entities and forms a clustering tree based on a specified clustering algorithm. Because there is no dominant algorithm that is suitable for all cases, three common algorithms, UPGMA, SLINK and CLINK, are used and compared in the case study.
3. Determine restructuring candidates: the clustering tree produced by the clustering analysis illustrates the existing structure of the function being analyzed, and provides heuristic advice on restructuring. The final decision on restructuring is made after combining the heuristic advice, the restructuring objective and the software engineer's experience, skills and understanding of the program.
4. Restructure poorly designed function: the code of the selected function is decomposed into different fragments, based on the restructuring decision made in step 3. The code that is related to a specific functional activity is composed into a new cohesive function.

After the restructuring analysis is completed, it is found that 17 functions in Table 5-1 involve more than one functional activity. Some of them also have duplicated code or interleaved code. They are difficult to understand and the extensibility is also less than desirable. In order to support long-term maintainability, those 17 functions are selected for restructuring. The restructuring results are illustrated in the next section.

5.3 Results of restructuring

At the end of the restructuring process, 17 functions have been restructured. Inside each function, independent functional activities are extracted to form new cohesive functions. For example, the *rsvpTeRx* function is used to process a received message, which includes the following three functional activities:

- 1) Check message's version, length, destination, etc. This part is extracted as a new function called *rsvpTeMsgSanityCheck*.
- 2) Decode message. If the message is determined to be a legal message after sanity checking, the contents of the message are decoded. This has already been done by a function (*rsvpTeDecodeMsg*) call.
- 3) Process message. After a message has been decoded, it is processed by different function calls based on the message type. This part is extracted as a new function called *rsvpTeProcessMsg*.

After restructuring, two new functions, *rsvpTeMsgSanityCheck* and *rsvpTeProcessMsg*, are generated. The three functional activities in the original function are implemented by the three function calls. The function size is reduced from the original 74 LOCs to 18 LOCs. The complexity decreases from 19 to 5. The function is significantly simplified and easy to understand.

Three metrics, size, cyclomatic complexity and cohesion, are chosen to evaluate the restructuring results. The size and cyclomatic complexity are calculated by the Krakatau metrics tool. The cohesion measure suggested by Anquetil and Lethbridge [Anq03] is used in this case study to measure the function cohesion improvement, in which cohesion of a function is the average resemblance coefficient among all pairs of entities in the function.

Size. Before restructuring, 17 functions have a total of 1,611 LOCs, and the average size of one function is 94.76 LOCs. After those 17 functions are restructured, 34 new functions are added and the total size reaches 1,902 LOCs, increased by 18.06% compared to the restructured code, or 4.48% compared to the whole system. However, the average size per function decreases by 60.65%, from the original 94.76 LOCs to 37.29 LOCs. Table 5-2 gives the results in detail.

Complexity. Although the complexity decreases with size [Fen97], in order to give a quantitative improvement of complexity, McCabe's cyclomatic complexity metric is calculated for each function before and after restructuring. Before restructuring, the average complexity of a function is 19.94. After restructuring, the complexity reduces by as much as 61.45%, with an average value of 7.69. The detailed results are shown in Table 5-2.

Cohesion. After restructuring, 34 new functions are created and each new function represents a specific functionality. On average, two new functions are generated for each original function. The less cohesive an original function, the more new functions are extracted from it. After restructuring, the average cohesion of a function increases by 100%, from the original 0.08 to 0.16. The detailed results are shown in Table 5-2. Here

the cohesion value is very low because it is based on pairwise cohesion between all entities in a function and some entities may not share any common attributes. The cohesion value may not reflect the real degree of cohesion, but it can be used to show the improvement after restructuring.

Table 5-2. Size, complexity and cohesion comparisons before and after restructuring

No.	Before Restructuring				After Restructuring			
	Function Name	LOCs	Complexity	Cohesion	Function Name	LOCs	Complexity	Cohesion
1	rsvpTeRx	74	19	0.048	rsvpTeRx	18	5	0.155
					rsvpTeMsgSanityCheck	51	12	0.081
2	rsvpTeDecodeMsg	283	64	0.182	rsvpTeProcessMsg	30	8	0.248
					rsvpTeDecodeMsg	223	54	0.178
					rsvpTeObjectListSanityCheck	38	8	0.536
					rsvpTeResetRsvpNodePointers	17	1	0.929
					rsvpTeProcessUnknownObj	21	4	0.098
3	rsvpTeRxPath	104	22	0.026	rsvpTeRxPath	64	12	0.038
					rsvpTeVerifyOpaqueObjLength	10	4	0.119
					rsvpTeVerifySndtmpObj	16	5	0.127
					rsvpTeDecodeEROandRROFromPMsg	40	6	0.103
					rsvpTeProcessERO	49	10	0.056
4	rsvpTeProcessERO	118	23	0.090	rsvpTeGetNextHopFromERO	57	12	0.079
					rsvpTeGetNextHopWithoutERO	25	5	0.119
					rsvpTeUpdateERO	27	6	0.167
5	rsvpTeUpdateERO	42	9	0.104	rsvpTeIsEROChanged	31	6	0.189
					rsvpTeUpdateRRO	27	6	0.167
6	rsvpTeUpdateRRO	42	9	0.104	rsvpTeIsRROChanged	31	6	0.189
					rsvpTeBuildPSB	81	7	0.123
7	rsvpTeBuildPSB	110	13	0.123	rsvpTeCreateSB	18	3	0.170
					rsvpTeAddERO2PSB	17	3	0.220
					rsvpTeAddRRO2PSB	15	3	0.234
					rsvpTeRxResv	38	7	0.049
8	rsvpTeRxResv	56	12	0.039	rsvpTeVerifyResvMsg	30	8	0.084
					rsvpTeReserve	69	22	0.040
9	rsvpTeReserve	100	32	0.034	rsvpTeUpdatePhopList	31	9	0.109
					rsvpTeVerifyStyle	27	7	0.154
					rsvpTeResvRefresh	17	3	0.170
10	rsvpTeResvRefresh	57	12	0.119	rsvpTeFindFirstUnknownObj	14	3	0.328
					rsvpTeFindFirstPolicyData	14	3	0.209
					rsvpTeGeneratePhopList	27	6	0.336
					rsvpTeUpdateRSB	39	7	0.136
11	rsvpTeUpdateRSB	58	12	0.091				

No.	Before Restructuring				After Restructuring			
	Function Name	LOCs	Comp- lexity	Cohes- ion	Function Name	LOCs	Comp- lexity	Cohes- ion
12	srvpTeRxPTear	70	13	0.078	rsvpTeUpdateFD	28	5	0.109
					srvpTeRxPTear	39	8	0.058
					rsvpTeForwardPTearMsg	36	5	0.048
13	rsvpTePSBTimeout	44	8	0.106	rsvpTePSBTimeout	16	2	0.257
					rsvpTePSBTimeoutUpdateRSBs	17	4	0.183
					rsvpTePSBTimeoutFreeSB	20	4	0.103
14	rsvpTeRxRTear	191	38	0.062	rsvpTeRxRTear	43	9	0.035
					rsvpTeProcessRTearFlowDescriptor	70	15	0.149
					rsvpTeForwardRTearMsg	99	16	0.121
15	rsvpTeResvTearFD	70	17	0.047	rsvpTeResvTearFD	52	13	0.044
					rsvpTeDeleteFilters	33	6	0.127
16	rsvpTeRxPErr	52	8	0.071	rsvpTeRxPErr	28	5	0.043
					rsvpTeIngressProcessPErr	25	3	0.160
					rsvpTeForwardPErrMsg	12	2	0.176
17	rsvpTeRxRErr	140	28	0.033	rsvpTeRxRErr	50	11	0.030
					rsvpTeDecodeFlowDescriptorFromRErr	28	4	0.150
					rsvpTeEgressProcessLargeRROErr	23	4	0.157
					rsvpTeProcessAdmissionErr	33	5	0.146
					rsvpTeRSBGenRErr	38	10	0.108
Total		1611			1902			
Average		94.76	19.94	0.080		37.29	7.69	0.160

Table 5-3 shows a summary of the comparisons before and after restructuring. In the case study, 17 poorly designed functions are restructured with a total of 1,611 LOCs, which represent 24.78% of the RSVP-TE program. After restructuring, 34 new functions are generated. Compared with the original 17 functions, after restructuring the average size of a function drops by 60.65%, the average complexity decreases by 61.45%, and the average cohesion increases by 100%. The restructuring shows a measurable improvement over the original functions. The complexity improvement is especially significant, which shows that the program understandability and maintainability are greatly improved.

Table 5-3. Comparison summary before and after restructuring

Metrics	Before	After	Increased
Average lines per function	94.76	37.29	-60.65%
Average cyclomatic complexity of a function	19.94	7.69	-61.45%
Average cohesion of a function	0.08	0.16	+100%

5.4 Empirical observations

In the case study, the restructuring approach is applied to 24 functions in the RSVP-TE program. In general the approach works very well. The clustering trees show meaningful clusters and also reveal some problems in the code, such as duplicated code and interleaved code. The trees give heuristic advice that will help software engineers make decisions on restructuring.

5.4.1 Functional clusters

In a clustering tree, related entities are grouped together to form a cluster. If the cluster corresponds to a specific functional activity, it is called a functional cluster. If a clustering tree of a function illustrates several independent clusters, the function may involve several different activities with low cohesion. Therefore, the clustering tree provides software engineers with heuristic advice when considering restructuring.

Figure 5-1 is a part of the clustering tree of the function *rsvpTeRxRErr*, which is a function that processes ResvErr messages. The tree clearly shows four functional clusters, C1, C2, C3 and C4. Cluster C1 represents the activity of generating ResvErr messages, cluster C2 describes the activity of processing admission control errors, cluster C3 is related to the activity of processing RRO errors, and cluster C4 contributes to the activity of decoding flow description objects. Therefore, this function is restructured in

such a way that each cluster forms a new function with a total of four new functions created.

Figure 5-1. Partial tree of *rsvpTeRxRErr*

5.4.2 Duplicated code

A clustering tree also shows some patterns. The same pattern that appears more than once in a clustering tree may illustrate problems related to duplicated code. Figure 5-2 is a part of the clustering tree of the function *rsvpTeProcessERO*, which is a function that processes ERO. In the tree, there are two small clusters, C1 and C2, which have the same pattern. By examining the source code, these two clusters represent exactly the same code, but in different control blocks. Both of them are related to the activity of finding the next hop in cases where there is no ERO. During restructuring, C1 and C2 are extracted to form a new function called *rsvpTeGetNextHopWithoutERO*. The duplicated code in the

Figure 5-2. Partial tree of *rsvpTeProcessERO*

original function is replaced by the same function call.

5.4.3 Interleaved code

Normally, if there is no interleaved code, a cluster corresponds to a contiguous fragment of code, which means all entity numbers are inside a certain range. If an entity number belongs to that range but is not grouped into that cluster, the entity may be an interleaved entity.

Figure 5-3 is a part of the clustering tree of the function *rsvpTeRxPTear*, which is a function that processes PathTear messages. Cluster C1 is related to the activity of building PathTear messages, which corresponds to the real code fragment from line 90 to line 126 in the source code. Entity 92 belongs to this code fragment, but it is not grouped into this cluster. By examining the source code, it is established that entity 92 is an interleaved entity.

Figure 5-3. Partial tree of *rsvpTeRxPTear*

5.4.4 Cut-point

In some cases, there is no single cut-point that can be used to cut the whole tree and get meaningful results. Especially in a large clustering tree, there may exist different cut-points used to cut different branches (functional clusters). Each branch that corresponds

to a specific functionality is cut and composed to a new function.

Take Figure 5-1 as an example. This figure shows that four clusters are generated by two cut-points. The first three clusters, C1, C2 and C3, are created by cut-point *cut1*, and the last cluster C4 is obtained by cut-point *cut2*. If only one cut-point, *cut1*, is chosen, cluster C4 will be separated; that is, a single functional activity will be separated. If *cut2* is selected as a single cut-point, entity 82 will be grouped into cluster C1, and entities 54 and 102 will be grouped into cluster C3. All of those results are undesirable.

5.4.5 Comparison between different clustering algorithms

In the case study, the restructuring approach is used over 24 functions with three clustering algorithms: UPGMA, SLINK and CLINK. Figure 5-4 illustrates the clustering results obtained from these three algorithms for the function *rsvpTeRxResv*. The clustering tree from SLINK clearly shows that there are two big functional clusters: one of them is related to the Resv message sanity check and the other is related to the processing message. The clustering tree from CLINK shows 13 small clusters and the clustering tree from UPGMA is between SLINK and CLINK. Different clustering algorithms generate different clusters. SLINK tends to build a small number of loose clusters, while CLINK tends to form a large number of small but compact clusters. UPGMA provides a compromised method between those two.

(a) UPGMA

(b) SLINK

(c) CLINK

Figure 5-4. Comparison of three algorithms for *rsypTeRxResv*

Table 5-4. Comparison between three clustering algorithms

No.	Function Name	Expected result given by		
		UPGMA	SLINK	CLINK
1	rsvpTeRx	√		
2	rsvpTeDecodeMsg	√		
3	rsvpTeRxPath	√		
4	rsvpTeProcessERO	√		
5	rsvpTeUpdateERO		√	
6	rsvpTeProcessRRO		√	
7	rsvpTeUpdateRRO		√	
8	rsvpTeBuildPSB		√	
9	rsvpTeUpdatePSB	√	√	
10	rsvpTeBuildPathMsg		√	
11	rsvpTeRxResv		√	
12	rsvpTeReserve	√		
13	rsvpTeResvRefresh		√	
14	rsvpTeProcessFlowDescriptor	√	√	√
15	rsvpTeBuildRSB	√	√	
16	rsvpTeUpdateRSB	√		
17	rsvpTeRSBAddFilter		√	
18	rsvpTeBuildResvMsg	√		
19	rsvpTeRxPTear	√		
20	rsvpTePSBTimeout	√		
21	rsvpTeRxRTear	√		
22	rsvpTeResvTearFD		√	
23	rsvpTeRxPErr		√	
24	rsvpTeRxRErr	√		
Total		14	13	1

Table 5-4 gives the comparison results between the three algorithms over 24 functions in the case study. There is no single clustering algorithm that works very well for all the functions. Here “works well” means that the algorithm generates meaningful functional clusters. Of all the 24 functions in the case study, there is only one function for

which all three algorithms give the expected result and two functions for which both UPGMA and SLINK work very well. In total, UPGMA works well for 14 functions and SLINK works well for 13 functions, but CLINK does not work well in the case study.

5.4.6 Limitations

Although clustering analysis in the restructuring approach can show functional clusters and reveal some problems that exist in the source code, there are still some limitations.

Non-functional clusters. A non-functional cluster refers to a cluster that does not contribute to a specific functionality. Examples of non-functional clusters are clusters that contain only control entities, or entities with one attribute, such as the same flag variable, etc. Usually a non-functional cluster is connected to a functional cluster, and both of them together form a more complete functional cluster. It is the software engineer's responsibility to identify whether a cluster is a functional cluster or a non-functional cluster.

Figure 5-5. Example of non-functional clusters

Figure 5-5 is the clustering tree of the function *rsvpTeUpdatePSB*. Cluster C1 is a

non-functional cluster in which all entities contain only one attribute – a flag variable *do_path_refresh*. Cluster C2 is a non-functional cluster in which all entities are control entities. Cluster C4 is also a non-functional cluster in which both entities contain only one attribute – a flag variable *do_resv_refresh*. All these three clusters, C1, C2 and C4, cannot form independent meaningful clusters. Cluster C3 is a functional cluster that is related to a real updating PSB activity. All four clusters and a single entity 106 (a control entity) form one complete functional cluster – updating PSB.

Singleton clusters. A singleton cluster refers to a cluster that contains only one entity. It usually represents a relatively independent entity, such as a control entity, a function call entity or an initialization entity. It is the software engineer's responsibility to decide whether a singleton cluster should be grouped with another cluster or not.

Figure 5-6 is the clustering tree of the function *rsvpTeUpdateERO*. It shows two functional clusters, C1 and C2, and three singleton clusters. Cluster C1 is related to the functional activity that checks whether ERO is changed. Cluster C2 is related to the functional activity that updates ERO. All three singleton clusters (entities 51, 57, and 45) contain a single control entity. After analyzing the source code, entities 51 and 57 go to cluster C2 and entity 45 goes to cluster C1.

Figure 5-6. Example of singleton clusters

Big data structure. In the RSVP-TE program, there is a global variable *rsvpNode*, which is a big data structure (struct) with 52 member variables. Some member variables are also big data structures. *rsvpNode* contains information about LSPs that already exist and are being set up, the most recent message information and other information. In the restructuring approach, a data structure variable is treated as one variable. Therefore different functional activities that are related to different member variables in the same data structure tend to be grouped together.

Figure 5-7 is a partial clustering tree of the function *rsvpTeDecodeMsg*. All entities in Figure 5-7 are assignment statements of *rsvpNode*. Cluster C1 is related to the activity of resetting object pointers in *rsvpNode*; cluster C2 counts the number of illegal objects in a message; and cluster C3 assigns object pointers in the *rsvpNode* to objects in a message. Entity 299 is related to processing unknown objects; entities (314, 320, 92, 66, 67, 70) are relatively independent entities that correspond to relatively independent member variables in *rsvpNode*. However, all these entities are grouped into a big cluster that is related to more functional activities, because they share the same data attribute or variable *rsvpNode*.

Figure 5-7. Example of big data structure

One variable related to more functionalities. In some functions, one variable may

be used in entities that participate in different activities. These entities tend to be grouped together. Figure 5-8 shows the clustering tree of the function *rsvpTeRxPErr*. In *rsvpTeRxPErr*, there is a local variable *len* that is used to represent the length of both the PathTear and PathErr messages. In Figure 5-8, entity 16 is a statement of *len* initialization. Entity 84 is related to the activity of forwarding the PathErr message (cluster C2), but it is grouped to cluster C1, which is related to the activity of initializing the PathTear message.

Figure 5-8. Example of one variable related to more functionalities

Chapter 6 Conclusions

6.1 Summary

This thesis presents a program restructuring approach using clustering techniques, for C programs. The main focus is on the selection of entities and attributes, the similarity measure, resemblance coefficient experiments, and the application of the approach in an industrial program. The main goal of the restructuring approach is to provide automated support for identifying ill-structured, low-cohesive functions and give heuristic restructuring advice in order to improve the cohesion of functions in both the software development and evolution phases.

In the restructuring approach, entities and attributes are executable statements and variables in a function, respectively. From the viewpoints of control dependence and data dependence, entities are divided into control entities and non-control entities. Similarly, attributes are divided into data attributes and control attributes. Based on selected entities and attributes, a new resemblance coefficient is defined in order to measure the similarity between entities regarding cohesion. An intensive set of experiments is performed to study the weight ratio between data attributes and control attributes in the resemblance coefficient. The experiments show that a weight ratio of 8:3 (or 5:2) generates the expected results for all the selected examples in the experiments.

As a case study, the restructuring approach is used to analyze 24 functions of a real telecommunication program and restructure 17 of them. In general, the approach works well. The clustering analysis based on the resemblance coefficient defined in this thesis can pick out high-cohesive functions from low-cohesive functions and reveal existing

problems in the code, such as duplicated code and interleaved code. In addition, three clustering algorithms, UPGMA, SLINK and CLINK, are compared in the case study. The results show that CLINK gives the worst result, UPGMA works well for some cases and SLINK works well for the others. The case study also shows that there may not exist a single cut-point that can be used to cut the whole tree. It is common that only branches, which form functional clusters, are cut to form new cohesive functions, and they may be cut from different cut-points.

In the real code, there are many artifacts, and code is constructed in an *ad hoc* manner. The resemblance coefficient defined in this thesis only considers the main factors related to the functional cohesion. Although the weight ratio between data attributes and control attributes is studied thoroughly, there are still some limitations. There are non-functional clusters and singleton clusters in the clustering tree. Software engineers need to identify which clusters are functional clusters and which are non-functional clusters. They also need to decide where those singleton clusters should be. In addition, big data structures with more independent member variables tend to group different functional activities together.

The main contribution of this thesis is to define and study intensively a new resemblance coefficient as a similarity measure for program restructuring at the function level. The restructuring approach based on this resemblance coefficient is applied to an industrial program. The result shows that the heuristic advice provided by the clustering analysis is helpful for software engineers in making restructuring decisions. This thesis builds the foundation for program restructuring using clustering techniques.

6.2 Future work

Future work will focus on the following areas:

- More experiments on real industrial programs. Although the restructuring approach is applied to 24 functions of a real telecommunication program and works well, this does not guarantee effective scaling up to other industrial programs. More experiments are still needed to validate the approach.
- Evaluation of the results with objective metrics. In this thesis, the clustering results are compared with the expected results. In the future, objective criteria should be studied to evaluate the clustering results.
- Cohesion measure. In the case study, the cohesion measure suggested by Anquetil and Lethbridge [Anq03] is used to compare cohesion improvement before and after restructuring. This measure is not desirable, because it is based on pairwise cohesion. How the quantitative measure of cohesion should be computed is still unresolved and is left for future work.
- Clustering algorithms. The case study favors the UPGMA and SLINK algorithms. However, the results from UPGMA and SLINK are not always same. For some cases, UPGMA gives the best results, but for the other cases SLINK works very well. In the future, these two algorithms and others, such as the weighted pair group arithmetic method, should be tried on more industrial programs.
- Object-oriented program. The current research was carried out for C programs, but the technique can be applied to other languages. However, when the approach is extended to support object-oriented programs, more factors, such as inheritance and polymorphism, need to be taken into consideration.

References

- [Anq03] Anquetil, N. and Lethbridge, T. C. (2003). "Comparative study of Clustering Algorithms and Abstract Representations for Software Remodularisation", *IEE Proceedings on Software*, 150(3), pp.185-201.
- [Anq99] Anquetil, N., Fourier, C. and Lethbridge, T. (1999). "Experiments with Hierarchical Clustering Algorithms as Software Remodularization Methods", *Proceedings of Working Conference on Reverse Engineering*.
- [Arn89] Arnold, R. S. (1989). "Software Restructuring", *Proceedings IEEE*, 77(4), pp.607-617.
- [Awd01] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V. and Swallow, G. (2001). *RSVP-TE: Extensions to RSVP for LSP Tunnels*, RFC 3209.
- [Bie98] Bieman, J. M. and Kang, B.-K. (1998). "Measuring Design-Level Cohesion", *IEEE Transactions on Software Engineering*, 24(2), pp.111-124.
- [Bie94] Bieman, J. M. (1994). "Measuring Functional Cohesion", *IEEE Transactions on Software Engineering*, 20(8), pp.644-657.
- [Bra97] Braden, R., Zhang, L., Berson, S., Herzog, S. and Jamin, S. (1997). *Resource ReSerVation Protocol (RSVP)*, RFC 2205.
- [Bri96] Briand, L., Morasca, S. and Basili, V. (1996). "Property-based software engineering measurement", *IEEE Transaction on Software Engineering*, 22(1), pp.68-86.
- [Chi90] Chikofsky, E. J. and Cross II, J. H. (1990). "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, 7(1), pp.13-17.
- [Cho90] Choi, A. C. and Scacchi, W. (1990). "Extracting and Restructuring the Design

of Large Software Systems”, *IEEE Software*, 7(1), pp.66-71.

- [Chu92] Chu, W. C and Patel, S. (1992). “Software Restructuring by Enforcing Localization and Information Hiding”, *Proceedings of the Conference on Software Maintenance*, pp.165-172.
- [Dha95] Dhama, H.(1995). “Quantitative models of cohesion and coupling in software”, *Journal of Systems and Software*, (29), pp.65-74.
- [Eve74] Everitt, B. (1974). *Cluster Analysis*. Heinemann Educational Books, London.
- [Fen97] Fenton, N. E. and Pfleeger, S. L. (1997). *Software Metrics: A Rigorous and Practical Approach*. PWS Publication.
- [Fow99] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.
- [Gri91] Griswold, W. G. and Notkin, D. (1991). *Program Restructuring as an Aid to Software Maintenance*, PhD Thesis, University of Washington.
- [Hut85] Hutchens, D. and Basili, V. R. (1985). “System Structure Analysis: Clustering with Data Bindings”, *IEEE Transaction on Software Engineering*, 11(8), pp.749-757.
- [Kan99] Kang, B.-K. and Beiman, J. M. (1999). “A Quantitative Framework for Software Restructuring”, *Journal of Software Maintenance: Research and Practice* 11, pp.245-284.
- [Kan98] Kang, B.-K. and Beiman, J. M. (1998). “Using Design Abstractions to Visualize, Quantify, and Restructure Software”, *The Journal of Systems and Software*, 42, pp.175-187.
- [Kim94] Kim, H. S. and Kwon, Y. R. (1994). “Restructuring Programs through Program

- Slicing”, *International Journal of Software Engineering and Knowledge Engineering*, 4(3), pp.349-368.
- [Lak99] Lakhotia, A. and Deprez, J. C. (1999). “Restructuring Functions with Low Cohesion”, *Proceedings of Working Conference on Reverse Engineering*, pp.36-46.
- [Lak98] Lakhotia, A. and Deprez, J. C. (1998). “Restructuring Programs by Tucking Statements into Functions”, *Journal of Information and Software Technology*, 40(11-12), pp.677-689.
- [Lak97] Lakhotia, A. (1997). “A Unified Framework for Expressing Software Subsystem Classification Techniques”, *Journal of Systems and Software*, 36, pp.211-231.
- [Lak93] Lakhotia, A. (1993). “Rule-based Approach to Computing Module Cohesion”, *Proceedings of the 15th International Conference on Software Engineering*, pp.35-44.
- [Lun04] Lung, C.-H., Zaman, M. and Nandi, A. (2003). “Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring”, to appear in *The Journal of Systems and Software*.
- [Lun98] Lung, C.-H. (1998). “Software Architecture Recovery and Restructuring through Clustering Techniques”, *Proceedings of the 3rd International Workshop on Software Architecture*, pp.101-104.
- [Man99] Mancoridis, S., Mitchell, B., Chen, Y. and Gansner, E. (1999). “Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Organizations of Source Code”, *Proceedings of the 6th International Workshop*

on Program Comprehension.

- [Man98] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y. and Gansner, E. R. (1998). "Using Automatic Clustering to Produce High-Level System Organizations of Source Code", *Proceedings of the 6th International Workshop on Program Comprehension*, pp.45-52.
- [Men04] Mens, T. and Tourwe, T. (2004). "A survey of software refactoring", *IEEE Transaction on Software Engineering*, 30(2), pp.126-139.
- [Mit01] Mitchell, B. S. and Mancoridis, S. (2001). "Comparing the Decompositions Produced by Software Clustering Algorithm Using Similarity Measurements", *Proceedings of International Conference of Software Maintenance*.
- [Mül95] Müller, H. A., Wong, K. and Tilley, S. R. (1995). "Understanding Software Systems Using Reverse Engineering Technology", *Object-Oriented Technology for Database and Software Systems, World Scientific*, pp.240-252.
- [Mül93] Müller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S. (1993). "A Reverse Engineering Approach to Subsystem Structure Identification", *Journal of Software Maintenance: Research and Practice*, 5(4), pp.181-204.
- [Mun03] Munson, C. J. (2003). *Software Engineering Measurement*, Auerbach Publications, ACRC Press Company.
- [Pre97] Pressman, R. S. (1997). *Software Engineering: A Practitioner's Approach*, 4th Edition, McGraw-Hill, Inc.
- [Rom90] Romesburg, H. C. (1990). *Cluster Analysis for Researchers*, Krieger Publishing Company, Malabar, Florida.
- [Ros01] Rosen, E., Viswanathan, A. and Callon, R. (2001). *Multiprotocol Label*

- [Sar01] Sartipi, K. and Kontogiannis, K. (2001). "Component Clustering Based on Maximal Association", *Proceedings of the IEEE International Conference on Software Maintenance*, pp.103-114.
- [Sch94] Schwanke, R. W. and Hanson, J. S. (1994). "Using Neural Networks to Modularize Software", *Machine Learning*, 15(2), pp.137-168.
- [Sch91] Schwanke, R. W. (1991). "An Intelligent Tool for Re-engineering Software Modularity", *Proceedings of the 13th International Conference on Software Engineering*, pp.83-92.
- [Sch89] Schwanke, R. W. and Platoff, M. A. (1989). "Cross References are Features", *Proceedings of the 2nd International Workshop on Software Configuration Management*, ACM Press.
- [Sne73] Sneath, P. H. A and Sokal, R. R. (1973). *Numerical Taxonomy: The Principles and Practice of Numerical Classification*, W. H. Freeman and Company, San Francisco.
- [Som96] Sommerville, I. (1996). *Software Engineering*, 5th Edition, Addison-Wesley, England.
- [Tse01] Tseng, L.-Y. and Yang, S.-B. (2001). "A Genetic Approach to the Automatic Clustering Problem", *Pattern Recognition*, 34, pp.415-424.
- [Tze98] Tzerpos, V. and Holt, R. C. (1998). "Software Botryology Automatic Clustering of Software Systems", *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, pp.811-818.
- [Wei84] Weiser, M. (1984). "Program Slicing", *IEEE Transactions on Software*

Engineering, 10(1), pp.352-357.

- [Wig97] Wiggerts, T. A. (1997). "Using Clustering Algorithms in Legacy Systems Modularization", *Proceedings of the 4th Working conference on Reverse Engineering*, pp.33-43.