Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 79 (2006) 1261–1279
The Journal of Systems and Software
www.elsevier.com/locate/jss

Program restructuring using clustering techniques

Chung-Horng Lung [a,*], Xia Xu [a], Marzia Zaman [b], Anand Srinivasan [c]

[a] Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, Ont., Canada K1S 5B6
[b] Cistel Technology, Ottawa, Ont., Canada
[c] EION Inc., Ottawa, Ont., Canada

Received 15 February 2006; accepted 18 February 2006
Available online 11 April 2006

Abstract

Program restructuring is a key method for improving the quality of ill-structured programs, thereby increasing the understandability and reducing the maintenance cost. It is a challenging task and a great deal of research is still ongoing. This paper presents an approach to program restructuring inside of a function based on clustering techniques with cohesion as the major concern. Clustering has been widely used to group related entities together. The approach focuses on automated support for identifying ill-structured or low-cohesive functions and providing heuristic advice in both the development and evolution phases. A new similarity measure is defined and studied intensively specifically from the function perspective. A comparative study on three different hierarchical agglomerative clustering algorithms is also conducted. The best algorithm is applied to restructuring of functions of a real industrial system. The empirical observations show that the heuristic advice provided by the approach can help software designers make better decision of why and how to restructure a program. Specific source code level software metrics are presented to demonstrate the value of the approach.

Keywords: Program restructuring; Clustering

1. Introduction

Software evolves over time primarily due to changes in requirements and technologies. As a result, huge amount of effort is spent in maintenance and evolution. Software evolution usually accounts for more than 60% of total software costs (Sommerville, 1996). In today's highly competitive era, software development is often driven by tight schedules. Hence, software designers often emphasize the functional aspect of a system. Even if a software product is well designed, the code is often modified over time in response to the changing needs of customers. As a consequence, its original structure gradually drifts and quality degrades. Hence, the program becomes difficult to understand. As a result, it is often costly to maintain.

Müller, et al. indicate that 50–90% of software evolution work focuses on program comprehension or understanding (Müller et al., 1995). Program understanding could be at various levels, including architecture, design, and code. At the implementation level, a large or poorly coded function usually involves multiple activities or has low functional cohesion, which makes the program difficult to understand and modify. Program restructuring (Chikofsky and Cross, 1990) or refactoring (Fowler, 1999) can transform these functions to functions that are better organized and easier to understand, without changing their behaviors. The new functions will usually be higher quality and less costly for further evolution. More importantly, a desirable restructuring should achieve high cohesion and low coupling (Briand et al., 1996; Munson, 2003; Pressman, 1997).

Cohesion, as an important measure in restructuring, is to measure how tightly related 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

* Corresponding author. Tel.: +1 613 5202600; fax: +1 613 5205727.
E-mail addresses: chlung@sce.carleton.ca (C.-H. Lung), xiaxu@sce.carleton.ca (X. Xu), marzia@cistel.com (M. Zaman), anand@eion.com (A. Srinivasan).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.02.037

elements in a component. Previous articles of software clustering demonstrate research potential in software clustering field (Tzerpos and Holt, 1998) and conclude that clustering methods may be a very good starting point for the remodularization of software (Wiggerts, 1997).

However, existing research on the software clustering field has mainly been concerned with software remodularization at the architecture level and has not been used in program restructuring at the source code level. Source code contains critical information regarding the behavior of a system. The understanding and manipulation of source code is a pressing issue for maintenance and evolution. This paper focuses on source code. Specifically, this paper deals with restructuring of each individual function. One challenge of restructuring at this level is how to meaningfully and effectively group related code segments together inside a large or poorly structured function to form small or cohesive functions, because it is not uncommon that unrelated fragments and functionally cohesive code segments are interleaved in real software products. In addition, the approach should be easy to understand and also effective in practice. Clustering techniques are suitable for this problem, because the objective of clustering is consistent with that of cohesion.

This paper presents an approach to program restructuring using clustering techniques at the function 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 designers identify ill-structured functions and provide them with heuristic advices. In detail, this paper discusses how to select entities and how to select attributes that are important to distinguish two different entities from the cohesion perspective. A new resemblance coefficient as a similarity measure is defined. Extensive experiments on the weights of different attributes are conducted. Three hierarchical agglomerative algorithms: single linkage algorithm (SLINK), complete linkage algorithm (CLINK) and weighted pair-group method using arithmetic averages (WPGMA[1]), are chosen and an intensive comparative study on them is conducted. These algorithms are highlighted as follows:

- 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.
- WPGMA: this defines the similarity measure between two clusters as the simple arithmetic average of resemblance coefficients between two clusters without considering the cluster size.

The algorithm that produces the best result will then be applied to program restructuring of an industrial system.

The structure of the rest of this paper is as follows. Section 2 reviews the related work in both program restructuring and software clustering areas. Section 3 proposes an approach to program restructuring using clustering techniques and discusses the issues involved in the approach. Section 4 provides an extensive study on the similarity measure by weighting attributes differently. Section 5 gives a comparative study of three clustering algorithms: SLINK, CLINK and WPGMA. Section 6 presents a case study of program restructuring using the clustering results on an industrial software system. Empirical observations are also summarized. Section 7 presents the conclusions and future work.

2. Related work

There has been extensive research on software restructuring. This section describes related research on restructuring at the function or the design level. Additionally, this section also presents related research on software clustering.

2.1. Restructuring at function level

The early days of restructuring efforts focused on making a program's control flow easier to follow. This category is quite mature (Arnold, 1989). Previous research on program restructuring at the function level has primarily used program slicing or input/output dependence techniques to restructure modules with cohesion as the main criterion (Kang and Beiman, 1998, 1999; Kim and Kwon, 1994; Lakhotia and Deprez, 1998, 1999). Conceptually, their works are similar.

Kim and Kwon (1994) present a method of restructuring a poor-structured module. 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 the code implementation, module strength is defined in terms of the level of sharing between processing blocks. However, the method does not give the information that is not related to output variables.

Kang and Beiman (1998, 1999) have introduced a method to restructure modules during the design or maintenance phases. The authors define the input/output dependence graph (IODG) of a module, similar to the variable dependence graph (VDG) in (Lakhotia, 1993), to model the 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 of program restructuring. The cohesion measure presented in these papers only considers dependence information between input and output

[1] In the previous version of this paper published in the *Proc. of SCAM 2004*, WPGMA was mistakenly termed as UPGMA.

components, and does not reflect code fragments that are not related to the output components.

Lakhotia and Deprez (1998, 1999) use tuck transformation to restructure program by breaking large functions into small functions. The method complements those reported in (Kang and Beiman, 1998, 1999; Kim and Kwon, 1994) by computing pairwise cohesion. Tuck includes three transformations: wedge, split and fold. A wedge is a subset 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 (Lakhotia, 1993) to compute pairwise cohesion between variables in the function as a criterion of restructuring. The empirical study in the paper (Lakhotia and Deprez, 1999) shows that the approach has some limitations for industrial applications.

The methods in (Kang and Beiman, 1998, 1999; Lakhotia and Deprez, 1998, 1999) 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 some code fragments, such as error handling routines, may not be related to output variables. In such cases, the slices of output variables cannot reflect the code fragment related to error handling. 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 activities. Therefore, previous approaches have some limitations.

Lung and Zaman (2004) apply clustering techniques to function restructuring and demonstrate how to restructure a low-cohesive function into high-cohesive functions using simple examples presented in the literature. They treat executable program statements as basic components, or entities, and variables as attributes. They also introduce artificial variables for iterative loops and logical control statements. This paper, however, extends the concept, defines a new similarity measure, and compares various weights systematically and applies the technique to industrial software.

2.2. Restructuring at design level

There also has been extensive research on software clustering conducted at the design or architectural level (Anquetil and Lethbridge, 2003; Anquetil et al., 1999; Arnold, 1989; Bieman and Kang, 1998; Bieman, 1994; Chikofsky and Cross, 1990; Choi and Scacchi, 1990; Chu and Patel, 1992; Hutchens and Basili, 1985; Kang and Beiman, 1998, 1999; Kim and Kwon, 1994; Lakhotia and Deprez, 1998, 1999; Lakhotia, 1993, 1997; Lung et al., 2004; Lung and Zaman, 2004; Lung, 1998; Mancoridis et al., 1999; Mancoridis et al., 1998; Maqbool and Babri, 2004; Mitchell and Mancoridis, 2001; Müller et al., 1993, 1995; Schwanke, 1991; Tzerpos and Holt, 1998; Wen and Tzerpos, 2004; Wiggerts, 1997). Tzerpos and Holt (1998) survey clustering approaches and find that classic clustering techniques can be used in the software context and that there is a research potential in the software clustering field. They point out that some structure is better than no structure. Wiggerts (1997) provides a general overview of clustering techniques and their applications to system re-modularization, highlighting the benefit of the general theory of clustering analysis. Lakhotia (1997) 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.

Previous software clustering approaches have concentrated on software system modularization or re-modularization at the architectural or design level. The entities to be clustered could be functions (routines), global variables (for identifying abstract data types), or files. Their similarity measures are either based on relationships between entities (Hutchens and Basili, 1985; Lung et al., 2004; Mancoridis et al., 1998; Mitchell and Mancoridis, 2001; Müller et al., 1993), or based on shared features (Anquetil and Lethbridge, 2003; Anquetil et al., 1999; Schwanke, 1991), with or without giving weights to the relationships or features. Researchers have used different information or formula to measure the similarity based on different perspectives.

2.3. Clustering algorithms

The clustering algorithms used in previous work fall into three categories: hierarchical algorithms (Anquetil and Lethbridge, 2003; Anquetil et al., 1999; Hutchens and Basili, 1985; Lung et al., 2004; Lung and Zaman, 2004; Schwanke, 1991), optimization algorithms (Mancoridis et al., 1999; Mancoridis et al., 1998; Mitchell and Mancoridis, 2001), and graph theoretic algorithms (Choi and Scacchi, 1990; Müller et al., 1993).

Among the hierarchical agglomerative algorithms, UPGMA (unweighted pair-group method using arithmetic averages) is a commonly used approach (Romesburg, 1990). Lung (1998) and Lung et al. (2004) present reverse engineering and reengineering experiences for architecture or design recovery based on UPGMA. Both UPGMA and WPGMA are average linkage clustering algorithms. The difference is that UPGMA considers all pair entities in two clusters or cluster size in calculating the average; whereas WPGMA calculates the simple average.

However, the survey in (Lakhotia, 1997) suggests that most researchers prefer the SLINK algorithm in subsystem classification. Girard et al. (1999) tailor the SLINK algorithm because the approach generated very large groups that were not useful. Alternatively, Anquetil and Lethbridge (2003) suggest the CLINK algorithm based on their experiments for software re-modularization using files. Maqbool and Babri (2004) present a weighted combined linkage algorithm of software clustering to support architecture recovery and a comparative study with some clustering algorithms.

Different algorithms may be suitable for different applications. Recently, Wen and Tzerpos (2004) presented a comparative study of software clustering, including hierarchical agglomerative algorithms, based on *MoJo* distance for architecture decomposition. The paper adopts the Jaccard coefficient to calculate the resemblance coefficients. Their approach also automatically generated clusters by the software based on the number of clusters selected by the user. Our paper, however, advocates user inputs to decide the final clustering results, because other factors in software could affect relationships of entities or components, especially at the code level. In addition, testing needs to be conducted after function restructuring. Function restructuring without user inputs could create a lot of burdens in post-mortem analysis and testing.

Software clustering is a complicated research area. The user of the clustering approach needs to decide how to choose entities and attributes, how to measure and compute similarity, and which algorithm to use for a specific problem. This paper targets clustering at the program statement level inside of a function, which is a relatively new area. We have conducted a number of experiments to help answer those questions.

3. An approach to program restructuring using clustering techniques

This section presents an approach to code restructuring using clustering techniques and discusses key issues of clustering techniques.

3.1. Program restructuring approach

The objective of program restructuring is to improve the structure or internal strength of a function. The program restructuring approach proposed in this paper is supported by a set of tools. The approach is based on clustering analysis, with cohesion as the main criterion. The existing structure of a program with quantitative measure is shown in a tree after a clustering analysis has been performed. The approach provides information about the existing structure of a function, the quantitative structure measure, and a heuristic guideline for improving the existing code. It can be used to help software designers make a decision – why and how to restructure an existing program.

Fig. 1 shows the approach for program restructuring using clustering techniques. The approach also deals with some fundamental and challenging issues of clustering for functions. Those issues include: definition of entities and attributes inside of a function, an algorithm to calculate resemblance coefficients, and selection of the best clustering method. Each of these issues will be discussed in detail later in this paper. Currently, the study is conducted for C programs; however, the technique can be applied to other languages as well.

The approach has four key phases as shown in Fig. 1. Phase one is data collection and processing. In this phase, the Parser tool parses source code automatically and generates raw data of entity–attribute matrix. The raw data may contain some "noises" (unwanted data), which are removed during data refining. Entities are the items or components that are going to be clustered. Each entity has one or more attributes. Entities are grouped based on the attributes that they share. In other words, the more attributes two entities have in common, the more closely related these two entities are. However, in order to apply the clustering technique to programs, we first need to define *entities* and *attributes* specifically relevant to functions in this phase, which will be presented in detail in Sections 3.2 and 3.3. The entity–attribute matrix generated 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. After entities and their attributes are defined, a metric called resemblance coefficient is calculated to measure the similarity between two entities. Basically, similarity or the resemblance coefficient between two entities is measured or calculated based on the common attributes that these two entities share. Many algorithms have been proposed or used to compute the coefficients for various applications

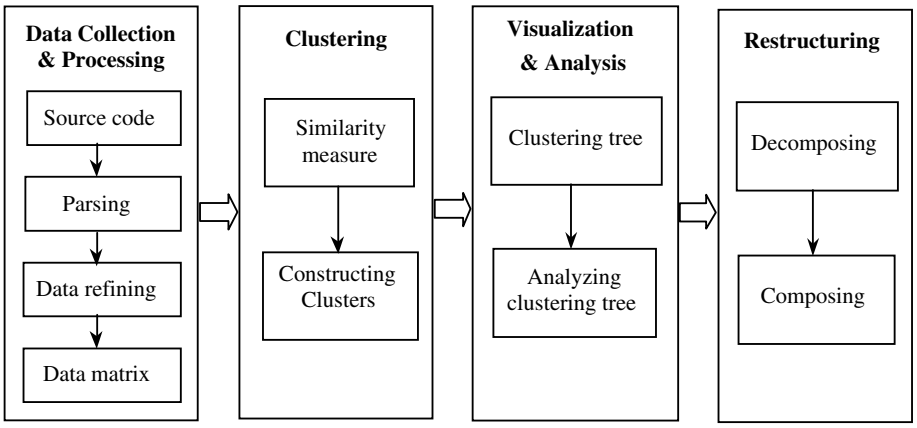

Fig. 1. An approach to program restructuring.

(Everitt, 1974; Romesburg, 1990; Sneath and Sokal, 1973). Software, however, potentially consists of many artificial factors for which those algorithms are not directly suitable. Section 3.4 discusses in length how to calculate resemblance coefficients specifically for program restructuring.

After the resemblance coefficient has been defined, clusters can be constructed using a clustering algorithm. Currently, three hierarchical agglomerative algorithms: SLINK, CLINK and WPGMA, are supported. Selecting the best clustering algorithm is another challenge. Different algorithms may be better suited for different applications as summarized in Section 2.3. We have conducted a number of comparative studies in different areas and selected the best one for program restructuring. Section 5 presents the comparison among the three clustering algorithms. The Clustering tool performs this phase automatically.

Phase three is visualization and analysis. After the clustering phase, the result is displayed as a tree, which shows the existing structure of a function being analyzed. Closely related entities are grouped into a cluster. The degree of relatedness of two entities in a cluster is represented by the resemblance coefficient. By examining the tree, ill-structured code fragments may be identified; these are candidates for restructuring. The clustering tree provides heuristic advice on how to restructure a function. But software designers must participate in making the final decision based on their experience, insights, and the restructuring objectives, though the tool itself can automatically generate clusters if the number of clusters is provided.

Phase four is the actual restructuring of a program. The identified low-cohesive functions will be decomposed into several code fragments, and some of them are composed into new functions. This phase is processed manually.

3.2. Entities

Entities are those items that needs to be grouped. For program restructuring at the function level, statements are chosen as entities, because statements are the basic units of a program and it is not our intention to change the behavior of a program. There are two types of statements: executable and non-executable statements. Non-executable statements, such as comments and declarations, have no real effect on 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 statement and so on. In the restructuring approach, only the executable statements that can be described by their attributes are considered to be entities (attributes are discussed in Section 3.3). Entities are further divided into control entities and non-control entities. A control entity refers to an entity that is either a predicate statement (such as *if* or *switch* statement) or iteration statement (such as *for* or *while* statement). If an entity is not a control entity, it is a non-control entity. Each entity is represented by a number, which corresponds to the line number of a statement in the source code.

3.3. Attributes

An attribute is a feature or property 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 predefined objective criterion. Attributes will be used to calculate how closely two entities are related based on the fact that entities are more similar if they share many common attributes.

In order to perform clustering on program statements (entities), we first need to identify the attributes of the entity. A statement consists of variables, constants, operators, keywords, 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, operators and keywords are not chosen as attributes.

Based on data dependence and control dependence relationship, variables are divided into data variables and control variables, which are described below.

3.3.1. Data variable

A data variable refers to the variable that is directly used in a statement. Data variables as a type of attribute reveal the data dependence relationship of entities. Data variables include local variables, global variables, and parameters passed to a function. They can also 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. In addition, a function name in a function call statement is also treated as a data variable.

3.3.2. Loop counter variable

A loop counter variable is another 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 associated with a composite variable, e.g., an index variable used in an array. Therefore, the loop counter variable is not counted as an attribute.

3.3.3. Control variable

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 describe entities 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, e.g., *if* or *while* block, in the source code.

Therefore, in the restructuring approach, data variables (excluding loop counter variables) and control variables are chosen as attributes to describe entities. They are also called data attributes and control attributes, respectively. An attribute can be measured with a quantitative scale or a qualitative scale. Based on our definition of attributes, each attribute is measured on a qualitative scale as a binary representation. Thus, each attribute has two states either presence or absence, which are described below.

0 – absence state of a control or data attribute;
1 – presence state of a control attribute;
2 – presence state of a data attribute.

In addition, the data attributes in control entities are treated as control attributes due to the fact that the entities are used for the control purpose. Based on the above discussion, between any two entities, there are six different types of combinations or matches for each attribute, as follows:

- 1–1 match: a control attribute is present in both entities.
- 2–2 match: a data attribute is present in both entities in case neither of them is a control entity.
- 0–0 match: an 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.

3.4. Similarity measure

Similarity measure is used to evaluate closeness between two entities and is represented with a resemblance coefficient. Many algorithms have been proposed depending on the nature of the data and selection of weights. The main idea is based on two features: Attributes and Matches.

3.4.1. Attributes

Generally, the more attributes two entities share, the closer they are related and the more similar they are. There are two types of attributes: data attributes and control attributes. From the cohesion point of view, these two types of attributes contribute to different degrees of cohesion because they describe different dependence relationships. Lakhotia (1993) indicates that two variables, which have a data dependence relationship, are more high-cohesive than two variables that have a control dependence relationship. Therefore, data attributes and control attributes should be weighted differently.

Data attributes have different types, namely a local variable, a global variable, a parameter passed to a function or a function name in a function call statement. It is important to understand if we need to treat these different types of data attributes differently. In addition, a data attribute may appear in a non-control entity or a control entity. Hence, it is also important to understand if a data attribute is measured equally when it describes different types of entities. We analyze different types of data attributes as follows:

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 variable and a local variable. In a low-cohesive function, a global variable may be referenced by several different activities. But a local variable may also have the same situation. Hence, a global variable and a local variable in a function play the same important role in function cohesion and are therefore, treated equally.

Function name: A function name in a function call statement is treated as a data variable. In the 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 is different from a non-control entity in that it has an indirect contribution to a functional activity. When a variable or a data attribute appears in a control entity, it has no direct relatedness to an activity. However, when a data attribute 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 the approach, data attributes in control entities are simply treated as control attributes.

Therefore, all the data attributes (e.g., variables) in the data entities (non-control statements) are considered to have equal importance for functional cohesion. As the data attributes in control entities are treated as control attributes (e.g., if statements), the problem of the weighting attributes boils down to the problem of the weighting between control attributes and data attributes. We believe that data attributes should be weighted more than control attributes, since a data attribute affects a functional activity directly while a control attribute affects indirectly.

3.4.2. Matches

Now, the problem is to determine the weights for various matches in the similarity measure.

0–0 match: A 0–0 match means that an attribute is not used in either of the two entities. An entity–attribute matrix gives all the attributes that are used in a function. However, each entity is only related to a few attributes, and most of them are valued with 0. There are many 0–0 matches in the matrix. Lung et al. (2004) address that counting 0–0

matches will generate distortion and result in dissimilarity. The study presented in (Anquetil and Lethbridge, 2003) 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. 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 attribute and there is already a 1–1 match that counts the control dependence. Thus, there is no need to use 1–2/2–1 matches to describe control dependence again. When these two entities are not in the same control block, they do not have control dependence. So 1–2/2–1 matches are also ignored.

1–1 match and 2–2 match: 1–1 matches and 2–2 matches mean that two entities share common attributes, which have a positive contribution to the similarity measure. 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. Because data dependence contributes more to cohesion than control dependence, a 2–2 match should have more weight than 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 for 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 the similarity between entities than matches on common control attributes (1–1 matches), then mismatches on data attributes (2–0/0–2 matches), 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; while 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

Based on discussion mentioned above, a new resemblance coefficient between two entities is defined as follows:

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

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 2–0/0–2 matches between two entities;
b_c – number of 1–0/0–1 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 and the weight of data attributes is heavier than that of 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 them, $b_d = 0$ and $b_c = 0$, then they achieve the maximum similarity with $coeff = 1$. The value of the resemblance coefficient is between 0 and 1.

4. Experiments on similarity measure

The resemblance coefficient has been defined, but how to decide the weights is still unsolved. Previous research did not give systematic study on this issue. Dhama (1995) uses a heuristic estimate to give the data parameters twice as much weight as the control parameters. Schwanke (1991) estimates the significance of a feature using Shannon information content, which gives rarely-used identifiers higher weights than frequently-used identifiers. In this paper, the weights of attributes are considered as positive integer and are decided through extensive experiments.

The experiments focus on the different weight ratios between the data attributes and the control attributes. A number of functions appearing in papers (Bieman and Kang, 1998; Bieman, 1994; Kim and Kwon, 1994; Lakhotia and Deprez, 1999), student assignments, and real industrial programs are used for evaluation. The size of each function ranges from 8 to 55 lines (not including comments and white spaces). The experiments start with a weight ratio of 2:1, that is, in Eq. (1), $w_d = 2$ and $w_c = 1$. The results shown in this section are generated by the WPGMA algorithm.

4.1. Weight ratio of 2:1

The weight ratio of 2:1 works well for most of selected examples. But it does not work well when it is used to analyze an example with communication cohesion in (Bieman and Kang, 1998). The example code is shown in Fig. 2, the entity–attribute input matrix (without function name and parameters, and variable declarations) is shown in Table 1, and Fig. 3 illustrates the clustering result depicted by a clustering tree or 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 a function. 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.

```
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 := 0;
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;
```

Fig. 2. Sample code 1: sum and prod (Bieman and Kang, 1998).

Table 1
Entity–attribute matrix of sample code 1

Entity	Attribute					
	Data attribute					Control attribute
	n	*arr*	*sum*	*prod*	*avg*	*for*
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

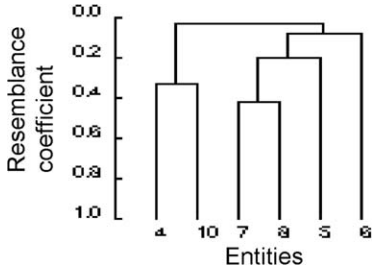

Fig. 3. Clustering tree with 2:1 weight ratio for sample code 1 in Fig. 2.

Fig. 3 shows that entities (7,8) are grouped together, and entities (4,10) are grouped together. But in fact, entities (4,7) are related to the same functional activity – the computation of *sum*. Entity 10 uses the result of *sum* to compute the average *avg*. Entities (5,8) contribute to the same activity – the computation of *prod*. The tree does not reveal the real functional structure in this example.

The resemblance coefficients between those entities give the explanation of the result.

$$coeff_{(4,7)} = \frac{2 \times 1}{(2 \times 1) + (2 \times 1) + (1 \times 1)} = 0.40.$$

$$coeff_{(5,8)} = \frac{2 \times 1}{(2 \times 1) + (2 \times 1) + (1 \times 1)} = 0.40.$$

$$coeff_{(7,8)} = \frac{(2 \times 1) + (1 \times 1)}{(2 \times 1) + (1 \times 1) + (2 \times 2)} = 0.43.$$

Because $coeff_{(7,8)} > coeff_{(4,7)}$ and $coeff_{(7,8)} > coeff_{(5,8)}$, the algorithm groups entities (7,8) together instead of entities (4,7) and entities (5,8). Although data attributes are weighted twice as heavily as control attributes, it seems that the control attributes still play more of a role in similarity measure than they should, and more weight should be added to data attributes.

4.2. Weight ratio of 3:1

The weight ratio of 3:1 is used in the sample code 1 in Fig. 2 and the result is shown in Fig. 4. The clustering tree illustrates 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 *prod*. Entity 6 is a control entity that is shared by two computation activities. The tree shows the real functional structure.

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

$$coeff_{(4,7)} = \frac{3 \times 1}{(3 \times 1) + (3 \times 1) + (1 \times 1)} = 0.43.$$

$$coeff_{(5,8)} = \frac{3 \times 1}{(3 \times 1) + (3 \times 1) + (1 \times 1)} = 0.43.$$

$$coeff_{(7,8)} = \frac{(3 \times 1) + (1 \times 1)}{(3 \times 1) + (1 \times 1) + (3 \times 2)} = 0.30.$$

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

Sample code 2 in Fig. 5 is an example from an industrial program. It is the 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 27 to line 57.

Fig. 6 shows the clustering 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 27 to 54, as mentioned above. Here 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 to 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 similarity measure than they should. In the experiments, differ-

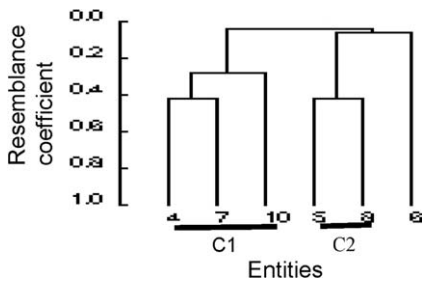

Fig. 4. Clustering tree with 3:1 weight ratio for sample code 1 in Fig. 2.

```
1 process_functionBody (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      ();
     //...
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) {
27       if (! search_decl_keywords (token) &&
              ! search_decl_user (token)){
28          //...
30          position = search_local_list (token);
31          if (position != -1) {
32             update_local_list (position, *cntl_flag, line_no);
               //...
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);
                   //...
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 }
```

Fig. 5. Sample code 2: Process functionBody.

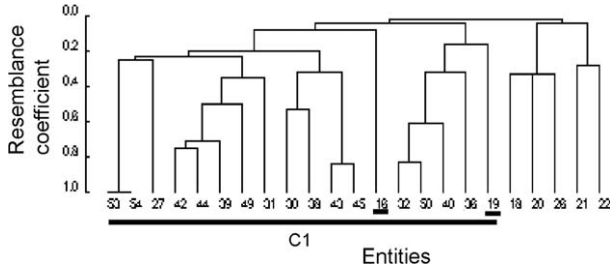

Fig. 6. Clustering tree with 3:1 weight ratio for sample code 2 in Fig. 5.

ent weight ratios, between 2:1 and 3:1, have been tested. Those ratios are 9:4, 7:3, 5:2, and 8:3.

4.3. Weight ratios of 9:4 and 7:3

When a weight ratio of 9:4 or 7:3 is used to the sample code 1 in Fig. 2, both of them generate a similar clustering tree as the one with a weight ratio of 2:1 shown in Fig. 3. So both 9:4 and 7:3 weight ratios do not work well for the sample code 1.

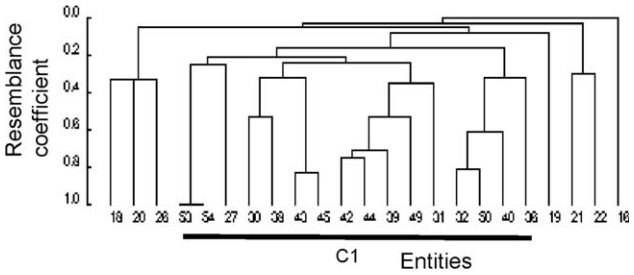

Fig. 7. Clustering tree with 8:3 weight ratio for sample code 2 in Fig. 5.

4.4. Weight ratios of 5:2 and 8:3

When a weight ratio of 5:2 or 8:3 is used to the sample code 1 in Fig. 2, both of them generate a similar clustering tree as the one with a weight ratio of 3:1 in Fig. 4. So both 5:2 and 8:3 weight ratios work well for the sample code 1. When these two ratios are used to the sample code 2 in Fig. 5, they generate very close results. Fig. 7 shows the clustering tree with an 8:3 weight ratio for the sample code 2.

Fig. 7 shows that cluster C1 contains exact entities that are related to the activity of processing the token body with an unreserved token type. Entities 16 and 19, which are inside cluster C1 in Fig. 6 with a weight ratio of 3:1, are now outside the cluster here. This is because the weight of 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. The same reason is for entity 19. The tree reveals the real functional structure of the sample code 2. Both 8:3 and 5:2 ratios work well in this example. These two weight ratios also give expected results for all selected examples in the experiments.

In summary, six different weight ratios were investigated in a series of experiments. The weight ratios of 2:1, 9:4, and 7:3 do not work with sample code 1 in Fig. 2. The weight ratio of 3:1 works well with the sample code 1, but does not work well with sample code 2 in Fig. 5. Both 5:2 and 8:3 ratios work very well with all selected examples and generate very close results. Therefore, the ratio of 8:3 is chosen to weigh the data and control attributes in the similarity measure.

5. Experimental comparison of WPGMA, SLINK, and CLINK

The WPGMA, SLINK, and CLINK clustering algorithms have been applied to more than 60 functions in different areas, including functions appeared in papers, student assignments and industrial programs. This section presents comparisons of these three algorithms.

```
1  procedure Sum_Max_Ave(n: integer; var arr: int_array;
                         var sum, max: integer;  var avg: float);

2  var I: integer;
3  begin
4     sum := 0;
5     max := arr[1];
6     for I := 1 to n do begin
7        sum := sum + arr[I];
8        if arr[I] > max
9           max := arr[I];
10 end;
11 avg := sum/n;
12 end;
```

Fig. 8. Sample code 3: Sum, Max and Average (Bieman and Kang, 1998).

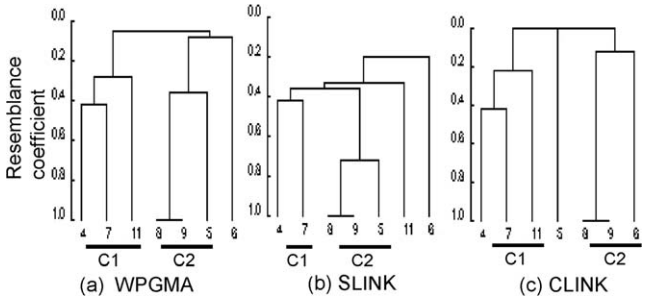

Fig. 9. Clustering results for sample code 3.

5.1. Comparison on small functions

Fig. 8 is an example code in (Bieman and Kang, 1998), which calculates the sum and average, and selects the maximum element of an array. The WPGMA, SLINK, and CLINK clustering algorithms are applied to sample code 3 of Fig. 8. Fig. 9 presents the results. Fig. 9(a) shows two clusters, (4,7,11) and (8,9,5). Statement 6, for loop, does not belong to any cluster; rather, it is shared by the two clusters. Fig. 9(b) shows two clusters, (4,7) and (8,9,5). Statements 11 and 6 join two previous groups later, one by one. This is called a "chain" phenomenon. Fig. 9(c) generates the worst result. Statement 5 does not show any relationship with other statements. Its resemblance coefficient is 0, which clearly does not reflect the real meaning of the function.

We have applied all three algorithms to 21 small functions (7 LOCs to 22 LOCs) presented in the related literature (Bieman and Kang, 1998; Bieman, 1994; Lakhotia and Deprez, 1998, 1999). For all examples provided in the papers, the WPGMA has an effective rate of 100% based on designers' evaluation. Alternatively, SLINK does not work well for three examples and CLINK does not work well for 5 examples used in (Bieman and Kang, 1998; Bieman, 1994).

5.2. Comparison on student assignments

The three algorithms have also been applied to fourteen functions from students' assignments that were designed for a variation of the classical smoker problem using semaphores and shared memory on the Linux operating systems. WPGMA works well for all but one function. SLINK generates good clusters for seven functions and CLINK has good results only for two examples. Table 2 demonstrates the results.

In a few assignments, all three clustering algorithms detected duplicate code segments. The duplicate code is detected by the replicated patterns in the dendrogram. The repeated portions can be replaced with an array, a function, or a loop depending on the nature of the code. Replacing repeated code segments or clones will improve the maintainability of the function.

5.3. Comparison on industrial programs

The next step was a comparison of these three clustering methods on industrial programs. First, these three algorithms were applied to analyze two functions from an industrial C parser used for metrics extraction. This was the preliminary step to evaluate clustering methods for practical programs. Table 3 summarizes the results.

Table 2
Comparison between three algorithms for students' programs

Function	WPGMA	SLINK	CLINK	LOCs	Cyclomatic complexity	No. of activities
1	x			182	36	3
2	x			85	19	3
3	x			30	6	2
4	x	x		52	14	1
5	x	x		43	8	1
6	x	x	x	16	2	1
7	x	x	x	55	14	3
8	x			68	13	3
9	x			120	20	3
10	x	x		68	14	3
11		x		87	14	3
12	x	x		71	13	3
13	x			154	30	3
14	x			112	27	3

Note: x indicates the best algorithm.

Table 3
Comparison between three algorithms for C parser software

Function	WPGMA	SLINK	CLINK	LOCs	Cyclomatic complexity	No. of activities
process_body	x	x		41	13	1
process_open_braces	x			57	19	3

Note: x indicates the best algorithm.

Table 4
Comparison between three algorithms for RSVP-TE signaling protocol software

Function	WPGMA	SLINK	CLINK	LOCs	Cyclomatic complexity	No. of activities
rsvpTeRx	x			74	19	3
rsvpTeDecodeMsg	x			283	64	2
rsvpTeRxPath	x			104	22	2
rsvpTeProcessERO	x			118	23	2
rsvpTeReserve	x			100	32	3
rsvpTeUpdateRSB	x			58	12	2
rsvpTeBuildResvMsg	x			164	24	1
rsvpTeRxPTear	x			70	13	2
rsvpTePSBTimeout	x			44	8	3
rsvpTeRxRTear	x			191	38	2
rsvpTeRxRErr	x			140	28	4
rsvpSBFind	x		x	46	17	2
rsvpTeUpdatePSB	x	x		76	13	1
rsvpTeBuildRSB	x	x		32	8	1
rsvpTeProcessFlowDescriptor	x	x	x	98	18	1
rsvpTeUpdateERO		x		42	9	2
rsvpTeProcessRRO		x		51	16	1
rsvpTeUpdateRRO		x		42	9	2
rsvpTeBuildPSB		x		110	13	1
rsvpTeBuildPathMsg		x		91	10	1
rsvpTeRxResv		x		56	12	2
rsvpTeResvRefresh		x		57	12	2
rsvpTeRSBAddFilter		x		50	7	1
rsvpTeResvTearFD		x		70	17	2
rsvpTeRxPErr		x		52	8	2

Note: x indicates the best algorithm.

WPGMA shows better results for both cases while SLINK generates functional cluster only for one case.

Second, the algorithms were applied to 25 functions (ranging from 32 LOCs to 283 LOCs) from the RSVP-TE network signaling protocol software. The background information of the software system is described in Section 6 on program restructuring. These 25 functions were selected by the designers as restructuring candidates. Table 4 summarizes the comparison results. Furthermore, the following observations are made:

- WPGMA generates good functional clusters for 15 functions whereas SLINK generates 13 and CLINK only generates 2.
- Only WPGMA generates good clusters for functions larger than 110 LOCs or with a cyclomatic complexity higher than 18.
- WPGMA generates functional clusters also when the size or complexity was small.
- WPGMA detects duplicated code segment better.

Fig. 10 illustrates the clustering results for the three algorithms for the *rsvpTeRxResv* function. The clustering tree from SLINK shows that there are two big functional clusters: one functional cluster is related to the RSVP-TE message sanity check and the other is related to message processing. The clustering tree from CLINK shows 13 small clusters and the clustering tree from WPGMA is between SLINK and CLINK. Different clustering algorithms generate different clusters. SLINK tends to build a small number of loose clusters or generates a ''chain'' phenomenon as illustrated in Fig. 11. On the other hand, CLINK tends to form a large number of small but compact clusters. WPGMA provides a compromise between those two.

6. Case study of program restructuring

So far, we have defined entities and attributes that are used in the similarity measure; devised a new algorithm to calculate the resemblance coefficient; and compared

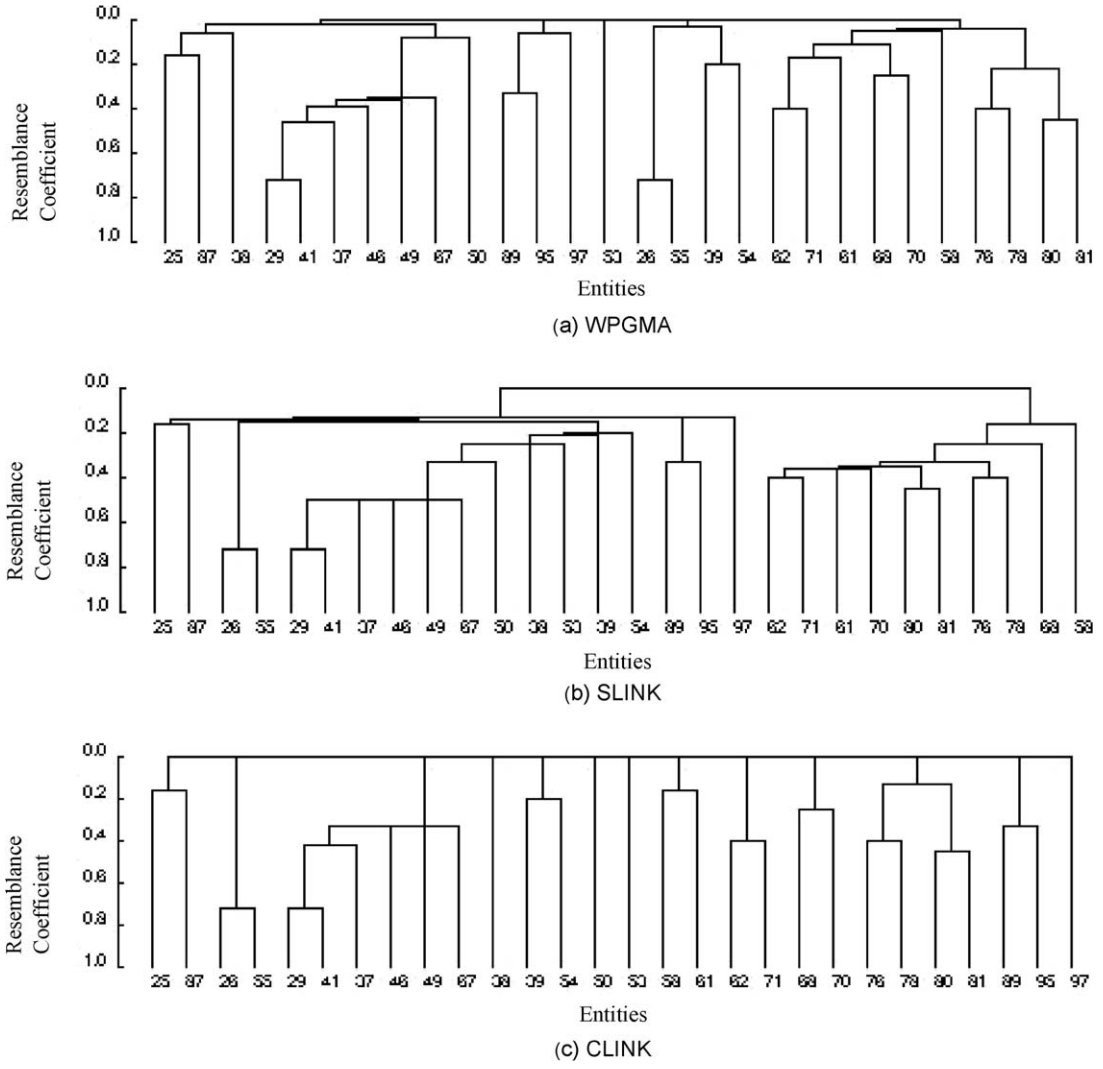

Fig. 10. Comparison of three algorithms for *rsvpTeRxResv*.

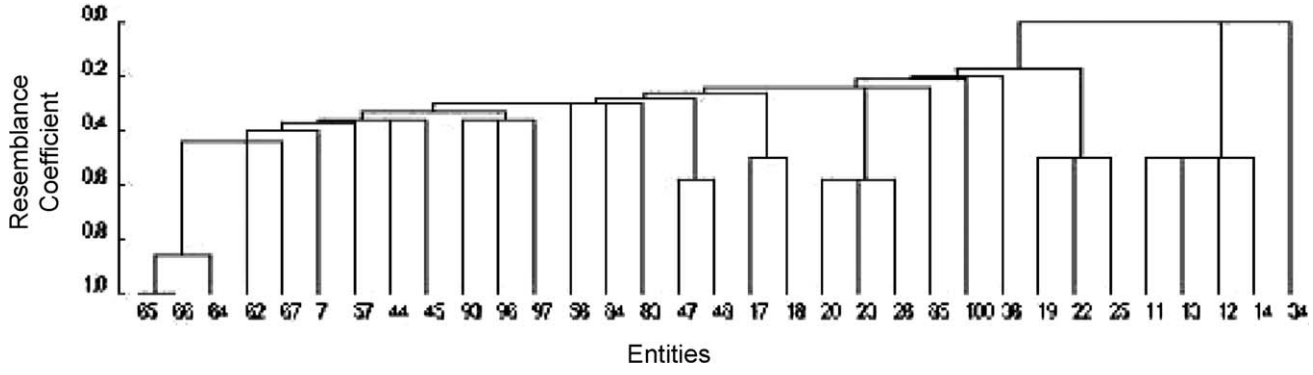

Fig. 11. Chain phenomenon generated by SLINK.

three clustering algorithms. In order to evaluate the effectiveness of the proposed approach, we have applied the approach to restructuring of a real industrial system in data networks.

6.1. System in the case study

The system in the case study is a real network protocol RSVP-TE program in the telecommunication industry.

RSVP (Braden et al., 1997) is a resource reservation protocol that enables Internet applications to obtain different qualities of service (QoS). RSVP-TE (Awduche et al., 2001) is a signaling protocol that extends the RSVP to support multiple protocol label-switching (MPLS) (Rosen et al., 2001) traffic-engineering applications. RSVP-TE provides a mechanism to establish and maintain explicitly routed label switched paths (LSPs) (Rosen et al., 2001), with or without resource reservation.

The original RSVP-TE program was completed on a schedule-driven basis. During the initial design, only basic functionalities for simple cases were coded, 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 6500 lines of code (LOCs) (in this paper, the LOC metric does not include comments and white spaces) and 110 functions. Some functions are large and involve multiple functional activities. As a result, the understandability of the code was low. Maintaining and extending the code for additional functionality were less than desirable.

Based on the evaluation conducted by the designer, 25 functions, as shown in Table 4, with a total of 2173 LOCs have been selected as restructuring candidates in order to improve the software quality. The function size ranges from 32 LOCs to 283 LOCs.

6.2. Restructuring process

For each of the selected function, the restructuring process follows the steps shown in figure and is described as follows:

- 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.
- Perform clustering analysis: the Cluster tool computes the resemblance coefficients between entities and forms a clustering tree based on a specified clustering algorithm. All three clustering algorithms, WPGMA, SLINK and CLINK, are used and compared in the case study.
- Determine restructuring candidates: After the clustering analysis, we identified that 17 out of the 25 functions in Table 4 involve more than one functional activity. Some of them also have duplicated 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 final decision on restructuring was made based on the combination of the heuristic advice, the restructuring objective and the software engineers' experience, skills and understanding of the program.
- Restructure poorly designed function: the code of the selected function is decomposed into different fragments, based on the restructuring decision made in the previous step. The code that is related to a specific functional activity is composed into a new cohesive function.

The restructuring results are illustrated in Section 6.3.

6.3. Restructuring results

In order to compare restructuring results, some software metrics are adopted for evaluation (Fenton and Pfleeger, 1997), including two commonly used metrics (LOC and cyclomatic complexity) and a cohesion measure. The Krakatau metrics tool is used to calculate metrics of size and cyclomatic complexity. Although the concept of cohesion is generally accepted, there is yet a standard cohesion metric in the literature. It is beyond the scope of this paper to discuss the cohesion metric in detail. This paper, on the other hand, adopts the cohesion measure suggested by Anquetil and Lethbridge (2003) as a way to compare the original code and the restructured program and to present the improvement of cohesion. Their definition of cohesion of a function is the average resemblance coefficient between any two entities in the function.

Table 5 presents a summary of the comparison before and after restructuring. In the case study, the 17 not well-designed functions with a total of 1611 LOCs were restructured, which represent 24.78% of the RSVP-TE program. After restructuring, 34 new functions were generated. Compared with the original 17 functions, the average size of a function, after restructuring, drops by 60.65% from 94.76 LOCs to 37.29 LOCs, the average cyclomatic complexity decreases by 61.45% from 19.94 to 7.69, and the average cohesion increases by 100%. The restructuring shows a measurable improvement. The complexity improvement is especially significant, which shows that the program understandability and maintainability are greatly improved.

The Appendix presents an example showing how one function was actually restructured. The original function has about 140 executable LOCs with cyclomatic complexity of 28. The function was decomposed into five functions ranging from 28 LOCs to 50 LOCs, and complexity values are from 4 to 11. The original code contained four activities. Each activity was then extracted to become a new function.

From the designer's perspective, the restructured software becomes smaller in size and simpler in complexity. Each function performs less or simply one activity. Therefore, the code becomes easier to understand and maintain.

Table 5
Comparison 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%

The restructured code was also accepted by the design team.

6.4. Empirical observations

In the case study, the restructuring approach is applied to 24 functions in the RSVP-TE software. In general the approach works well and provides heuristics for restructuring. The following presents some observations and limitations.

Functional clusters: Related entities are grouped together to form a cluster. If a cluster corresponds to a specific functional activity, it is a functional cluster. A clustering tree shows functional clusters and gives heuristic advice to designers to consider restructuring.

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. This happened in the case study.

Interleaved code: Normally, if there is no interleaved code, a cluster corresponds to a contiguous fragment of code, e.g., 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.

Cut-point: In some cases, there is no single cut-point that can be used to divide the whole clustering 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 moved to a new function for cohesion perspective. The decision should be made by designers to avoid maintenance problems, even though the clustering tool can achieve it using some threshold values.

Clustering algorithms: In the case study, the restructuring approach has been experimented on all 25 functions with three clustering algorithms: WPGMA, SLINK and CLINK. Table 2 reveals that WPGMA and SLINK generate similar best results for six out of fourteen functions. The result obtained from the network protocol software, however, shows that only one function for which all three algorithms generate the expected result and only three functions for which both WPGMA and SLINK work very well. In total, WPGMA works well for 14 functions and SLINK works well for 13 functions. CLINK does not work well in the case study as well as for the students' assignments.

Restructuring may not necessarily always reduce the size and/or complexity: In some cases, the size or complexity of a function may not be reduced much after restructuring. For example, a function called *rsvpTeDecodeMs* has similar metric values before and after the restructuring. This particular function has a *switch* statement that consists of a large number of cases. The big *switch* statement is used to decode different possible RSVP-TE objects in an RSVP-TE message. It is logical to keep those cases in one place although they may not be related to each other.

Although the clustering analysis in the restructuring approach can show functional clusters and reveal some potential problems that exist in the source code, there are still some limitations, which are describe as follows:

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. But it may also appear independently. We suggest that the software designers be responsible to identify whether a cluster is a functional cluster or a non-functional cluster primarily due to possibly complicated program semantics and other factors, e.g., performance.

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

Big data structures: In the RSVP-TE program, there is a global variable *rsvpNode*, which is a big data structure (struct) with 52 member variables. In the restructuring approach, such variable is treated as one variable. Therefore, different functional activities that are related to different member variables could be grouped together.

One variable related to multiple functionalities: In some functions, one variable may be used in entities that participate in different activities. These entities tend to be grouped together.

7. Conclusions and future directions

This paper presented a program restructuring approach using the clustering technique, for C programs. Specifically, we have discussed the selection of entities and attributes, similarity measure, resemblance coefficient experiments, hierarchical agglomerative algorithms comparison, and the application of the approach to an industrial program. The main goal of the restructuring approach was to provide automated support to identify poorly designed or low-cohesive functions and give heuristic restructuring advice to software designers in order to improve the cohesion of functions in both the software development and evolution phases.

In the restructuring approach, entities are divided into control entities and non-control entities. Similarly, attributes are divided into data attributes and control attributes. A new resemblance coefficient is defined to measure the similarity between entities in terms of cohesion. The experimental study of various weight ratios between the data attribute and the control attribute shows that a weight ratio of 8:3 (or 5:2) consistently generates the expected results for all the selected examples under study.

Based on the study of more than 60 functions in different areas, WPGMA generally works best among the three commonly used algorithms, especially when the software size or complexity is high. It improves the quality of the code and supports evolution, resulting in software that is more understandable and maintainable. SLINK works well

The original code (without variable declarations and comments) for rsvpTeRxRErr before restructuring:

```
11 void rsvpTeRxRErr(rsvp_pt rpdu,
           int cid)
17 {
    // variable declarations
38
39 RSTRACE_RSB(("rsvp_te: rx reservation
      error\n"));
40
42 phop = rsvpNode->n_rx_hop;
43 if(!phop) return;
44
46 espec = rsvpNode->n_rx_espec;
47 if(!espec) return;
48 flags = espec->u.ev4.flags;
49 ecode = espec->u.ev4.ecode;
50 eval = n2u16(espec->u.ev4.eval);
51
53 c = rsvpCircFromId(cid);
54 if(!c)
55    return;
56
```

The code between line 58 and line 80 corresponds to cluster C4 shown in Figure 12

```
58 len = n2u16(rpdu->r_mlen);
59 robj = (RSVP_OBJ *)((byte *)rpdu +
      RSVP_HLEN);
60 len -= RSVP_HLEN;
61
63 num_filt = 0;
64 while(len > 0 )
65 {
66    switch(robj->o_class)
67    {
68     case RSVP_CL_FLOWSPEC:
69        flow = (ROBJ_FLOW *)robj;
70     break;
71     case RSVP_CL_FILTER:
72        filtss[num_filt]=(ROBJ_FI
             *)robj;
73        num_filt++;
74     break;
75     default:
76     break;
77     }
78     len -= RSVP_OLEN(robj);
79     robj = RSVP_OFWD(robj);
80 }
81
82 if(!flow || !num_filt)
83 {
84     rsvpTeGenResvErr
        (ecode,eval,flags,0,0,0);
85    return;
86 }
87
89 sess = rsvpNode->n_rx_sess;
90
92 stobj = rsvpNode->n_rx_style;
93 if(!stobj) return;
94 style = stobj->style;
95
97 if((sb = rsvpTeSBFind(sess)) == 0)
98    return;
99
101 dst4 = XIPA(sess->u.te_sv4.dst);
102 if(!rsvpTeAddrInCirc(c, dst4, 32))
103 {
105    if((ecode == REC_NOTIFY) && (eval
          == REV_RRO_LARGE))
106    {
```

The code between line 107 and line 123 corresponds to cluster C3 shown in Figure 12

```
107        for(i=0; i < num_filt; i++)
108        {
109           psb = rsvpTePSBFind
                    (sess,filtss[i]);
110           if(!psb) return;
111
113           src4 = XIPA(psb-
                    hop.u.hv4.addr);
114           obd = rsvpTeBuildPErrMsg
                  (psb, ecode, eval, cid);
115           if(obd)
116           {
117              rhdr = (rsvp_pt)obd-
                        ibd_buf;
118              len = n2u16(rhdr-
                        r_mlen);
119              rsvpTeTx(obd, len, src4,
                      IP_PROT_RSVP, 1, 0);
120              return;
121           }
122        }
123     }
124     return;
125 }
126
```

Fig. 12. The original source code of *rsvpTeRxRErr* without variable declarations and comments.

The code between line 128 and line 162 corresponds to cluster C2 shown in Figure 12

```
128 if(ecode == REC_ADMIT)
129 {
134 if(style == STYLE_FF)
135 {
136    bsb = rsvpTeBSBFindByFilter
                  (sb,filtss[0]);
137    if(!bsb)
138       rsvpTeBSBCreate
             (sb,phop,flow,filtss[0]);
139    else
140    {
141       p2TimerSet(bsb->timer, rsvpNode-
          n_refresh * rsvpNode-n_lifemult,
          (vfcnptr)rsvpTeBSBTimeout,
          &rsvpNode-n_timers);
142       rsvpCopyObj((RSVP_OBJ *)&bsb-
          flow, (RSVP_OBJ *)flow);
143    }
144 }
145 else
146 {
147    for(i = 0; i < num_filt;i++)
148    {
149       bsb = rsvpTeBSBFindByFilter
                (sb,filtss[i]);
150       if(!bsb)
151          rsvpTeBSBCreate
                (sb,phop,flow,filtss[i]);
152       else
153       {
154          p2TimerSet(bsb-
             timer,rsvpNode->n_refresh *
             rsvpNode-n_lifemult,
             (vfcnptr)rsvpTeBSBTimeout,
            &rsvpNode->n_timers);
155          rsvpCopyObj((RSVP_OBJ *)&bsb-
             flow,(RSVP_OBJ *)flow);
156       }
157    }
158 }
159
161 rsvpTeResvRefreshPhop(sb,phop,1);
162 }
```

The code between line 168 and line 192 corresponds to cluster C1 shown in Figure 12

```
166 for(rsb=sb->rsbfwd;rsb;rsb=rsb-fwd)
167 {
168    if(rsb->out_if == cid)
169       continue;
170    match = 0;
171    for(i = 0;i< num_filt;i++)
172    {
173       if(rsvpTeRSBFindFilter
             (rsb,filtss[i]))
174       {
175          match = 1;
176          break;
177       }
178    }
179    if(!match) continue;
180
182    if(style != rsb->style) return;
183
184    if(ecode == REC_ADMIT &&
         (flags & E_INPLACE))
185    {
190       if(rsvpCmpFlows(flow,&rsb-
             flow) == FLOW_GTR)
191       continue;
192    }
193
195    rsvpNode->n_rx_hop = &rsb->nhop;
196
197    rsvpTeGenResvErr(ecode,
          eval,flags,0,0,0);
198    }
199 }
```

Fig. 12 (*continued*)

in some cases, but the results are more difficult to predict. On the other hand, CLINK does not produce good functional clusters. To reveal more cohesive clusters, WPGMA and SLINK can be applied at the same time.

As a case study, the approach was used to analyze a real telecommunication program and subsequent restructuring. In general, the approach works well. The clustering analysis based on the resemblance coefficient defined in this paper can identify high-cohesive sub-functions inside a large low-cohesive function and reveal potential problems in the existing code.

In real programs, there are many artifacts, and the code may be written in an ad hoc manner or drifted away from the original design idea due to evolution. The resemblance coefficient defined in this paper only considers the main factors related to functional cohesion. Although the weight ratio between data and control attributes was studied extensively, there are still some limitations. Software designers need to identify which clusters are functional clusters. They also need to decide where those singleton clusters should be placed. In addition, big data structures with more independent member variables tend to group different functional activities together and coupling may increase as a result of the restructuring.

In this paper, the restructuring approach was applied to a real telecommunication program and provided useful information. Different types of applications may have different features, which might affect the cohesion or similar-

ity measure. More experiments are still needed for other types of programs. In addition, the clustering results were only compared with the expected results. More objective criteria to evaluate clustering results should be developed in the future. The cohesion measure defined in (Anquetil and Lethbridge, 2003) is based on pairwise similarity measure and therefore it may not be entirely objective. And the value of the cohesion measure is very low because some

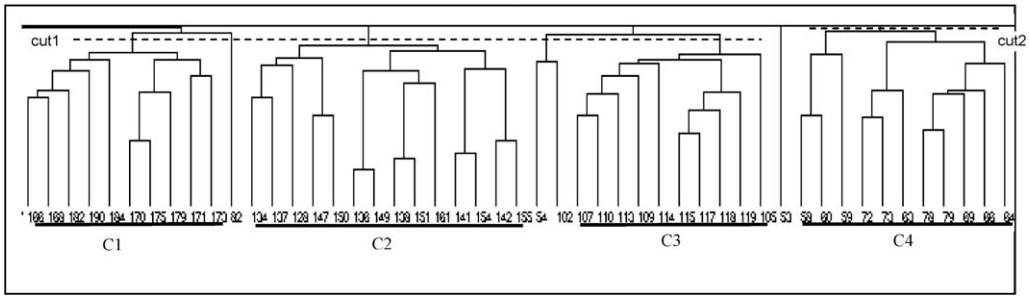

Fig. 13. Partial clustering tree of *rsvpTeRxRErr*.

```
void rsvpTeRxRErr(rsvp_pt rpdu, int cid)
{
...

RSTRACE_RSB(("rsvp_te: rx reservation
    error\n"));

phop = rsvpNode->n_rx_hop;
if(!phop) return;

espec = rsvpNode->n_rx_espec;
if(!espec) return;
// The following statement has been moved
// to a new function rsvpTeRSBGenRErr()
// flags = espec->u.ev4.flags;
ecode = espec->u.ev4.ecode;
eval = n2u16(espec->u.ev4.eval);

c = rsvpCircFromId(cid);
if(!c) return;

// The next new function corresponds
// to cluster C4 shown in Figure 13

rsvpTeDecodeFlowDescriptorFromRErr
    (rpdu, flow, filtss, &num_filt);

sess = rsvpNode->n_rx_sess;

stobj = rsvpNode->n_rx_style;
if(!stobj)
   return;
style = stobj->style;

if(!num_filt)
   return;

if((num_filt > 1) &&
   (style == STYLE_FF))
  return;

if((sb = rsvpTeSBFind(sess)) == 0)
   return;

sess_dst = XIPA(sess->u.te_sv4.dst);
if(!rsvpTeAddrInCirc(c, sess_dst, 32))
{
   if((ecode == REC_NOTIFY) &&
      (eval == REV_RRO_LARGE))
   {
   // The next new function corresponds
   // to cluster C3 shown in Figure 13

      rsvpTeEgressProcessLargeRROErr(
        sess, num_filt, filtss, ecode,
        eval, cid);
   }
   return;
}

if(ecode == REC_ADMIT)
{
// The next new function corresponds
// to cluster C2 shown in Figure 13

   rsvpTeProcessAdmissionErr(sb,
     num_filt, filtss, flow, phop,
     style);


   rsvpTeResvRefreshPhop(sb,phop,1);
}

// The next new function corresponds
// to cluster C1 shown in Figure 13

rsvpTeRSBGenRErr(sb, num_filt, filtss,
   flow, style, cid);
}
```

Fig. 14. Restructured *rsvpTeRxRErr*.

entities may not share any common attributes. How to quantitatively measure the cohesion still needs further research.

Acknowledgements

The authors would like to thank Dr. M. Zaid and Dr. R. Crawhall of NCIT (National Capital Institute of Telecommunications), Ottawa and Dr. R. Munikoti and Dr. K. Kalaichelvan of EION Inc., for supporting this research. The authors also want to thank the anonymous referees for their helpful suggestions.

Appendix

The following presents an example from the RSVP-TE protocol to show the effect of restructuring. Fig. 12 is a partial listing of the original code without variable declarations, comments, and white spaces. Hence, the line numbers are not continuous. Fig. 13 presents part of the clustering tree. The clustering tree demonstrates four relatively independent clusters, C1, C2, C3, and C4. The code can be more modularized by replacing those clusters with functions. Fig. 14 is the restructured code which consists of four new functions correspond to the four clusters shown in the clustering tree in Fig. 13.

References

Anquetil, N., Lethbridge, T.C., 2003. Comparative study of clustering algorithms and abstract representations for software remodularisation. IEE Proc. Softw. 150 (3), 185–201.

Anquetil, N., Fourrier, C., Lethbridge, T., 1999. Experiments with hierarchical clustering algorithms as software remodularization methods. In: Proc. Work. Conf. Reverse Eng., pp. 235–255.

Arnold, R.S., 1989. Software restructuring. Proc. IEEE 77 (4), 607–617.

Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V., Swallow, G., 2001. RSVP-TE: Extensions to RSVP for LSP Tunnels, RFC 3209.

Bieman, J.M., 1994. Measuring functional cohesion. IEEE Trans. Softw. Eng. 20 (8), 644–657.

Bieman, J.M., Kang, B.-K., 1998. Measuring design-level cohesion. IEEE Trans. Softw. Eng. 24 (2), 111–124.

Braden, R., Zhang, L., Berson, S., Herzog, S., Jamin, S., 1997. Resource ReSerVation Protocol (RSVP), RFC 2205.

Briand, L., Morasca, S., Basili, V., 1996. Property-based software engineering measurement. IEEE Trans. Softw. Eng. 22 (1), 68–86.

Chikofsky, E.J., Cross II, J.H., 1990. Reverse engineering and design recovery: a taxonomy. IEEE Softw. 7 (1), 13–17.

Choi, A.C., Scacchi, W., 1990. Extracting and restructuring the design of large software systems. IEEE Softw. 7 (1), 66–71.

Chu, W.C., Patel, S., 1992. Software restructuring by enforcing localization and information hiding. In: Proc. Conf. Softw. Maint., pp. 165–172.

Dhama, H., 1995. Quantitative models of cohesion and coupling in software. J. Syst. Softw. (29), 65–74.

Everitt, B., 1974. Cluster Analysis. Heinemann Educational Books, London.

Fenton, N.E., Pfleeger, S.L., 1997. Software Metrics: A Rigorous and Practical Approach. PWS Publication.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Girard, J.-F., Koschke, R., Schied, R., 1999. A metric-based approach to detect abstract data types and state encapsulations. Autom. Softw. Eng. 6 (4), 357–386.

Hutchens, D., Basili, V.R., 1985. System structure analysis: clustering with data bindings. IEEE Trans. Softw. Eng. 11 (8), 749–757.

Kang, B.-K., Beiman, J.M., 1998. Using design abstractions to visualize, quantify, and restructure software. J. Syst. Softw. 42, 175–187.

Kang, B.-K., Beiman, J.M., 1999. A quantitative framework for software restructuring. J. Softw. Maint.: Res. Pract. 11, 245–284.

Kim, H.S., Kwon, Y.R., 1994. Restructuring programs through program slicing. Int. J. Softw. Eng. Knowl. Eng. 4 (3), 349–368.

Lakhotia, A., 1993. Rule-based approach to computing module cohesion. In: Proceedings of the 15th International Conference on Software Engineering, pp. 35–44.

Lakhotia, A., 1997. A unified framework for expressing software subsystem classification techniques. J. Syst. Softw. 36, 211–231.

Lakhotia, A., Deprez, J.C., 1998. Restructuring programs by Tucking statements into functions. J. Inform. Softw. Technol. 40 (11–12), 677–689.

Lakhotia, A., Deprez, J.C., 1999. Restructuring functions with low cohesion. In: Proc. Work. Conf. Reverse Eng., pp. 36–46.

Lung, C.-H., 1998. Software architecture recovery and restructuring through clustering techniques. In: Proceedings of the Third International Workshop on Software Architecture, pp. 101–104.

Lung, C.-H., Zaman, M., 2004. Using clustering technique to restructure programs. In: Proceedings of the International Conference on Software Engineering Research and Practice, 853–858.

Lung, C.-H., Zaman, M., Nandi, A., 2004. Applications of clustering techniques to software partitioning, recovery and restructuring. J. Syst. Softw. 73 (2), 227–244.

Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y., Gansner, E.R., 1998. Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of the Sixth International Workshop on Program Comprehension, pp. 45–52.

Mancoridis, S., Mitchell, B., Chen, Y., Gansner, E., 1999. Bunch: A clustering tool for the recovery and maintenance of software system organizations of source code. In: Proceedings of the International Workshop on Program Comprehension.

Maqbool O., Babri, H.A., 2004. The weighted combined algorithm: a linkage algorithm for software clustering. In: Proceedings of the Eigth Euromicro Working Conference on Software Maintenance and Reengineering, pp. 15–24.

Mitchell, B.S., Mancoridis, S., 2001. Comparing the decompositions produced by software clustering algorithm using similarity measurements. In: Proceedings of International Conference of Software Maintenance.

Müller, H.A., Orgun, M.A., Tilley, S.R., Uhl, J.S., 1993. A reverse engineering approach to subsystem structure identification. J. Softw. Maint.: Res. Pract. 5 (4), 181–204.

Müller, H.A., Wong, K., Tilley, S.R., 1995. Understanding Software Systems using Reverse Engineering Technology. Object-Oriented Technology for Database and Software Systems. World Scientific, pp. 240–252.

Munson, C.J., 2003. Software Engineering Measurements. Auerbach Publications, ACRC Press Company.

Pressman, R.S., 1997. Software Engineering: A Practitioner's Approach, 4th ed. McGraw-Hill, Inc.

Romesburg, H.C., 1990. Cluster Analysis for Researchers. Krieger Publishing Company, Malabar, FL.

Rosen, E., Viswanathan, A., Callon, R., 2001. Multiprotocol Label Switching Architecture, RFC 3031.

Schwanke, R.W., 1991. An intelligent tool for re-engineering software modularity. In: Proceedings of the 13th International Conference on Software Engineering, pp. 83–92.

Sneath, P.H.A., Sokal, R.R., 1973. Numerical Taxonomy: The Principles and Practice of Numerical Classification. W.H. Freeman and Company, San Francisco.

Sommerville, I., 1996. Software Engineering, fifth ed. Addison-Wesley, England.

Tzerpos, V., Holt, R.C., 1998. Software botryology automatic clustering of software systems. In: Proceedings of the 20th Annual International Conference of the IEEE vol. 3, pp. 811–818.

Wen, Z., Tzerpos, V., 2004. An effectiveness measure for software clustering algorithms. In: Proceedings of the 12th International Workshop on Program Comprehension, pp. 194–203.

Wiggerts, T.A., 1997. Using clustering algorithms in legacy systems modularization. In: Proceedings of the Fourth Working Conference on Reverse Engineering, pp. 33–43.