Program Restructuring Through Clustering Technique

Xia Xu Chung-Horng Lung
Dept. of Systems and Computer Eng
Carleton University, Ottawa, Canada

Marzia Zaman
Cistel Technology
Ottawa, Canada

Anand Srinivasan
EION Inc.
Ottawa, Canada

Abstract

Program restructuring is a key method to improve the quality of ill-structured programs and therefore to increase the understandability and reduce the maintenance cost. It is a challenging task and much research is still ongoing. This paper presents an approach to program restructuring at the functional level based on the clustering technique 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 development and evolution phases. A new similarity measure is defined and intensively studied. The approach is used to restructure real industrial programs. 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.

1. Introduction

In software lifecycle, the software evolution usually accounts for more than 60% of total software costs [37]. Müller, et al., indicate that 50-90% of software evolution work focuses on program comprehension or understanding [29]. An ill-structured program usually contains functions involving multiple activities, which makes the program difficult to understand. In real-world software development, software products are usually driven by tight schedules. Hence software designers often emphasize more on the functional perspective than the non-functional quality attributes. It is difficult to maintain good design quality during the whole development process. Even though a software product is well designed, 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 program becomes difficult to understand and change, and therefore it is often costly to maintain.

Program restructuring is an important option in software evolution to improve deteriorated structure and to keep software maintenance cost under control. It is also used in software development to turn a poorly designed program into a well-designed one [3]. The early days of restructuring efforts focused on making a program's control flow easier to follow. This category is quite mature [3]. But for the functional structure, one challenge of restructuring is how to group meaningfully related code segments together inside of 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 practice.

Previous research on program restructuring at the function level primarily uses program slicing and input/output dependence techniques to restructure modules with cohesion as the main criterion [17-21]. Conceptually, their works are similar. The methods presented in [17-19] use data and control dependence information between input and output variables to measure cohesion and make restructuring decision. These methods, however, do not reflect the code fragments that are not directly related to the output variables. The tuck transformation presented by Lakhotia and Deprez [20,21] complements those methods in [17-19] by computing pairwise cohesion. The method considers data and control dependence between all variables, but its process is complicated and difficult to be used in industry. In addition, if a function has only one output variable, there is only one slice and therefore the function reaches the greatest cohesion.

In practice, especially in telecommunication program, it is common that some code fragments, such as error handling routines, may be not related to output variables. In such case, 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.

Cohesion, as an important measure in restructuring, is to measure how tightly related between elements 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 relationship between elements in the component. Previous articles of software clustering demonstrate research potential in software clustering field [38] and conclude that clustering methods may be a very good starting point for the remodularization of software [39]. But previous research on software clustering field is mainly concerned with software remodularization at the architecture level and has not been used in program restructuring at code level.

This paper presents an approach for program restructuring with the clustering technique at the functional level. It focuses on the automated support for identifying low cohesive function and helps make restructuring decision, instead of automated restructuring process. The purpose is to help software engineers identify ill-structured functions and give them heuristic advice. In detail, this paper discusses how to select entities and how to select attributes that are important to distinguish two different entities from the functional cohesion point of view. A new resemblance coefficient as a similarity measure is defined. Extensive experiments on the weight of different attributes are conducted. Three agglomerative hierarchical algorithms: single linkage algorithm (SLINK), complete linkage algorithm (CLINK) and unweighted pair-group method using arithmetic averages (UPGMA), are chosen in this paper. The comparison among them is made with case studies.

The structure of the rest of this paper is as follows. Section 2 reviews the related works in both program restructuring and software clustering areas. Section 3 proposes an approach for program restructuring with clustering technique and discusses the issues involved in the approach. Section 4 gives an extensive study on similarity measure by weighting attributes differently. Section 5 is a case study on an industrial program. Empirical observations are also summarized. Section 6 presents conclusions and future works.

2. Related Works

The closely related works on program restructuring at the function level use the program slicing and input/output dependence techniques with cohesion as a criterion [17-21]. Their ideas are similar, but they use different cohesion measures and different restructuring processes.

Kim and Kwon [19] present a method of restructuring an ill-structured module, which applies program slicing to extract processing blocks and identify multi-function module. The method uses module strength as a criterion to decide how to restructure program. The processing blocks refer to tightly coupled sub-modules, similar to the data slices in [6], in which a slice is a group of data tokens that contribute to a particular output variable in terms of data and control dependence. Based on code implementation, module strength is defined in terms of the level of sharing between processing blocks.

Kang and Beiman [17,18] introduce a method to restructure modules during the design or maintenance phase. The authors define input/output dependence graph (IODG) of a module, similar to the variable dependence graph (VDG) in [23], to model the data dependence and control dependence relationship between input and output components of a module. They also define association-based design-level cohesion (DLC) measure, and slice-based DLC and functional cohesion (FC) measures. Cohesion measures presented in these papers only consider dependence information between input and output components and do not reflect code fragments that are not related to the output components.

Lakhotia and Deprez [20,21] use tuck transformation to restructure program by breaking large functions into small functions. 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 rule-based approach proposed in [23] to compute pairwise cohesion between variables in the function as a criterion of restructuring. The empirical study in the paper [20] shows that the approach has some limitations for industrial applications.

Research on software clustering has also been done at design or architectural level. Tzerpos and Holt [38] present a survey and research potential of clustering approaches to software engineering, where they indicate that classic clustering techniques can be used in software context and there is research potential in software clustering field. Wiggerts [39] provides a general overview of clustering techniques and their applications to system re-modularization, where the benefit from the general theory of clustering analysis is highlighted. Lakhotia [22] gives a survey on subsystem classification techniques and provides a unified framework for entity description and clustering methods to facilitate comparison between various subsystem classification techniques.

Previous software clustering approaches concentrate on software system modularization or remodularization at the architectural or design level. They follow software engineering principle to obtain 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 [16,24,25,27, 28,30], or based on shared features [1,2,35], with or without giving weights to the relationships or features. Researchers use different information to measure the similarity based on different points of view.

The clustering algorithms used in previous works fall into three categories: hierarchical algorithms [1,2,16,24, 25,35], optimization algorithms [26-28], and graph theoretic algorithms [10,30]. The hierarchical algorithms are used the most. In hierarchical algorithms, the survey in [22] shows that most researchers prefer SLINK algorithm, but CLINK algorithm is suggested by [1] based on their experiments. It seems that different algorithms are suitable for the different applications.

Clustering analysis in software engineering is a sophisticated research domain. It is researchers' job to decide how to choose entities and entity's attributes, how to measure and compute similarity, and which algorithm to use for a particular clustering problem.

3. An Approach to Program Restructuring with Clustering Technique

This section presents an approach for program restructuring with the clustering technique at the function level and discusses key issues of clustering technique.

3.1 Program restructuring approach

The program restructuring approach proposed in this paper is supported by a suite of tools. The objective of program restructuring is to improve the structure or internal strength of a function. Cohesion is used as the main criterion. The approach is based on clustering analysis for the entities and their attributes extracted from source code. The existing structure of a program with quantitative measure is shown in a clustering tree after clustering analysis. The approach provides information about existing structure of a function, quantitative structure measure, and heuristic advice for improving existing code. It can be used to help software engineers make a decision - why and how to restructure an existing program. Figure 1 shows the approach for program restructuring with clustering technique. Currently, the study is conducted for C programs; however, the technique can be applied to other languages as well.

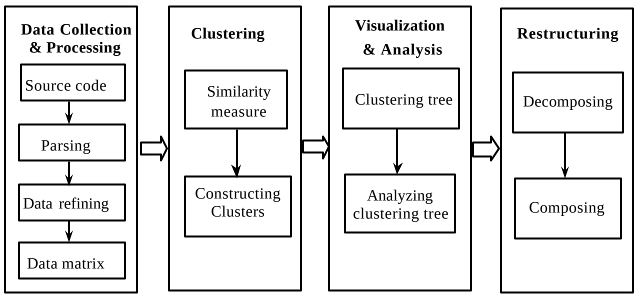

Figure 1. An approach of program restructuring

The approach has four key phases as shown in Figure 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) and therefore data refining is used to remove those noises. 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 similarity measure. When entities and their attributes are defined, a metric called resemblance coefficient is adopted to measure the similarity between entities, which is discussed in section 3.4. After resemblance coefficient has been defined, clusters can be constructed using a certain 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 result is displayed as a clustering tree. It shows the existing structure of a function. Closely related entities are grouped into a cluster. The degree of relatedness of two entities in a cluster is represented by a resemblance coefficient, which is also shown in the tree. By examining the tree, ill-structured code fragments can be identified, which are candidates for restructuring. Clustering tree provides heuristic advice on how to restructure a function. But software engineers must participate in making the final decision based on their experience, insights, and the restructuring objectives.

Phase four is the actual restructuring of a program. 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

To apply clustering to programs, we need to identify entities. 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, an entity is an executable statement that is related to a functional activity and can be described by its attributes (attributes are discussed in the next section). 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 is corresponding to the line number of a statement in the source code.

3.3 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. But selected attributes must contribute to the understanding of predefined objective criterion.

A statement consists of variables, constants, operators, key words, brackets, function names (in function call statements) and semicolon (for C programs). 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 key words 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 in the following:

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 of the entities. Data variables include local variables, global variables, and parameters passed to the 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.

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 to have 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 of an entity.

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 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, e.g., if or while, 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. Each attribute is measured on qualitative scale as 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. Between any two entities, there are six different types of matches described below.

- 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 cohesion and is represented with a resemblance coefficient. It has two features: Attributes and Matches.

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 degree of cohesion because they describe different dependence relationship between entities. Lakhotia [23] indicates that two variables which have data dependence relationship have higher cohesive than two variables that have 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 the 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 the 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. But at the function level, 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 or a parameter passed to the function may also have the same situation. Hence, a global variable and a local variable in a function play the same important role on function cohesion and therefore are treated equally.

Function call: A function name in a function call statement is treated as a data variable. In this 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 corresponding to different functionality. So 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. But 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 proposed approach, data attributes used in control entities are simply treated as control attributes.

Therefore, all data attributes in data entities are considered to have equal importance to functional cohesion, thus having the same weight. As the data attributes in control entities are treated as control attributes, 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.

Matches. Now, the problem is to determine the weights for matches that play different roles on similarity measure.

0-0 match: An 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 attributes and most of them are valued with 0. There are many 0-0 matches in the matrix. Lung et al. [24] address that counting 0-0 matches will generate distortion and result in dissimilarity. The study presented in [1] also shows that better results are obtained without considering 0-0 matches. In the 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 attribute and there is 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 that of control dependence, 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 on data dependence. Both contribute to the dissimilarity between entities. If matches on common data attributes (2-2 matches) play more important role on similarities 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 on 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 have more contribution to the similarity than that of 1-1 matches; and 2-0 / 0-2 matches have more contribution 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.

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

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

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 comparing to other attributes. Attributes in the same type are weighted the same and 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 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

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 [12] uses heuristic estimate to give data parameters twice weight as much as control parameters. Schwanke [35] estimates the significance of a feature by 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 decided through extensive experiments.

Different weight ratios are used in the resemblance coefficient to analyze 30 functions appeared in papers [5, 6,19,20], student assignments, and real industrial programs. The size of each function ranges from 8 lines to 55 lines (not including comments and white spaces). The experiments start with weight ratio of 2:1, that is, in equation (1), $w_d = 2$ and $w_c = 1$.

Throughout the experiments, all three clustering algorithms: SLINK, CLINK and UPGMA are used. CLINK tends to form large number of compact clusters and only gives good results for small examples. SLINK and UPGMA give similar results throughout the experiments. The results shown in this section are generated by UPGMA algorithm.

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 [5]. The example code is shown in Figure2, the entity-attribute input matrix is shown in Table1 and Figure 3 illustrates the clustering result.

```
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;
```

Figure 2. Sample code 1: Sum and Prod [5]

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

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

Figure 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 – computation of *sum*. Entity 10 uses the result of *sum* to compute average *avg*. Entities (5,8) contribute to the same activity – computation of product *prod*. The tree does not reveal the real functional structure in this example.

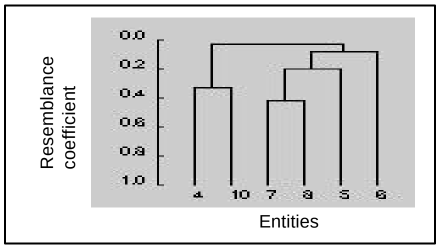

Figure 3. Clustering tree with 2:1 weight ratio for sample code 1 in Figure 2

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

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

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

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

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) together, respectively. Although data attributes are weighted twice as much as control attributes, it seems that control attributes still play a little bit more role on similarity measure than they should and more weight should be added to data attributes.

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

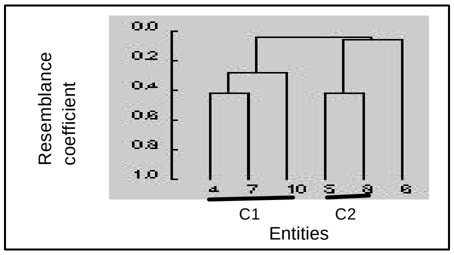

Figure 4. Clustering tree with 3:1 weight ratio for sample code 1 in Figure 2

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

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

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

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

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 Figure 5 is an example from an industrial program. It is the implementation of processing token body based on token type in a C code parser program. The main functional activity is to process token body with unreserved token type, which is implemented in the source code from line 27 to line 57.

Figure 6 shows the clustering result with weight ratio of 3:1. The cluster C1 is related to the activity of processing body with unreserved token type, which should be involved by entities between 27 to 54 as mentioned above, entity 26 is grouped with entities (18,20) because they share the same control attribute *token_type*. But entities 16 and 19 interleave the 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 3:1 weight ratio may play a little bit more role on similarity measure than they should. In the experiment, different weight ratios between 2:1 and 3:1 have been tested. Those ratios are 9:4, 7:3, 5:2, and 8:3.

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

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

Figure 7 shows that cluster C1 contains exact entities that are related to the activity of processing token body with unreserved token type. Entities 16 and19, which are inside the cluster in Figure 6 with the weight ratio of 3:1, are now outside the cluster. This is because the weight of data attributes is reduced. The relationship between entity 16 and entities (30,38,43,45) by 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 experiment.

```
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
      //...
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 }
```

Figure 5. Sample code 2: Process Body

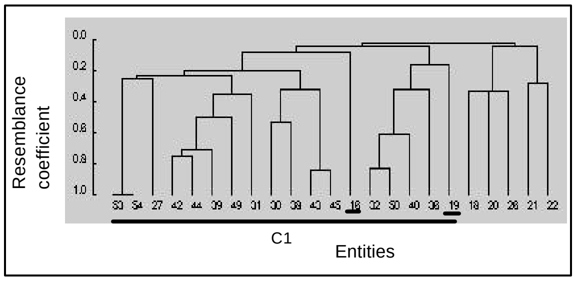

Figure 6. Clustering tree with 3:1 weight ratio for sample code 2 in Figure 5

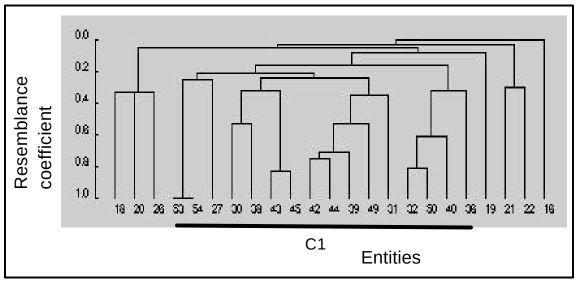

Figure 7. Clustering tree with 8:3 weight ratio for sample code 2 in Figure 5

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

5. Case Study

In order to evaluate how effectively the proposed approach can be applied to the real industrial software, the approach is used to restructure a real industrial program.

5.1 System under case study

The system under case study is a real network protocol RSVP-TE program in telecommunication industry. RSVP [7] is a resource reservation protocol that enables Internet applications to obtain different qualities of service (QoS). RSVP-TE [4] is a signaling protocol that extends the RSVP to support multiple protocol label switching (MPLS) [34] traffic-engineering applications. RSVP-TE provides a mechanism to establish and maintain explicitly routed label switched paths (LSPs) [34] with or without resource reservation.

The original RSVP-TE program was completed by schedule-driven. It was designed by providing only basic functionality for simple cases, and then was added more functionality during conformance test to satisfy the specification of the protocol. The emphasis of the software development is on functionality. RSVP-TE program was written in C with about 6,500 lines of code (LOCs) (in this paper, all LOCs do not include comments and white spaces) and 110 functions. Some functions are large and involve more than multiple functional activities. The understandability of the code was low. Maintaining and extending the code for additional functionality was less than desirable. In order to improve the design quality, 24 functions with total 2,147 LOCs code are selected as restructuring candidates. The function size ranges from 32 LOCs to 253 LOCs with an average of 89.46 LOCs.

5.2 Restructuring results

After clustering analysis, 17 functions out of 24 selected functions that are identified involve more than one functional activity. Some of them also have duplicated code or interleaved code. In order to support long-term maintainability and evolution, those 17 functions are chosen for restructuring. In order to compare restructuring results, Krakatau metrics tool is used to calculate metrics of size and cyclomatic complexity. Cohesion measure suggested by Anquetil and Lethbridge [1], in which cohesion of a function is the average resemblance coefficient between any two entities in the function, is used to measure function cohesion.

Table 2. Comparison before and after restructuring

Metrics	Before	After	changed
Average lines per function	93.24	37.29	- 60.00%
Average cyclomatic complexity of a function	19.47	7.69	- 60.50%
Average cohesion of a function	0.08	0.16	+100%

Table 2 gives the summary of comparison before and after restructuring. In the case study, the 17 poorly designed functions with total 1,585 LOCs are restructured, which represent 24.38% of the RSVP-TE program. After restructuring, 34 new functions are generated. Compared with original 17 functions, after restructuring, the average size of a function drops by 60% from 93.24 LOCs to 37.29 LOCs, the average cyclomatic complexity decreases by 60.5% from 19.47 to 7.69, and average cohesion increases by 100% from 0.08 to 0.16. The restructuring shows measurable improvement over the original functions. The complexity improvement is significant.

5.3 Empirical observations

In the case study, the restructuring approach is used for 24 functions in the RSVP-TE program. In general the approach works well and provides heuristic advice. The following presents a list of empirical observations and limitations.

Functional clusters. Related entities are grouped together to form a cluster. If the 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 used to cut 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.

Comparison of algorithms. In the case study, the restructuring approach has been experimented on all 24 functions with three clustering algorithms: UPGMA, SLINK and CLINK. There is only one function for which all three algorithms generate 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 for13 functions. But CLINK does not work well in the case study. The detailed results are not shown here because of page limits.

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

Non-functional clusters. A non-functional cluster refers to a cluster that dose 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 and etc. Usually a non-functional cluster is connected to a functional cluster and both of them together form a more completed functional cluster. But it may also appear independently. It is the software designer's responsibility to identify whether a cluster is a functional cluster or 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 to 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 are participated in different activities and these entities tend to be grouped together.

6. Conclusions and Future Directions

This paper presented a program restructuring approach using the clustering technique for C programs. The main focus was on the selection of entities and attributes, similarity measure, resemblance coefficient experiments, and the application of the approach to an industrial program. The main goal of the restructuring approach was to provide automated support to identify ill-structured low cohesive functions and give heuristic restructuring advice to software designers improve the cohesion of functions in both 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 similarity between entities with respect to cohesion. The experimental study of various weight ratios between the data attribute and the control attribute shows that the weight ratio of 8:3 (or 5:2) consistently generates the expected results for all selected examples under study. 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 of 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 main factors related to functional cohesion. Although the weight ratio between data and control attributes was extensively studied, there are still some limitations. Software designers 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 placed. In addition, big data structure with more independent member variables tends to group different functional activities together.

The main contribution of this paper is that a new resemblance coefficient as similarity measure for program restructuring at the function level was defined and intensively studied. The restructuring approach based on this resemblance coefficient was applied to an industrial program. The result showed that the heuristic advice provided by the clustering analysis was helpful.

In this paper, the restructuring approach was applied to a real telecommunication program and worked well. Different types of program may have different features which might affect the cohesion or similarity measure. More experiments are still needed for other types of programs. In addition, the clustering result were only compared with the expected result, objective criteria to evaluate clustering results should be developed in the future. The cohesion measure defined in [1] 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 entities may not share any common attributes. How to quantitatively measure the cohesion still needs further research.

Acknowledgement

The authors would like to thank M. Zaid and R. Crawhall of NCIT, Ottawa and R. Munikoti and K. Kalaichelvan of EION Inc., for supporting this research.

References

[1] Anquetil, N. and Lethbridge, T. C. (2003). "Comparative study of Clustering Algorithms and Abstract Representations for Software Remodularisation", *IEE Proc. on Software*, 150(3), pp.185-201.

[2] Anquetil, N., Fourrier, C and Lethbridge, T. (1999). "Experiments with Hierarchical Clustering Algorithms as Software Remodularization Methods", *Proc. of Working Conf. on Reverse Eng.*

[3] Arnold, R. S. (1989). "Software Restructuring", *Proc. IEEE*, 77(4), pp.607-617.

[4] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V. and Swallow, G. (2001). *RSVP-TE: Extensions to RSVP for LSP Tunnels*, RFC 3209.

[5] Bieman, J. M. and Kang, B.-K. (1998). "Measuring Design-Level Cohesion", *IEEE Trans. on Software Eng.*, 24(2), pp.111-124.

[6] Bieman, J. M. (1994). "Measuring Functional Cohesion", *IEEE Trans. on Software Eng.*, 20(8), pp.644-657.

[7] Braden, R., Zhang, L., Berson, S., Herzog, S. and Jamin, S. (1997). *Resource ReSerVation Protocol (RSVP)*, RFC 2205.

[8] Briand, L., Morasca, S. and Basili, V. (1996). "Property-based software engineering measurement", *IEEE Trans. on Software Eng.*, 22(1), pp.68-86.

[9] Chikofsky, E. J. and Cross II, J. H. (1990). "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, 7(1), pp.13-17.

[10]Choi, A. C. and Scacchi, W. (1990). "Extracting and Restructuring the Design of Large Software Systems", *IEEE Software*, 7(1), pp.66-71.

[11]Chu, W. C and Patel, S. (1992). "Software Restructuring by Enforcing Localization and Information Hiding", *Proc. of the Conf. on Software Maintenance*, pp.165-172.

[12]Dhama, H.(1995). "Quantitative models of cohesion and coupling in software", *J. of Sys. and Software*, (29), pp.65-74.

[13]Everitt, B. (1974). *Cluster Analysis*. Heineman Educational Books, London.

[14]Fenton, N. E. and Pfleeger, S. L. *Software Metrics: A Rigorous and Practical Approach*. PWS Publication.

[15]Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

[16]Hutchens, D. and Basili, V. R. (1985). "System Structure Analysis: Clustering with Data Bindings", *IEEE Trans. on Software Eng.*, 11(8), pp.749-757.

[17]Kang, B.-K. and Beiman, J. M. (1999). "A Quantitative Framework for Software Restructuring", *J. of Software Maintenance: Research and Practice*11, pp.245-284.

[18]Kang, B.-K. and Beiman, J. M. (1998). "Using Design Abstractions to Visualize, Quantify, and Restructure Software", *The J. of Sys. and Software*, 42, pp.175-187.

[19]Kim, H. S. and Kwon, Y. R. (1994). "Restructuring Programs through Program Slicing", *Int'l J. of Software Engineering and Knowledge Eng.*, 4(3), pp.349-368.

[20]Lakhotia, A. and Deprez, J. C. (1999). "Restructuring Functions with Low Cohesion", *Proc. of Working Conf. on Reverse Eng.*, pp.36-46.

[21]Lakhotia, A. and Deprez, J. C. (1998). "Restructuring Programs by Tucking Statements into Functions", *J. of Info. and Software Technology*, 40(11-12), pp.677-689.

[22]Lakhotia, A. (1997). "A Unified Framework for Expressing Software Subsystem Classification Techniques", *J. of Sys. and Software*, 36, pp.211-231.

[23]Lakhotia, A. (1993). "Rule-based Approach to Computing Module Cohesion", *Proceedings of the 15th Int'l Conf. on Software Eng.*, pp.35-44.

[24]Lung, C.-H., Zaman, M. and Nandi, A. "Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring", to appear in *The J. of Sys. and Software*.

[25]Lung, C.-H. (1998). "Software Architecture Recovery and Restructuring through Clustering Techniques", *Proc. of the 3rd Int'l Workshop on Software Architecture*, pp.101-104.

[26]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", *Proc. of Int'l Workshop on Program Comprehension*.

[27]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", *Proc. of the 6th Int'l Workshop on Program Comprehension*, pp.45-52.

[28]Mitchell, B. S. and Mancoridis, S. (2001). "Comparing the Decompositions Produced by Software Clustering Algorithm Using Similarity Measurements", *Proc. of Int'l Conf. of Software Maintenance*.

[29]Müller, H. A., Wong, K. and Tilley, S. R. (1995). "Understanding Software Systems Using Reverse Engineering Technology", *Object-Oriented Technology for Database and Software Sys., World Scientific*, pp.240-252.

[30]Müller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S. (1993). "A Reverse Engineering Approach to Subsystem Structure Identification", *J. of Software Maintenance: Research and Practice*, 5(4), pp.181-204.

[31]Munson, C. J. (2003). *Software Engineering Measurement*, Auerbach Publications, ACRC Press Company.

[32]Pressman, R. S. (1997). *Software Engineering: A Practitioner's Approach*, 4th Edition, McGraw-Hill, Inc.

[33]Romesburg, H. C. (1990). *Cluster Analysis for Researchers*, Krieger Publishing Company, Malabar, Florida.

[34]Rosen, E., Viswanathan, A. and Callon, R. (2001). *Multiprotocol Label Switching Architecture*, RFC 3031.

[35]Schwanke, R. W. (1991). "An Intelligent Tool for Re-engineering Software Modularity", *Proc. of the 13th Int'l Conf. on Software Eng.*, pp.83-92.

[36]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

[37]Sommerville, I. (1996). *Software Engineering*, 5th Edition, Addison-Wesley, England.

[38]Tzerpos, V. and Holt, R. C. (1998). "Software Botryology Automatic Clustering of Software Systems", *Proc. of the 20th Annual Int'l Conf. of the IEEE*, 3, pp.811-818.

[39]Wiggerts, T. A. (1997). "Using Clustering Algorithms in Legacy Systems Modularization", *Proc. of the 4th Working conf. on Reverse Eng.*, pp.33-43.