Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 73 (2004) 227–244
The Journal of Systems and Software
www.elsevier.com/locate/jss

Applications of clustering techniques to software partitioning, recovery and restructuring

Chung-Horng Lung [a,*], Marzia Zaman [b], Amit Nandi [c]

[a] *Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, ON, Canada K1S 5B6*
[b] *Cistel Technology, 200-210 Colonnade Road, Ottawa, ON, Canada K2E 7L5*
[c] *Nortel Networks, P.O. Box 3511, Station C, Ottawa, ON, Canada K1Y 4H7*

Received 17 April 2002; received in revised form 26 May 2003; accepted 26 May 2003
Available online 10 December 2003

Abstract

The artifacts constituting a software system are sometimes unnecessarily coupled with one another or may drift over time. As a result, support of software partitioning, recovery, and restructuring is often necessary. This paper presents studies on applying the numerical taxonomy clustering technique to software applications. The objective is to facilitate those activities just mentioned and to improve design, evaluation and evolution. Numerical taxonomy is mathematically simple and yet it is a useful mechanism for component clustering and software partitioning. The technique can be applied at various levels of abstraction or to different software life-cycle phases. We have applied the technique to: (1) software partitioning at the software architecture design phase; (2) grouping of components based on the source code to recover the software architecture in the reverse engineering process; (3) restructuring of a software to support evolution in the maintenance stage; and (4) improving cohesion and reducing coupling for source code. In this paper, we provide an introduction to the numerical taxonomy, discuss our experiences in applying the approach to various areas, and relate the technique to the context of similar work.

Keywords: Clustering; Software partitioning; Design recovery; Reverse engineering; Restructuring; Evolution; Cohesion and coupling

1. Introduction

Non-functional quality attributes such as maintainability and reliability are essential factors in controlling software life-cycle costs. It has been widely acknowledged that maintaining existing software accounts for as much as 60–80% of a software's total cost. Cohesion and coupling are two properties that have great impact on some critical software quality attributes, including maintainability. Therefore, management of cohesion and coupling is of critical importance for system design and cost reduction.

Cohesion refers to a component's internal strength, that is, the strength that holds the internal elements in a component together to perform a certain functionality. A component used in this paper is generic in that it could be a high-level architecture component; a module consisting of procedures; a procedure; a class; or even a variable. While cohesion is an intra-component property, coupling measures the interdependence among components. A desirable system partitioning should achieve high cohesion and low coupling, so that all the elements in one component are closely related for the realization of a certain feature, and changes made to that component will have as little impact as possible on other components. Alexander (1964) also postulated that the major design principle which is common to all engineering disciplines is the relative isolation of one component from other components.

Software engineering is a relatively new area compared to other well-established disciplines, such as mechanical engineering and manufacturing. Software partitioning is usually conducted in an ad hoc manner and is primarily based on the designer's experience. However, software systems may be either ill-designed, or often drift or erode over time due to changes in

* Corresponding author. Tel.: +1-61-3520-2600x2642; fax: +1-61-3520-5727.
E-mail address: chlung@sce.carleton.ca (C.-H. Lung).

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(03)00234-6

requirements and technology (Perry and Wolf, 1992). In other words, software evolves over time and is non-static, as a result of requirement changes. The resulting system could be highly coupled, which in turn creates problems for downstream software phases or evolution. Thus, effective partitioning or re-partitioning is needed. Effective partitioning or clustering is also a paramount goal in other disciplines. Clustering techniques have been used successfully in many areas to assist grouping of similar components and support partitioning of a system. In this research, clustering and partitioning are viewed as two sides of a coin. Partitioning is similar to a top–down approach to decomposing a system into smaller subsystems. Clustering, on the other hand, is a bottom–up method. With clustering, similar components are grouped together to form clusters or subsystems. Those clusters or subsystems are partitions which constitute a system.

In fact, partitioning or clustering analysis has been of long-standing interest and is a fundamental method used in science and engineering. The technique can facilitate better understanding of the observations and the subsequent construction of complex knowledge structures from features and component clusters. For instance, the technique has been used to classify botanical species and mechanical parts. The key concept of clustering is to group similar things together to form a set of clusters, such that intra-cluster similarity (cohesion) is high and inter-cluster (coupling) similarity is low. The objective—high cohesion and low coupling—is similar in software design.

Various clustering techniques have also been studied in software engineering. In this paper, we borrow some clustering ideas from established disciplines, and tailor them to software partitioning, recovery, and restructuring. The clustering techniques adopted in this paper are based on numerical taxonomy or agglomerative hierarchical approaches. Numerical taxonomy uses numerical methods to classify components. There are several reasons for adopting numerical taxonomy. The first is its conceptual and mathematical simplicity, as will be demonstrated in Section 2. Although its concept is simple, no scientific study has shown that numerical taxonomy is inferior to other, more complex multiversity methods (Romesburg, 1990). Another reason is that existing clustering techniques used in software engineering are often limited to only the reverse engineering process, based on source code. The approach presented in this paper can also easily be applied to various levels of abstraction and be used in round-trip engineering (e.g., both forward engineering and reverse engineering processes). Furthermore, the technique can provide more added value by facilitating software (design or code) restructuring, rather than simply design recovery. Lastly, the computation time is fast, which is an important factor if it is applied interactively or incrementally.

The objective of this paper is to examine existing numerical clustering techniques used in other well-established disciplines, tailor those techniques for various software applications, and present empirical studies of the techniques in software engineering. The approach has been applied to several projects at Nortel Networks and some of the results are presented in this paper. The rest of paper is organized as follows: Section 2 presents an overview of the clustering technique and discusses the method adopted for this research and the rationale behind it. Section 3 demonstrates several practical applications of the clustering technique to software partitioning, recovery, and restructuring. Section 4 discusses some lessons learned from applying the approach to various projects. Section 5 highlights some related work in software engineering. Finally, Section 6 presents the summary and discusses future directions.

2. Clustering

This section first describes the general concept behind the numerical taxonomy clustering technique. Following that, we will discuss the method adopted in this research.

2.1. General clustering concepts

Applications of clustering analysis can be found in many disciplines. Many clustering methods have been presented (Anderberg, 1973; Everitt, 1980; Romesburg, 1990; Wiggerts, 1997). In this paper, we focus on numerical taxonomy or agglomerative hierarchical approaches. Those approaches comprise the following three common key steps:

- Obtain the data set.
- Compute the resemblance coefficients for the data set.
- Execute the clustering method.

An input data set is a component–attribute data matrix. Components are the entities that we want to group based on their similarities. Attributes are the properties of the components. For example, the components could be people; the attributes, a set of responses to a medical test. A further example: the components could be mammals; the attributes, their dental formulas. Lastly, the components could be fossils; the attributes, their dimensions. The components and attributes can be many things in various areas, to which clustering analysis can be applied.

A resemblance coefficient for a given pair of components indicates the degree of similarity or dissimilarity between these two components, depending on the way in which the data is represented. For instance, the data may be represented by means of a binary variable. A 1 value may indicate that the component has the property.

On the other hand, if the data represents a misfit, then a 1 value stands for one possible kind of misfit or dissimilarity. A resemblance coefficient could be qualitative or quantitative. A qualitative value is a binary representation; e.g., the value is either 0 or 1. A quantitative coefficient measures the literal distance between two components when they are viewed as points in a two-dimensional array formed by the input attributes.

There are various methods of calculating the resemblance coefficients. This paper does not discuss those in detail. Instead, we will briefly illustrate an algorithm adopted in this paper and discuss some related approaches. In general, there are two types of algorithms for calculating resemblance coefficients, based on the scales of measurement used for the attributes. One type of resemblance coefficient can be computed based on qualitative input data or nominal scales for attributes; the other is based on quantitative input data or the attributes that are measured on ordinal, interval, or ratio scales.

An explanation follows of how one algorithm based on binary relations or nominal scales works. Table 1 shows three components with eight attributes. A 1 entry indicates that the attribute is present in the corresponding component. For instance, a 1 entry may mean a symptom is present in a person and 0 may mean it is absent. Component x in Table 1, then, consists of attributes 1, 3, 4, and 8; component y is positive to attributes 1, 2, 3, and 7. Components x and y share two common attributes 1 and 3, or these two components have two 1–1 matches. In other words, a 1–1 match means that the same attribute is coded 1 for both components. Similarly, there are 1–0, 0–1, and 0–0 attribute matches between two components. Let a, b, c, and d represent the number of 1–1, 1–0, 0–1, and 0–0 matches between two components.

Therefore, based on the definition, we obtain for components x and y that $a = 2$, $b = 2$, $c = 2$, and $d = 2$. Similarly, for components x and z, we obtain that $a = 1$, $b = 3$, $c = 3$, and $d = 1$; components y and z, $a = 3$, $b = 1$, $c = 1$, and $d = 3$.

To ascertain the similarity between two components, we calculate the proportion of relevant matches between the two components. In other words, the more relevant matches there are between two components, the more similar the two components are. There are different methods of counting relevant matches and there exist many algorithms to calculate the similarity or resemblance coefficient (Romesburg, 1990). Let c_{xy} be the resemblance coefficient for components x and y. Some examples are

- Jaccard coefficient: $c_{xy} = a/(a + b + c)$
- Russel and Rao coeffient: $c_{xy} = a/(a + b + c + d)$
- Simple matching coefficient: $c_{xy} = (a + d)/(a + b + c + d)$
- Sokal and Sneath: $c_{xy} = 2a/[2(a + d) + b + c]$
- Sorenson coefficient: $c_{xy} = 2a/(2a + b + c)$
- Yule coefficient: $c_{xy} = (ad - bc)/(ad + bc)$

The Jaccard, simple matching, and Sorenson coefficients are much used primarily because of their clear conceptual bases. The Jaccard coefficient is the ratio of 1–1 matches in a set of comparisons, without considering 0–0 matches. The simple matching coefficient counts both 1–1 and 0–0 matches as relevant. The Sorenson coefficient is similar to the Jaccard coefficient, but the number of 1–1 matches, a, is given twice the weight.

By applying the Sorenson matching coefficient to the example in Table 1, we get $c_{xy} = (2 \times 2)/(2 \times 2 + 2 + 2) = 1/2$. Likewise, $c_{xz} = (2 \times 1)/(2 \times 1 + 3 + 3) = 1/4$ and $c_{yz} = (2 \times 3)/(2 \times 3 + 1 + 1) = 3/4$. This procedure is repeated for each component pair in order to obtain the resemblance matrix. For this particular data representation, the higher a coefficient, the more similar the two corresponding components represent. Hence, components y and z in this example are the most similar pair, since the resemblance coefficient c_{yz} is the largest.

Clustering can also be performed on quantitative rather than qualitative data. Quantitative attributes may be measured on ratio scales, interval scales, ordinal scales, or a mixture of all three. Ordinal scales are widely used. For instance, $x = (3, 5, 10, 2)$ indicates that component x contains four attributes with values of 3, 5, 10, and 2, respectively. To calculate the resemblance coefficients based on the quantitative input data, an approach called the Euclidean distance coefficient is commonly used. Euclidean distance measures the literal distance between two components which are viewed as points in the space formed by their attributes. Stating this differently, the Euclidean distance e_{xy} between two components x and y is defined as

$$e_{xy} = \left(\sum_{i}^{n} (x_i - y_i)^2 \right)^{1/2}$$

Table 1
Input data matrix: an illustration

Component	Attribute							
	1	2	3	4	5	6	7	8
x	1	0	1	1	0	0	0	1
y	1	1	1	0	0	0	1	0
z	0	1	1	0	1	0	1	0

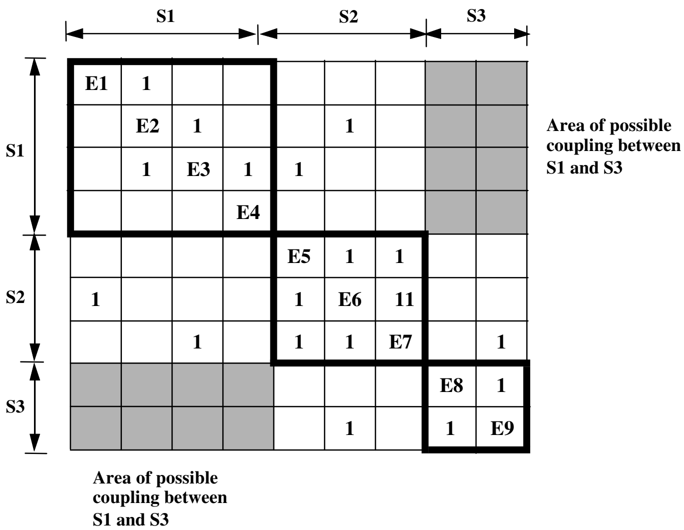

Fig. 1. N-square chart representation: an illustration (adapted from (Heyliger, 1994)).

where x_i and y_i are the values of attribute i for components x and y, respectively, and n is the number of attributes. Similar to the above example, we work on each component pair to calculate a Euclidean distance between these two components and populate the resemblance matrix. However, in this case, in general, the smaller the distance or the coefficient, the more similar two components are.

Given a resemblance matrix, calculated from either quantitative or qualitative data, a clustering method, the third step, is then used to group similar components. In essence, a clustering method is a sequence of operations that incrementally groups similar components into clusters. The sequence begins with each component in a separate cluster. At each step, the two clusters that are closest to each other (either the largest or the smallest coefficient, depending on your viewpoint) are merged and the number of clusters is reduced by one. Once these two clusters have been merged, the resemblance coefficients between the newly formed cluster and the rest of the clusters are updated to reflect their closeness to the new cluster. An algorithm called UPGMA (unweighted pair-group method using arithmetic averages; Romesburg, 1990) is commonly used to find the average of the resemblance coefficients when two clusters are merged.

With regard to the above example, component y and component z are first formed as a new cluster (y, z), since c_{yz} is the largest resemblance value. Recall that c_{xy} and c_{xz} are 1/2 and 1/4, respectively. The resemblance coefficient between the new cluster (y, z) and component or cluster x is then the average of c_{xy} and c_{xz}, which is $(1/2 + 1/4)/2 = 3/8$. The process repeats until all clusters are exhausted or a pre-defined threshold value has been reached.

Of further relevance is the representation of clusters. Three common methods are used to show the groupings of the components: matrix, graphical representation, and dendrogram. Using a matrix representation, the similarity between individual pairs is represented clearly. For instance, Fig. 1 shows an N-square chart presented in Heyliger (1994). The diagram depicts three subsystems (S1, S2, and S3) and their components. The traditional graphical representation easily identifies a group of related components. Take Fig. 2 as an example. The diagram shows three subsystems, S1, S2, S3, their components and the interrelationships among components. A dendrogram is a further example of a representation method. Dendrograms are widely used in clustering to demonstrate the process and the proximity between components. Fig. 3 illustrates the concept. In this example, the clustering steps are (a,c), (b,d), ((a,c), e), and finally ((a,c,e), (b,d)). The dendrogram grasps the relative degree of similarity among components or clusters. Fig. 3 also shows the resemblance coefficients between clusters. In general, the lower the level, the more similar the components or clusters.

2.2. Selection of a clustering method for software applications

This paper adopts the Sorenson coefficient, as described above, and tailors it to various software applications. Whether qualitative or quantitative data should be chosen depends on the applications and attributes. We have selected the qualitative values over quantitative primarily for the following reason: the information obtained from software that is reliable and applicable is static and often depends on specific scenarios. In other words, if component A is related to component B, a 1 value will represent the relationship; otherwise, a 0 value will be used. The relationship could be a function call, the inheritance in OO programming, or a shared feature.

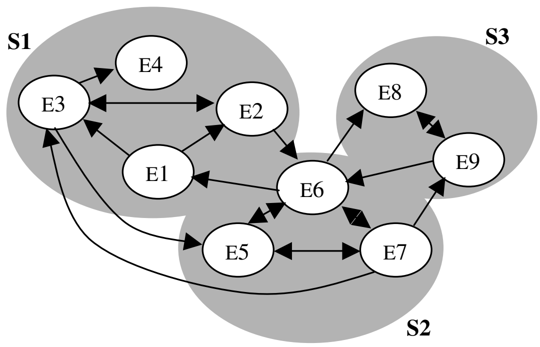

Fig. 2. A graphical representation of partitioning.

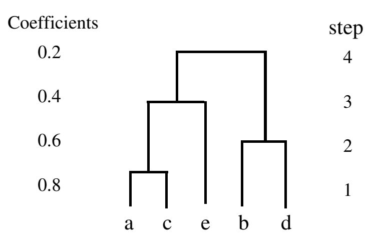

Fig. 3. An example of a dendrogram.

On the other hand, the number of function calls is an example of quantitative data.

The number of function calls based on syntax analysis, however, does not reflect the actual number of invocations of those functions. Dynamic information may be more insightful, but it is difficult to get and is highly dependent on run time behavior. Even if the number of invocations of a function could be accurately obtained dynamically, the number could distort the result due to the nature and complexity of the software. The point is further illustrated below.

Consider the following code fragment. Syntactically, functionB appears only once and functionC is called by functionA three times. That gives functionC a 3 value and functionB a 1 value in the input matrix, if quantitative data are selected as input. However, the value of MAX may only be determined at run time and the value could be huge in practical software design. Moreover, there may be complicated if–else statements inside functionA. The outcome of those logical statements can only be evaluated dynamically. Therefore, the quantitative values based on syntactic data can be misleading.

```
functionA () {
   for (i = 0; i < MAX, i++)
      functionB ();
   if (condition1)
      functionC ();
   else
      ...
      functionC ();
      ...
      functionC ();
      ...
}
```

Why have we chosen the Sorenson coefficient over other qualitative approaches? Both the Jaccard and Sorenson approaches ignore 0–0 matches. The argument of leaving d out of the formulation is that the joint lack of attributes or features between two components should not be counted toward their similarity. 0–0 matches may be relevant in some applications (Romesburg, 1990). From the software perspective, however, the argument that d should be ignored is favored because there may be a huge number of components (functions or methods), many of which usually do not share commonalities and have no relations. Counting 0–0 matches, d, in some methods, will generate a great deal of distortion, and the result will be skewed toward dissimilarity. The Sorenson coefficient also gives more weight to the 1–1 matches, because the attributes are present in both components at the same time. The idea of assigning heavier weights to more important factors is intuitive and has been used in many areas, including software engineering. In the area of software metrics, assigning twice the weight to certain metrics or attributes has also been used (Dhama, 1995). The Sorenson method was also successfully used to classify a number of simulation software models into a set of generic models, from which specific models can be instantiated (Mackulak and Cochran, 1990; Lung et al., 1994).

3. Applications of clustering to software

This section presents applications of the Sorenson method and demonstrates different ways to define and obtain the contents of the input matrix. By defining the matrix differently, we show the various uses of the clustering method. Specifically, examples include software partitioning, recovery, restructuring, and decoupling. A generic example is given first as an illustration of the technique. Some measures have also been adopted which can be used to quantitatively evaluate clustering techniques. The example is generic in the sense that the components in this example could be at different levels of detail.

We use the clustering technique in two different ways, depending on the available input data. The first approach is similar to that illustrated in Section 2.1, i.e., a set of component–attribute pairs are identified and used to calculate the resemblance coefficients. Section 3.1 presents a case study for software partitioning, based on the concept. However, for some software applications, there is no obvious component–attribute information. For instance, many existing software reverse engineering tools can extract coupling or dependency information. A typical example would be function–function or method–method call relationships. To make use of the clustering approach, we need to tailor the technique based on component–component independencies. In this case, the data set represents the interdependencies or interconnections among components instead of component–attribute relationships. Section 3.2 demonstrates a study in which the Sorenson coefficient was applied, based on component interconnections. Section 3.3 presents how the approach can be used to support software restructuring when used together with a scenario-based evaluation method. Finally, Section 3.4 demonstrates that the approach can be adopted to source code analysis with the aim of increasing cohesion and reducing coupling.

3.1. Software architecture partitioning based on components and features

This section presents an application where the mapping of components and functions/features information is available. The information can be derived by walking through the architecture with a number of scenarios or use cases. The input data set reveals a regular component–attribute matrix. An X entry in the matrix reveals that the component is involved in the particular feature. For instance, the function *Provide Prompt* has to do with components C1, C2, C3, and C6. Data like these can also be obtained from UML (Unified Modeling Language) diagrams, e.g., message sequence charts.

Based on the data, the clustering approach can be directly applied to support software architecture partitioning in the forward engineering process. Table 2 presents a mapping of functions to components for a telecommunications software architecture for illustration. Mapping of functions to components may affect software architectural quality attributes (Kazman et al., 1994). Section 3.3 presents an example that also echoes this point.

We evaluate the application of the clustering technique by comparing the result to the partitions performed by a group of architects through several iterations. The result of the clustering reveals a high degree of similarity. Out of 29 components depicted in the software architecture diagram, 21 components are clustered exactly the same. Six components are close to the architects' grouping, which is explained as follows. The original design has multiple levels of logical groupings in some areas. For example, Fig. 4 shows part of the design. There are two logical groups: *Service Framework* and *Command Interface*. The *Service Framework* consists of two clusters: *Protocol Handler* and *Service Handler*, which in turn consists of some components. The result of the clustering method indicates the following clustering steps: ((*Service Model*, *Service Policy*), ((*Facility*, *Party*), *Dialog*))). In other words, *Facility* and *Party* have a higher resemblance than that of *Facility* and *Dialog* in the actual design, as represented in Fig. 4. In addition, *Protocol Handler*, according to the clustering result, reveals a closer relationship with *Party* compared with *Service Handler*. Hence, three components, *Facility*, *Party*, and *Dialog*, are grouped slightly differently from the actual design. A similar result occurs with another three components, but this will not be discussed further here. Another two components, based on the clustering technique, however, reflect further discrepancies than those six components. The following paragraph explains the reasoning behind it.

There are two observations with regard to the discrepancies. The first is that two components, e.g., *Party*, serving as interfaces to a cluster or subsystem, are grouped together with the subsystem by the clustering method. However, the designers separated these two components as two stand-alone clusters. With regard to the two components that differ from the designers' partitioning, they act as factories or facilities. Hence, the "noise" level is high (high b and c values) for these two components, since these two components are involved in many features. As a result, the gap between the clus-

Table 2
Mapping of components and functions/features: a partial illustration

Function/feature	Component												
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
Provide prompt	X	X	X			X							
Collect digits	X	X	X		X	X	X						
Translate digits			X	X									
Select a route			X				X	X					
Verify authority			X		X								
Set-up logical connection			X						X				
Set-up physical connection			X			X			X	X	X	X	X
Generate ring		X	X			X							X
Handle answer	X	X	X			X			X		X	X	X
Handle release	X	X	X			X			X		X	X	X

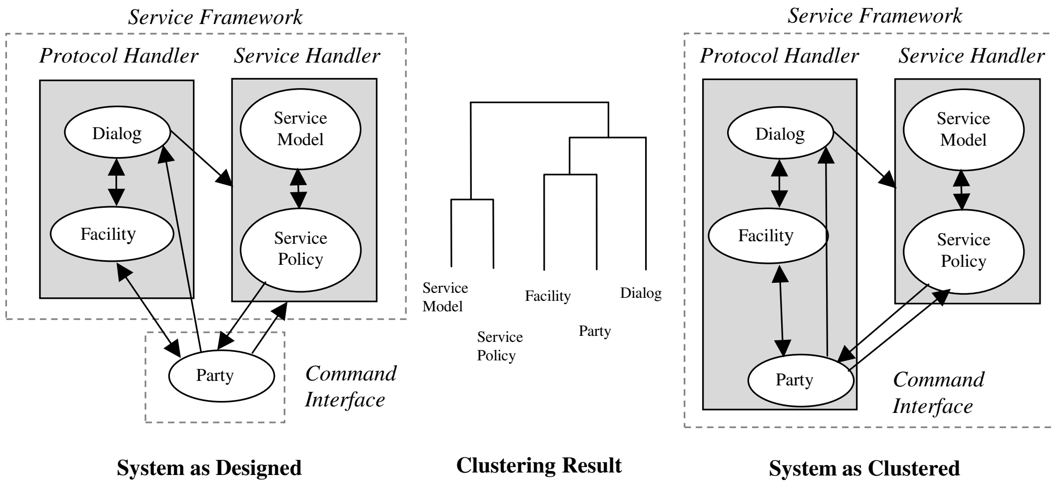

Fig. 4. A comparison of software architecture design and clustering results: an illustration.

tering output and the original design is wider for these two components.

3.2. Clustering based on component interdependencies

This section first presents a general example to demonstrate how the clustering method is used for system partitioning based on component interdependencies.

The term "component" used here for software applications is generic and does not necessarily represent a specific level of abstraction or an artifact. In other words, a component could represent a subsystem, a directory, a file, a class, a function, a data structure, a variable, and so on. In addition, the analysis could be applied to a mixture of various types of components, such as function and data structures. With regard to the software area, the input data represent interdependencies among components. For example, in Table 3, the 1 entries show that the corresponding components are interdependent. For components at the file or function level, the table represents the calling relationships among files or functions. For example, if the components in Table 3 represent functions, then function E1 calls E3, E6, and E9.

Note that the matrix is symmetrical and 1 entries are used in the main diagonal. This means that we do not distinguish between A calls B and B calls A, because A and B are coupled in either situation. The 1 entries in the main diagonal indicate that components are "coupled to itself" or closely related to itself. The rationale behind it is to obtain 1–1 matches with other components. Otherwise, a 1–0 match and a 0–1 match will be generated instead for each two components, which contributes to dissimilarity. Take Table 3 as an example. Component E1, shown in Table 3, is coupled with E3. Hence, there are 1 entries in (E1, E3) and (E3, E1). These two 1 entries, together with the 1 entries in (E1, E1) and (E3, E3), will constitute two 1–1 matches, which count for similarity for E1 and E3. Without the two 1 entries in the main diagonal, 1–0 and 0–1 matches will be used for the calculation of the Sorenson coefficients. The value of 1–1 match may even be 0 if there are no other common 1 entries. As a result, the similarity or interdependency between these two components will become much lower, even if they are related.

The approach involves only simple numerical computations and the overall computational complexity of the algorithm is $O(n^3)$ for an $n \times n$ matrix. The

Table 3
Matrix representation for component interdependency

	E1	E2	E3	E4	E5	E6	E7	E8	E9
E1	1	0	1	0	0	1	0	0	1
E2	0	1	1	1	1	0	0	1	0
E3	1	1	1	0	0	1	0	0	0
E4	0	1	0	1	1	1	0	1	0
E5	0	1	0	1	1	1	1	0	1
E6	1	0	1	1	1	1	0	0	0
E7	0	0	0	0	1	0	1	0	1
E8	0	1	0	1	0	0	0	1	0
E9	1	0	0	0	1	0	1	0	1

algorithm can also easily be modified to improve performance through the use of more efficient calculations, such as integer operations. This means that it requires only small computational effort.

Heyliger (1994) presents an example of coupling analysis. Figs. 5 and 6 show the component interconnections and an arbitrary grouping, respectively. To apply the clustering technique, the graph is converted to a matrix representation, as shown in Table 3. Given the input, we can calculate resemblance coefficients and apply the clustering method. The result for this example is shown in Fig. 7.

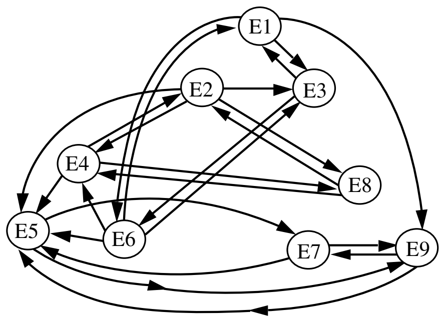

Fig. 5. An example of component interconnections.

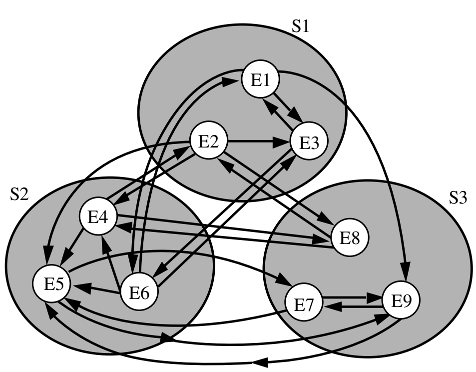

Fig. 6. Arbitrary grouping of components.

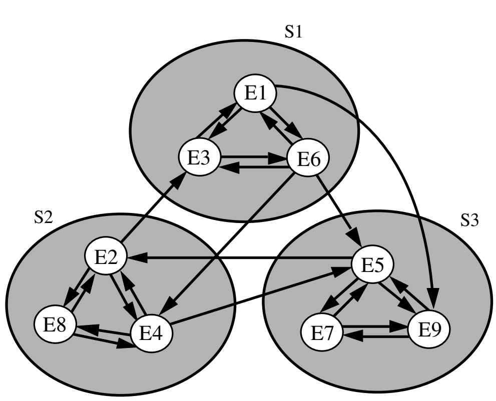

Fig. 7. Grouping based on clustering.

It is obvious that the inter-subsystem connection or coupling is significantly reduced between Figs. 6 and 7. To obtain some quantitative comparisons, two metrics are used and calculated. One metric is structural complexity (Card and Glass, 1990) and the other is system strength (Andreu and Madnick, 1977; Heyliger, 1994). The values shown in Table 4 reflect the significant improvement in terms of coupling. A negative value for system strength implies that the subsystem cohesion is less than the corresponding subsystem couplings to other subsystems.

3.2.1. Architecture recovery based on component interconnections

Clustering based on the component interconnections for design recovery is similar to the example just presented. This area has been discussed intensively in the literature. This section illustrates an application of the technique used together with open source or commercially available reverse engineering tools on a Nortel Networks product. Fig. 8 demonstrates the framework used for capturing software architecture. This approach is useful in better understanding software that has evolved over time or for joint venture programs. The approach may provide a different view and supports architecture recovery in the absence of documentation, or in cases where the document is outdated.

In order to apply the clustering technique, we first need to capture the coupling information or component interconnections. There are different types of coupling measures for object oriented software. For example, Briand et al. (1997) postulated three object oriented coupling metrics, i.e., class–attribute, class–method, and method–method. The coupling information or other coupling information may be captured using a reverse engineering tool.

This section presents a case study performed on a subsystem of object-oriented real-time software. The subsystem is written in C++ and consists of 213 classes, or about 75,000 lines of code. Two types of coupling information, namely method–method coupling (MMC) and method–parameter coupling (MPC), are gathered using a commercial reverse engineering tool. The coupling information represents component interconnections. The coupling measures are defined as follows:

- Method–method coupling (MMC)—If a method m_x of class c_i calls a method or is called by a method

Table 4
Quantitative comparison between two system partitions

	Structural complexity	System strength
Arbitrary grouping	23.3	−0.148
Grouping based on clustering	4	0.778

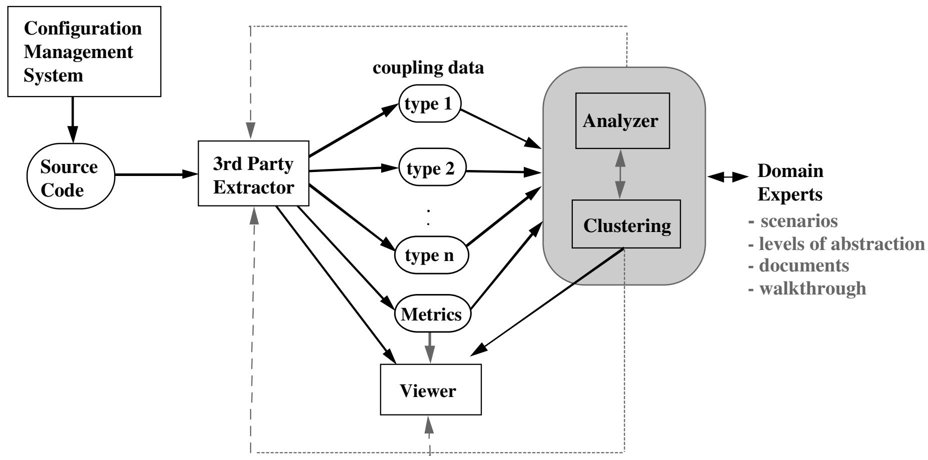

Fig. 8. Framework for applying clustering techniques for architecture recovery.

m_y of class c_j, c_i is considered to be coupled with class c_j by method–method coupling.
- Method–parameter coupling (MPC)—If the signature of a method m_x of class c_i has a reference to class c_j, c_i is considered to be coupled with class c_j by method parameter coupling. Note that c_i and c_j may be the same class.

Three different analyses are conducted based on each type of coupling. The clustering technique is applied separately to both data sets and a combination of the two data sets.

Again, we evaluate the results against the original design. In both cases, two main subsystems can be easily identified from the clustered output. The results are compared to the existing directory structure. Overall, there are 20 functional units or subsystems, including some non-critical units like library functions, error tracing, user interfaces, APIs, and some utility classes. Out of 213 classes, more than 70% of the classes are grouped similarly to the directory structure if we also apply MPC or MMC.

This study resulted in a number of interesting findings. First of all, it was observed that clusters based on method–parameter coupling measures give more accurate results than those based on method–method coupling measures. One reason for this could be the encapsulation feature of object-oriented programming. In other words, there are more intra-connections within a module than interconnections among modules. Another reason could be product-specific, as the system has self-defined data types as well as many data structures. The observation is also supported by real numbers. There are 399 MPC interactions and 142 MMC interactions.

After investigating the clustering results based on both coupling measures, a new data set was prepared using a linear combination of both measures. MPC was given twice the weight of MMC, based on the reason stated above. Clustering analysis was then performed on the combined data set. The clustering result using the combined data in some cases gave more insight than the others. The result revealed that more than 80% of the 213 classes correlate strongly with the existing design in terms of class grouping. The number is higher than those using MPC or MMC individually.

Table 5 shows some quantitative methods applied to three important subsystems, identified using the clustering technique. The table includes the following metrics for the three analyses, using MMC, MPC and combined data, respectively.

- the number of modules;
- number of intra-cluster or internal couplings (IC), defined as the number of links to other components within a subsystem;

Table 5
Cluster metrics for three analyses

Subsystem	Number of classes	Method–method coupling (MMC)			Method–parameter coupling (MPC)			Combined		
		IC	EC	SS	IC	EC	SS	IC	EC	SS
Protocol	15	27	0	0.12	69	12	0.3026	71	12	0.3115
Access	14	14	1	0.071	16	29	0.0712	16	30	0.0708
Resource	10	10	1	0.1	28	2	0.279	28	2	0.279

- inter-cluster or external couplings (EC), defined as the number of links to other subsystem components; and finally
- system strength (SS) (Heyliger, 1994), defined as the difference between the subsystem cohesion or internal coupling and the corresponding subsystem external couplings to all other subsystems.

Note that MMC and MPC are not totally disjointed from the coupling perspective. In other words, a method within a class may contribute to both MMC and MPC. In this case, only one is counted toward the combined data set.

Table 5 also reveals that the *Access* subsystem has more external couplings (30) than its intra-cluster couplings (16) shown in the combined column. The system strength metric is also low for *Access*. Further investigation confirmed that the design of this particular subsystem is highly coupled with many other subsystems and it also provides some utility-type of functionalities to other classes. This explanation conforms with the resulting outputs obtained from the clustering technique.

The clustering does not only show the clusters, but also indicates the closeness of the clusters. This point is explained below and is demonstrated in Fig. 9. A partial dendrogram which shows three subsystems, based on MPC, is presented in Fig. 9. For simplicity, the actual class names have been replaced with numbers. It has been verified from the original design that the modules p1, p2, and p3 represent the protocol modules; m1, m2, and m3 represent the mapping modules; and t1 and t2 represent the transaction modules. The outcome matches the design: p1, p2, and p3 are only variations of the same type of protocol, but used for a different marketplace. Therefore, p1, p2, and p3 share common inheritance properties. Similarly, m1, m2, and m3 are in the same group, and t1 and t2 are in the other group.

The results obtained from the combination of MMC and MPC convey a slightly different view. Visually, Fig. 10 shows the relationships. Some parts of *Protocol* and *Mapper* are grouped together (p1, m1) and (p2, m2), and parts of the *Mapper* (m3) show more closeness with some classes in the *Transaction* (t1). Fig. 9 is mainly dominated by inheritance properties, whereas Fig. 10 includes method–method relations between classes. Therefore, the groupings are evident, such as (p1, m1) and (p2, m2).

We compared the observations with the actual design. The *Protocol* module performs decoding and encoding of a message. This module contains two main portions, p1 and p2, each having a corresponding portion in the

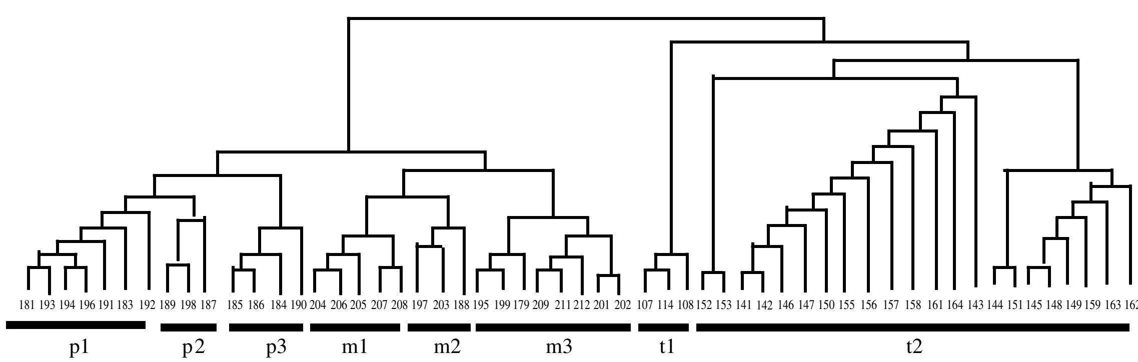

Fig. 9. Dendrogram based on method–parameter coupling.

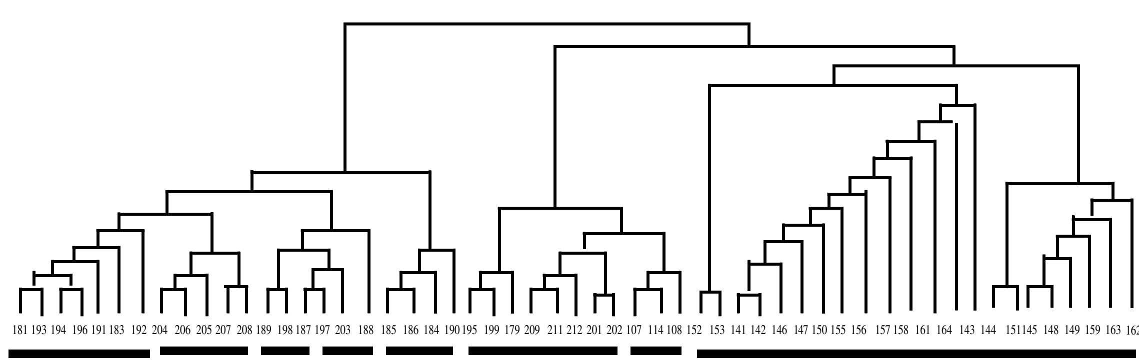

Fig. 10. Dendrogram based on combined data.

Mapper (m1 and m2) that translates the format to the *Transaction*-specific representations. There is also a set of classes in the *Protocol* (p3) and the *Mapper* (m3) that provides generic methods and translations. Three classes (t1) in *Transaction* are highly coupled with the *Mapper* (m3). The explanation for this is that they are used for some special types of data structure and data conversion for internal data representations within the application. The rest of the classes (t3) in *Transaction* are more general purpose components.

The difference between Figs. 9 and 10 is mainly due to different types of interconnection data. Different input can provide different viewpoints which can be useful for practical object-oriented applications. The reason for this is that object-oriented designs are sometimes documented separately for each subsystem, using class diagrams. Interactions between classes in different subsystems or teams are usually not well-documented or understood. Fig. 10 reveals the close relationships for (p1, m1), (p2, m2), and (m3, t1). It also shows the actual classes based on the class interdependencies. This type of information is usually hidden in the design document.

The difference between the clustering output and the existing partitions helps the designer in comparing various artifacts and evaluating the architectural design. The idea is similar to the software reflexion model technique (Murphy et al., 2001). This process may also lead to the effort of software architecture restructuring, which is discussed in the next section.

3.3. Software architecture restructuring in support of evolution

This section presents a study of architecture restructuring for a real-time telecommunications system with the support of the clustering technique. The problem with the system was that the time required to introduce new services was often longer than expected. Whenever a service group wanted to add a new service or make minor modifications to the existing services, the service designers had to spend extra effort dealing with another design group, working on unanticipated details. The server group did not think the effort was necessary. The goal was then to reduce the maintenance effort by evaluating the software architecture through a decoupling task.

We applied the Objective/Scenario/Metric paradigm outlined by Lung and Kalaichelvan (2000) to this problem. The objective was to decouple two subsystems in order to reduce the evolution time. A few critical use cases or scenarios were then identified to evaluate the software architecture, based on the objective. Those scenarios dealt primarily with service and protocol handler registration during the initialization phase and service execution during the execution stage. We used the clustering technique together with those scenarios and worked with some designers. However, this approach distinguishes from others in that we apply the clustering to scenarios individually. In other words, for each scenario we identified the components and interconnections that were involved for that particular scenario only and then performed the evaluations, including clustering analysis. Fig. 11 shows only relevant components and connections for one particular scenario, service and protocol handler registration of the original design. It contains a global table and two subsystems, *ServiceLogic* (sl) and *Protocol* (prot). The clustering analysis for this study was based on component interconnectivity.

This specific scenario analysis indicated that some components between these two subsystems were highly coupled, due to the many interactions between components and data copying among components in these two subsystems during the initialization stage. The outcome of the clustering showed two clusters (5, 7, 2, 1, 3) and (9, 10, 6, 8, 4), as presented in Fig. 12. This figure also indicates that components 5 (prot_sl_protocol_support) and 7 (prot_transaction_mapper) are more closely

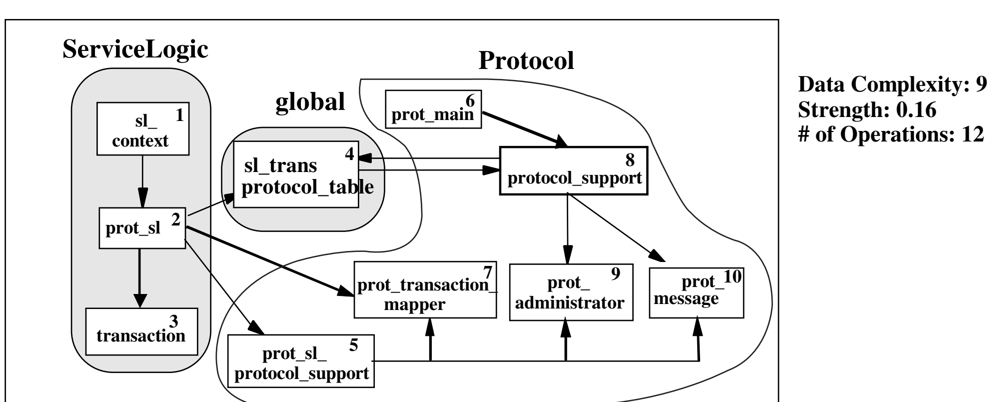

Fig. 11. Initialization and registration of protocol handlers—original design.

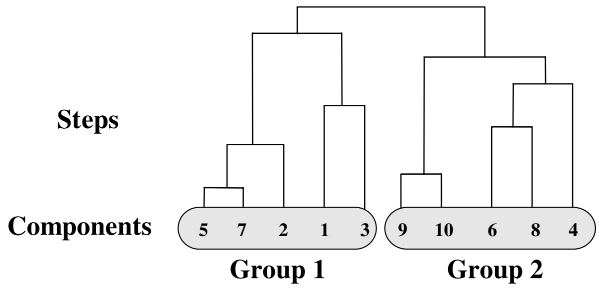

Fig. 12. Clustering of components.

related to the subsystem *ServiceLogic* than *Protocol*, which is different from the conceptual design.

The gap between the actual design and the conceptual model triggered us to conduct further reasoning of the design. The result of the clustering helped us focus on components 5 and 7, and their interconnections with other components. The concept is similar to the reflexion model presented by Murphy et al. (2001). Clustering is used in our study to help in the reasoning process. By examining the design and implementation, we identified the areas in which these two components are highly coupled. Those two components were involved with duplicated data copying operations and methods inconsistent with other designs in other components. Specially, the initialization method for class 7 should be implemented in class 6, similar to those used for classes 9 and 10. Instead, the initialization method for class 7 was actually in class 2 in the original design. After carrying out the validation and restructuring analysis, we moved certain methods from one class to another. Fig. 13 demonstrates the new design. For this example, the changes included moving an initialization method from class 2 to class 6, which helped eliminate the interaction between classes 2 and 7; moving data copy operations from class 2 to class 6; and a method with some modifications from class 8 to class 7, which supported the removal of classes 3 and 4 (with regard to this specific use case).

Both the original design and the new design perform the same functions, but their structures are significantly different. The coupling is significantly improved. The unnecessary coupling between *ServiceLogic* and *Protocol* is removed. Two metrics were used as measurements: structure complexity (Card and Glass, 1990) is reduced from 9 to 1, while the strength (Heyliger, 1994) is improved from 0.16 to 0.4. In addition, the number of operations is also reduced from 12 to 10. The new design also supports the dynamic link of new protocol handlers. By moving the method from class 2 to class 6, the registration of class 7 is consistent with classes 9 and 10, which facilitates future maintenance. More importantly, the main objective of this effort has been achieved after those modifications were carried out, i.e., changes can be handled by one designer in the *Protocol* subsystem instead of two designers in two subsystems working together.

The ideas presented in this section are not completely new. A similar approach has been discussed in the literature (Murphy et al., 2001). This example, however, demonstrates that the clustering analysis can be used to select areas to focus, which could be useful in comparing the actual design and the conceptual model. This example also illustrates that evaluation can be more effective if scenarios are analyzed individually rather than collectively. Again, the connections shown in Figs. 12 and 13 are based on one scenario. Separating the scenarios could provide more insights as to why components are connected. There are still other connections or dependencies between these two subsystems, but they are irrelevant to this scenario. If, on the other hand, we construct the structure based on all the scenarios, the chance of identifying specific problem areas for improvement will generally become lower. This is because, in many cases, a

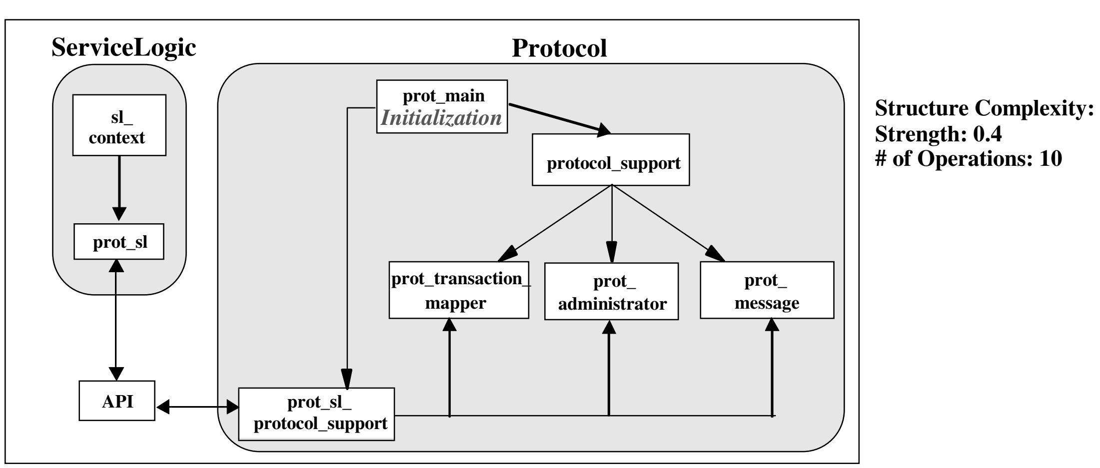

Fig. 13. Initialization and registration of protocol handlers—new design.

system may contain far too many connections that are not necessary for specific analysis. Take this problem as an example: if we also consider scenarios for service execution at the same time, classes 5 and 7 in Fig. 12 exhibit stronger relations with components in the *Protocol* subsystem, which is consistent with the conceptual model. In this case, it does not reveal a potential point for high coupling or dependency. From the clustering point of view, those extra connections produce "noise" in the calculations of resemblance coefficients.

3.4. Source code decoupling

This section presents an application of clustering to source code decoupling at the procedure level. We demonstrate how to define the input matrix that can be used for the clustering purpose.

There are metrics proposed for measuring cohesion and coupling at the code level (Bieman and Ott, 1994; Bieman and Kang, 1998; Dhama, 1995). Dromey (1996) presented a conceptually simple approach based on a graphical representation to increase cohesion for a procedure by identifying the relationship among input and output variables. Two output variables are cohesive if they depend on at least one common input variable. Bieman and Kang (1998) presented a similar concept, but they used it mainly for measuring program cohesion.

The identification of closely related variables is similar to a classification of variables into separate cohesive clusters. The main issue with Dromey's graphical representation is scalability. As the number of variables increases, the manual task becomes more difficult. The clustering method can remedy this problem. In this type of application, each variable can be treated as a component. The relationships between input and output variables are their interdependencies. For instance, the example given in Dromey's (1996) paper is illustrated as follows.

$a, b, c := \text{Name}(v, w, x, y, z)$
...
$a := v + w;$
$b := w * x;$
$c := y/z$
end_Name

An input data set for clustering can be derived by identifying the interdependencies between input variables and output variables, as depicted in Table 6. By applying the clustering method, variables a and b are grouped in one cluster, while variable c has nothing to do with a and b. This suggests that the procedure should be split into two more cohesive procedures, one containing the first two statements, and the other, the third statement.

Table 6
Interdependencies between input and output variables

	v	w	x	y	z
a	1	1	0	0	0
b	0	1	1	0	0
c	0	0	0	1	1

The motivation of this application is to help reduce the complexity of complex modules or functions based on metrics extracted from the source code. Software metrics are used for diagnosing module complexity, but they often fall short of providing a guideline to improve the code or reduce the complexity.

The input data for clustering can also be simplified by treating each statement as a component (Lung and Zaman, in preparation). That way, statements do not need to be divided. An example from Bieman and Kang (1998) is described below. The example demonstrates a feature called communicational cohesion.

```
Procedure Sum_and_Prod (n: integer; arr: int_array;
var sum, prod: integer; var avg: float)
var i:integer;
begin
1.  sum = 0;
2.  prod = 0;
3.  for i = 1 to n do begin
4.     sum = sum + arr[i];
5.     prod = prod * arr[i];
6.  end;
7.  avg = sum/n;
end;
```

The input data for this procedure is shown in Table 7. Statements 3 and 6 are not considered, since they are used as utilities. The input data can be fed into the clustering method. As a result, statements 1, 4, and 7 are grouped together, and statements 2 and 5 are in another group. More importantly, these two groups share a low resemblance coefficient. Therefore, the procedure can be divided into two high cohesive procedures: one calculating the sum and the average; the other computing the product.

Table 7
Input data for a procedure (0 entries are not shown)

Statement number	sum	prod	n	arr	avg
1	1				
2		1			
4	1			1	
5		1		1	
7	1		1		1

4. Lessons learned

As discussed earlier in this paper, the approach has been applied to various software applications. Specifically, we have applied the approach to software architecture partitioning at the design stage, design recovery in the reverse engineering process, software restructuring to support evolution, and source code decoupling. The approach provides a useful method for revealing the degree of similarity or coupling among components.

When applied to partitioning, as discussed in Section 3.1, there is a pre-condition for obtaining a more accurate outcome, i.e. that the scenario coverage used to walk through the architecture and components should be high. The particular architecture presented in Section 3.1 has been studied and evaluated with a large number of scenarios. That high scenario coverage makes the clustering result useful and reliable.

Scenarios are also useful in dividing the system complexity if they are used individually. Evaluation of the software based on many scenarios at one time may overlook some problematic areas. Scenarios, however, if evaluated separately, may give further insight. The study of restructuring, as discussed in Section 3.3, is a good example. If many other scenarios were analyzed at the same time, the result may not reflect much difference due to the many other interactions or connections.

We have also applied the approach to other projects in telecommunications and network, in cases where we had no prior knowledge of the systems. For one project, we first extracted the function calls relationships. The information was then collected and abstracted up one level higher to reveal the coupling between files. The clustering method was used to recover the software structure based on the file coupling data. The recovered architecture resembled the system to a high degree. There was an interesting observation. Two files did not appear relevant based on their names, but showed a strong relationship as a result of using the clustering method. A designer later confirmed after going through the code that these two files had actually been one file before a split because the size of the file had become too large.

For another project, however, the results were not as successful when the clustering method was applied at the function level. This was because the system was extremely coupled. The resemblance coefficients, therefore, were very low for most component-pairs, which means that the internal cohesion or degree of similarity is low. For cases like this, it would be difficult to map the clustering hierarchy to a partition.

When used for architecture recovery, it is likely that discrepancies will exist between the clustering results and the existing partitions performed by the designer. We have identified some possibilities based on the experience.

- The reverse engineering tool may not generate complete and correct information, especially for object-oriented applications. We have used various tools. Some tools are based strictly on the syntactic information, while some need to be compiled with the code in order to generate various types of dependency relations. Different tools usually generate different information.
- The problem scope may cause some issues. If only a subsystem is examined, then it is likely that not all dependency relations will be included. For instance, Table 5 indicates that the value of the inter-cluster for MMC of the subsystem *Protocol* is zero. This is because *Protocol* is invoked by another subsystem, which is not included in the analysis.
- Some components may have far more connections than most of the other components. Typical examples are shared utility functions, APIs, and factory-type classes. The grouping of those components may be unpredictable. The reason is that there are many 1–0 and 0–1 matches (higher b and c values) in this case. According to the Sorenson coefficient, large b and c values will result in a small resemblance number.
- Some modules may perform a specific task and the number of files in this module may be much smaller than those in many other modules. In a system that we studied, for instance, three modules had a very small number of files or classes. Specifically, one module had only one file; another, two files; and the other, four files. This may result in some distortions. The reason is similar to that described in the previous point (some components may have far more connections than most of the other components).
- The design may not be well partitioned. In this case, the gap could actually help the designer better understand or restructure the software.
- There may be a specific rationale for the design. A typical example is that some critical parts are hard-coded and/or highly coupled for the purpose of performance.
- Limitations of the clustering techniques. No matter what clustering technique is adopted, there is always a chance that the method may generate unexpected results or will not generate expected results. Expert involvement is recommended for postmortem analyses.

We also have observed some liabilities specific to the numerical taxonomy-based clustering technique for software applications:

- There is no clear-cut value in determining clusters. Simply stated, there is no clear rule for mapping a dendagram to a partition. The resemblance coefficients differ from application to application. For cases where many coefficients are small and close, it is difficult to decide where to make the cut.

- The technique is sensitive to input data. This is also common to many numerical taxonomy approaches. The point has also been addressed in the previous sections. Various methods of coping with this problem have been discussed in the discipline.

5. Related work

Applications of the clustering concept specific to the software partitioning have been studied. Andreu and Madnick (1977) applied the partitioning concept to a database management system in order to minimize coupling. The requirements and their interdependencies were first identified and were converted to a graph problem. Various alternatives for partitioning were examined and a quantitative metric was calculated for each alternative. The alternative with the lowest value of coupling was chosen as the optimal partitioning.

Heyliger (1994), on the other hand, has proposed using *N* square charts to partition a large system into subsystems. For an *N* square chart, the rectangles along the main diagonal represent the system partition. Fig. 1 shows an example of an *N* square chart with three subsystems, S1, S2, and S3. The 1s within a subsystem indicate internal strength or cohesion. The 1s outside the partitions or clusters represent coupling among subsystems.

The objective of Heyliger's approach was to incrementally refine the design to maximize cohesion and minimize coupling. Heyliger has identified a set of patterns that characterize specific interfaces among system elements. These patterns provide mechanisms for system structural reorganization and refinement for low inter-subsystem couplings. This process, as depicted by the author, is labor intensive and the rearrangement of the elements is a major problem even for small or modest systems.

Both of these papers share a common goal, which is to minimize the interconnectivity among components. Selby and Reimer (1995) presented an analysis on the interconnections of components and software and system errors. They also discussed various approaches to clustering software, based on component interconnections. Lakhotia (1997) has also conducted a survey on different subsystem classification techniques that have been proposed for classifying software into a particular subsystem. The main objective of this paper is to present a unified framework for expressing subsystem classification techniques. Hutchens and Basili (1985) specifically applied clustering methods with data bindings to represent system modularization. However, this technique and other approaches (Anquetil and Lethbridge, 2000) centered around the code level and/or were only used in reverse engineering process. Here, we demonstrate various methods of applying the clustering technique to system partitioning, recovery, and restructuring.

Schwanke (1991) and Mancoridis et al. (1998) also presented clustering-based methods to generate high-level designs from source code. They adopted neighboring algorithms and genetic algorithms to perform the clustering task. The computation time for these algorithms is very high, even for moderate numbers. Mancoridis et al. (1998) show that for 153 modules which have 103 module level relationships, the execution time is over 1 h. In contrast, the example that will be presented in Section 3.3 has a much larger number of entities (213 classes) and relationships (541 interactions), and the execution time is almost instantaneous on a similar machine. We have also tested our method on their example. The result shows that 13 out of the 16 modules are grouped in the same cluster as theirs. In fact, their algorithms may produce different results for different runs, because the algorithms depend on the probability of randomly selecting the initial partition.

Sartipi and Kontogiannis (2001) presented clustering based on maximal association. Conceptually, their approach is similar to our work in that both methods consider a number of common features for components and determine the cohesion as the degree of sharing different features between components. The property is also referred to as the association coefficient. Davey and Burd (2000) consider the association coefficient as the most suitable property for detecting similar entities. One difference between our work and their research is that ours is based on mathematically simple numerical hierarchical taxonomy, whereas Sartipi and Kontogiannis adopt a complicated optimization search algorithm based on the data mining technique, apriori. During the clustering phase of their approach, the user may also need to interact to reduce the search complexity. However, the numerical taxonomy usually generates similar results.

If the clustering method is only used once, as in the case of many reverse engineering practices, efficiency is less important. If, however, as Wiggerts (1997) pointed out, software systems will continue to evolve after the system has been modularized or remodularized, incrementally updating a modularization due to evolution is often needed. In addition, the results from the clustering analysis may provide a method for supporting the evaluation of various design alternatives. In applications like these, the clustering algorithm will be applied interactively and incrementally, and response time plays a vital role in supporting this process.

As Selby and Reimer (1995) pointed out, interconnectivity-related approaches provide a useful means for analyzing software. In our experiments, we also agree that clustering based on interconnectivity of entities is practical and useful.

Another approach to architecture recovery is to group files or procedures based on their names. Approaches based on file names work to some extent and may be successful for some specific systems (Anquetil and Lethbridge, 1998; Neighbors, 1996). However, these approaches are highly dependent on naming conventions, which may not reflect real software structures, and the application of these approaches is limited to reverse engineering process only. Some practical problems with file name-based approaches we have encountered include:

- Inconsistent file names. A subsystem of a project that we dealt with had various method of naming a file as a result of evolutions. Here are some examples. The name of a subsystem is rh, which stands for request handler. There are some examples for the prefix of the file names: rh, RH, Crh, CRH, o_rh, s_rh, t_rh, and DF_rh. However, the prefixes represent different concepts. Some show releases, some subsystems, some objects, and some other information. We simply could not recover the overall design or architecture based on the file names for this project.
- Joint venture projects or projects across teams. Joint venture projects between or among companies are common nowadays. Different projects may have different naming mechanisms. Another problem is that different projects may also consist of some legacy software which may not follow the same naming convention as the more recent releases.
- Limited applications. For some applications, grouping of files based on names could show a higher level of abstraction, which is consistent with the architecture. However, the result may be misleading, because the design may not be well partitioned or some file names may not convey semantic information in the first place (Lung, 1998). In addition, the results usually show similarity with the existing directory structure. There is not much added value in general to support restructuring, as presented in Section 3.4.

Another school of thought is known as concept lattice analysis (Snelting, 2000). Concept analysis uses a maximal association property based on functions and their attributes to build a concept lattice. The clustering algorithm is then applied to analyze the properties of the structure to partition them into clusters. Concept analysis conveys rich information, including not only what components are similar, as depicted in the dendrogram, but also how components are similar by showing their shared attributes. Concept analysis, however, in general may have a scalability problem and it may be difficult to partition the overlapped concepts between different clusters (Snelting, 2000).

There are many other papers that discuss software architecture recovery (Arnold, 1993; Kazman and Carriere, 1998). A comprehensive survey of these techniques is beyond the scope of this paper. A general observation of these approaches is that the applications often are restricted to either reverse engineering of source code or are based on the number of shared features. This paper presents several methods of making use of different types of data for various applications.

6. Summary and future work

This paper presented a clustering method and demonstrated how it can be applied to software partitioning, recovery, restructuring, and decoupling. The key value of this approach is that it can support a rapid and effective evaluation of a system based on the relationships between components and features, or component interdependencies at various levels of abstraction. System partitioning is usually performed by designers based on their experiences. The proposed method can help designers quickly obtain an outline of the architecture or design and provide an alternative view. More evaluations could then be conducted to identify potential problems early in the development process. The architecture recovered from the approach, if different from the current or existing design, could also force the designers to reason and compare the differences, and potentially restructure the system. We also advocated that scenarios be used individually as well as collectively, even for the clustering analysis. That way, the result may reveal more insights.

In short, clustering is a technique that can be used in a variety of ways in both forward and reverse engineering and different levels of abstraction. The clustering technique presented in this paper is simple, fast, and yet it produces similar results to that of other clustering methods. In this paper, we also demonstrated different ways of defining and populating the input matrix to feed into the clustering technique.

The result of the clustering has provided useful feedback. However, there are other practical constraints that may need to be considered for specific products. These constraints may include development responsibilities, geographical issues, personal/political concerns, physical resources, or reusable components. Nevertheless, the approach has merit if consistently applied for refinement (Heyliger, 1994), since the allocation of functionality to components is crucial in architecture design (Kazman et al., 1994; Monroe et al., 1997). The approach presented in this paper is also easy to apply if the components and their interconnections are modified.

Some other areas are still in progress. In this research, coupling and dependency are treated the same way. In reality, they are different. One area which still requires research is identifying different types of interconnections and evaluating their impacts on the clustering. Another area is to develop more effective metrics to facilitate comparisons among design alternatives. The metric system strength, adopted from Heyliger (1994), does not scale well. When the approach is applied to the class or function level, or if the number of components is large, the metric values will become very small, as shown in Table 5. The integration of the clustering method with other tools will provide valuable information. The clustering method has been integrated with a visualization tool, which can display the architecture at various levels. We are also working on tools integration to support other aspects. Tools that allow the user to select a view and generate it accordingly will have a great deal of value. Another area which would benefit from further research is comparing various clustering techniques, including normalization of data. The numeric values, which show the number of interconnections among components, are obtained from a static analysis of the source code. With normalization, dynamic data may provide other useful information.

To apply the clustering method to reduce program code complexity, a tool is needed to automatically parse the source code to identify the relationship between input and output variables. In addition, experiments are required in order to clearly determine the relationship between input and output variables. Examples of more complex situations include logical statements and loops, parameter types identification, pointers, data structures like arrays, and records or structures. For object-oriented programs, more factors need to be considered.

Acknowledgements

Much of the research effort presented in this paper has been carried out while the authors were affiliated with Nortel Networks. We are grateful to Kalai Kalaichelvan and Rama Munikoti for their support. We would also like to express our gratitude to the reviewers and editors for their time and helpful comments.

References

Alexander, C., 1964. Notes on the Synthesis of Form. Harvard University Press, Cambridge, MA.

Anderberg, M.R., 1973. Cluster Analysis for Applications. Academic Press, NY.

Andreu, R.C., Madnick, S.E., 1977. A systematic approach to the design of complex systems: application to DBMS design and evaluation. Technical Report (CISR 32), MIT Sloan School of Management.

Anquetil, N., Lethbridge, T., 1998. Extracting concepts from file names: a new file clustering criterion. In: Proceedings of International Conference on Software Engineering, pp. 84–93.

Anquetil, N., Lethbridge, T., 2000. Experiments with clustering as a software remodularization method. In: Proceedings of the 6th Working Conf on Reverse Engineering, pp. 235–255.

Arnold, R.S., 1993. Software Reengineering. IEEE Computer Society Press, Los Alamitos, CA.

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

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

Briand, L., Devanbu, P., Melo, W., 1997. An investigation into coupling measures in C++. In: Proceedings of International Conference on Software Engineering, pp. 412–421.

Card, D.N., Glass, R.L., 1990. Measuring Software Design Quality. Prentice-Hall, Eaglewood Cliffs, NJ.

Davey, J., Burd, E., 2000. Evaluating the suitability of data clustering for software remodularization. In: Proceedings of the 7th Working Conference on Reverse Engineering, pp. 268–276.

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

Dromey, R.G., 1996. Cornering the Chimera. IEEE Software 13 (1), 33–43.

Everitt, B., 1980. Cluster Analysis. Heinemann Educational Books, Ltd., London.

Heyliger, G., 1994. Coupling, In: Marciniak, J. (Ed.), Encyclopedia of Software Engineering.

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

Kazman, R., Bass, L., Abowd, G., Webb, M., 1994. SAAM: a method for analyzing the properties of software architecture. In: Proceedings of the International Conference on Software Engineering, pp. 81–90.

Kazman, R., Carriere, S.J., 1998. View extraction and view fusion in architectural understanding. In: Proceedings of the 5th International Conference on Software Reuse.

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

Lung, C.-H., Cochran, J.K., Mackulak, G.T., Urban, J.E., 1994. Computer simulation software reuse by generic/specific domain modeling approach. Int. J. Software Eng. Knowl. Eng. 4 (1), 81–102.

Lung, C.-H., 1998. Software architecture recovery and restructuring through clustering techniques. In: Proceedings of the 3rd International Workshop on Software Architecture, pp. 101–104.

Lung, C.-H., Kalaichelvan, K., 2000. An approach to quantitative software architecture sensitivity analysis. Int. J. Software Eng. Knowl. Eng. 10 (1), 97–114.

Lung, C.-H., Zaman, Z., in preparation. Program restructuring and decoupling through clustering.

Mackulak, G.T., Cochran, J.K., 1990. Generic/specific modeling: an improvement to CIM simulation techniques. In: Shunk, D.L. (Ed.), Optimization of Manufacturing Systems Design. Elsevier Science Publishers, North-Holland, pp. 331–363.

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 International Workshop on Program Comprehension.

Monroe, R., Kompanek, A., Melton, R., Garlan, D., 1997. Architectural styles, design patterns, and objects. IEEE Software 14 (1), 43–52.

Murphy, G.C., Notkin, D., Sullivan, K.J., 2001. Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Software Eng. 27 (4), 364–380.

Neighbors, J., 1996. Finding reusable software components in large systems. In: Proceedings of Working Conference on Reverse Engineering, pp. 2–10.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. Software Eng. Notes 17 (4), 40–52.

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

Sartipi, K., Kontogiannis, K., 2001. Component clustering based on maximal association. In: Proceedings of the 8th Working Conference on Reverse Engineering, pp. 103–114.

Schwanke, R.W., 1991. An intelligent tool for reengineering software modularity. In: Proceedings of International Conference on Software Engineering, pp. 83–92.

Selby, R.W., Reimer, R.M., 1995. Interconnectivity analysis for large software systems, Technical Report, (UCIrv-95-PROC-CSS-001), University of California at Irvine.

Snelting, G., 2000. Software reengineering based on concept analysis. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 1–8.

Wiggerts, T.A., 1997. Using clustering algorithms in legacy systems modularization. In: Proceedings of the 4th Working Conference on Reverse Engineering, pp. 33–43.