Component Clustering Based on Maximal Association *

Kamran Sartipi[1] Kostas Kontogiannis[2]

University of Waterloo
Dept. of Computer Science[1] and,
Dept. of Electrical & Computer Engineering[2]
Waterloo, ON. N2L 3G1, Canada
{*ksartipi, kostas*}*@swen.uwaterloo.ca*

Abstract

In this paper, we present a supervised clustering framework for recovering the architecture of a software system. The technique measures the association between the system components (such as files) in terms of data and control flow dependencies among the groups of highly related entities that are scattered throughout the components. The application of data mining techniques allows to extract the maximum association among the groups of entities. This association is used as a measure of closeness among the system files in order to collect them into subsystems using an optimization clustering technique. A two-phase supervised clustering process is applied to incrementally generate the clusters and control the quality of the system decomposition. In order to address the complexity issues, the whole clustering space is decomposed into sub-spaces based on the association property. At each iteration, the sub-spaces are analyzed to determine the most eligible sub-space for the next cluster, which is then followed by an optimization search to generate a new cluster.

1 Introduction

With the increase of size and complexity of the software systems, the role of software architecture recovery as a prelude for most software maintenance tasks is increasingly important. The approaches to architectural recovery usually employ a clustering technique based on the properties of coupling and cohesion or association, in order to partition the software system into cohesive subsystems.

In this paper, we present a supervised (user-assisted) clustering technique, as a framework for architectural recovery, and define two similarity metrics based on the *association* property in highly related groups of system entities.

In this context, the software system is represented as a graph, where the system entities are denoted as nodes and data/control dependencies are denoted as edges. The application of data mining techniques on this graph reveals the groups of highly related entities.

The level of dependency among the groups of system entities is the basis for extracting association values between system components such as files. In this respect, the *component association* is defined as the degree to which, the entities in one component are related to the entities in another component. We define a new similarity measure between two system components in terms of the component association values and use it for the clustering algorithm.

The proposed supervised clustering technique, iteratively generates the subsystems of components (files) using the similarity measure between the system components and based on a two-phase optimization clustering process. The search space for the whole clusters are decomposed into a number of sub-spaces to handle the search complexity. At each iteration of the clustering process, a selection algorithm is used to select a sub-space for the next cluster, and an optimization search algorithm is used to collect the highly similar components around the core component of the selected sub-space, known as *main-seed.*

In a nutshell, this paper first provides a new similarity metric based on maximal association property (maximum number of shared properties) between two groups of entities such as files; next discusses a supervised clustering technique for decomposing a large system of files into cohesive subsystems; and finally discusses a search space reduction technique to manage the search complexity. The results of experimenting with two systems are also presented.

We implemented a prototype reverse engineering tool, Alborz [15], to recover the architecture of a software system as cohesive components. Depending on the user exper-

*This work was funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto) and the National Research Council of Canada.

1095-1350/01 /$10.00 © 2001 IEEE

tise and knowledge about the system, the user interaction can range from a few steps of guidance to the clustering algorithm, up to determining a whole cluster. The tool represents the result of the clustering as the *subsystems and interconnections* representation using both HTML pages to browse and analyze the quality of the result, and different graphs to visualize and investigate the graph topological properties.

2 Background

Cluster analysis is defined as: *the process of classifying entities into subsets that have meaning in the context of a particular problem* [6]. The clustering techniques are designed to extract an existing cluster structure of entities. However, the choice of technique affects the detected clusters which may or may not be the existing structure. In the following, we briefly discuss the issues pertaining to clustering.

Clustering algorithms

Wiggerts [25], Anquetil [2], and Tzerpos [23] have surveyed different aspects of clustering algorithms for software systems. Important clustering algorithms that apply to the field of software reverse engineering can be categorized as: i) *hierarchical algorithms*, where each entity is first placed in a separate cluster and then gradually the clusters are merged into larger and larger clusters until all entities are in a single cluster; ii) *optimization algorithms*, where a partitioning of the whole system is considered and with iterative entity movements between partitions the partitions are improved to an optimal partition; and iii) *graph-theoretic algorithms*, where an entity relationship graph of the system is considered and the algorithm searches to find subgraphs with special properties such as maximal connected subgraphs or minimal spanning trees. A *supervised* clustering process requires guides from the user in different stages to perform the clustering, whereas, an *unsupervised* clustering only relies on the similarity matrix consisting of the similarities of every pair of entities [6].

Similarity metrics

A *similarity* metric is defined so that two entities that are alike possess a higher similarity value than two entities that are not alike. Different methods for similarity measure fall into two general categories:

The first category is based on *relationships between the entities* in the form of function call, where the similarity is intuitively measured based on the static occurrences of the calls between the functions. The second category is based on shared properties (called *feature*) between two entities. Patel provides an interesting social relation analogy between "finding similar entities to an entity" and "finding the friends of a person in a party" [12].

There are a number of similarity measures proposed in the literature and Wiggerts provides a summary of different categories namely *association coefficients, correlation coefficients*, and *probabilistic measures* [25]. An evaluation of these similarity metrics can be found in [4]. We are interested in the *association coefficient* based similarity metrics.

In the related literature, there are many proposals for measuring similarity (or distance) between the entities based on single or multiple criteria including: i) data binding of functions via global variables [5]; ii) sharing a single feature [8] or a vector of features [12]; and inter-/intra-connectivity relations among modules [10]. In this connection, our proposed similarity metric is based on multiple features such as function, type, and variable to extract association, hence, it is more general than single criterion metrics.

Related work

In [10] a partitioning method is used to partition a group of system files into a number of clusters. A hill-climbing search considers different alternatives based on neighboring partitions, where the initial partition is randomly selected. In comparison, our method carefully finds a collection of rather separated and highly qualified sub-spaces that can be viewed as an initial partition of clusters, and then a search algorithm selects a sub-optimal group of files for each cluster. Therefore, the chance of trapping in a local optimum caused by random partitioning and hill-climbing, is eliminated.

In [24], a number of system structural properties are used as evidences to cluster the system files into a hierarchy of clusters. The method uses subgraph dominator nodes to find subsystems of almost 20 members, and builds up the hierarchy of subsystems accordingly. To simplify the computation, the interactions of more than 20 links to/from a file are disregarded. In contrast, our technique does not assume any pre-existing structure for the system such as directory structure, instead relies on an overall data/control flow dependencies among the system entities to be used for clustering.

Our work also relates to approaches on *concept lattice analysis* in using maximal association property [9]. A concept is a group of entities with maximal shared properties. In these techniques, a matrix of functions and their attributes (i.e., called/used functions, types, and variables) is built, from which a concept lattice is generated. The clustering algorithm then tries to analyze the structural properties of the concepts to partition them into clusters, where a cluster is a small set of concepts with large overlapped attributes. In contrast, we encode the maximal association property of a concept into a similarity metric and use it in a clustering technique.

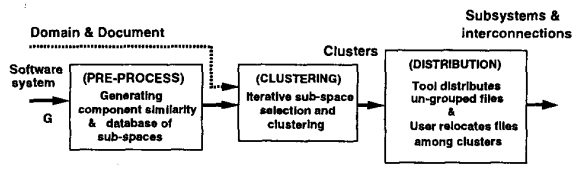


Figure 1. Supervised clustering framework.

Relation to our previous work

In our initial work on the development of a framework for software architecture recovery [18], we presented a new environment for applying data mining techniques on reverse engineering. Moreover, we used a query language to define abstract modules and interconnections and a search algorithm based on a commonly used score function (Jaccard) to instantiate the abstract modules and interconnections. The original environment has been formalized in terms of graph pattern matching technique which has been presented in [17]. Following the development of an architectural recovery system, in [16] we presented a metric to be used for evaluation of a system in terms of the modularity quality and design properties of the system. Specific characteristics of a developed prototype tool-set have been discussed in a short paper [15].

In the current paper, we present an incremental clustering technique based on a new similarity metric, as well as its application on two systems. In summary, the contributions of this paper include: i) proposal of a two stage supervised incremental clustering environment based on main-seed selection and optimization search; ii) definition of two new similarity measures, namely "entity-association" and "mutual component-association" for clustering at two levels of abstraction based on maximal association property; iii) comparison of two techniques for extracting the "maximal association" property among the system entities; and iv) use of a bounded queue heuristic for the branch and bound search algorithm to control the complexity of the search process.

In the following sections, we discuss the notion of maximal association between two entities, and then extend it to define the notion of association between two components.

3 Framework for supervised clustering

The proposed approach to supervised clustering consists of three phases (Figure 1).

In the first phase, *pre-process*, the software system is parsed and presented as a graph[1] with nodes as source code entities (i.e., *file, function, type*, and *variable*), and edges as data and control flow dependencies (i.e., *call, define, set, update*, and *declare*). The low-level relations between entities are aggregated into more abstract relations (i.e., *call* and *use*) to be used for the architectural level of the system and are represented as the source model graph G. Using a data mining algorithm, the entities in graph G are grouped based on the association property, and the result is represented as a collection of domains, denoted as $D(G)$, where each domain corresponds to a system entity. Based on the entity domains in $D(G)$ the similarity metrics between files are determined. Also the group of system files (whole search space) is decomposed into a collection of sub-spaces to be used for the clustering algorithm.

In the second phase, *clustering*, two distinct operations are performed for generating each subsequent cluster: i) a sub-space selection algorithm analyzes the collection of sub-spaces to provide a ranking list of the qualified sub-spaces for the next cluster, and the user selects a sub-space from this list; and ii) an optimization clustering algorithm searches to find the similar files in the selected sub-space to put them in a cluster. The result is shown to the user and if required the next iteration is performed.

In the third phase, *distribution*, the role of the user increases. The user can: i) allow the tool to distribute a portion of the ungrouped files among the clusters based on the closeness values of individual files to particular clusters; or ii) selectively relocate the files between the clusters based on cluster quality or size consideration.

4 Maximal association property

Most clustering techniques attempt to detect highly cohesive components, hence, producing a modular clustered system [10, 12, 11, 8, 9, 3]. In a software system consisting of modules, *cohesion* is a measure of the *"relative functional strength of a module"* [13]. A number of authors view the cohesion as *"coherence"* [11] or *"intra-connectivity"* [10] which is in fact a form of *external* property of a function as opposed to internal properties extracted by the slicing methods [3]. The proposals in this group consider a number of shared features for each function and determine the cohesion as the degree of sharing different sets of features such as: global variables, function calls, or data types [9, 7]. This property is referred to as *association coefficient* in clustering literature, and is considered as the most suitable property to detect two similar entities [4]. Two different techniques that detect *maximal association* (i.e., maximum number of shared features) among the groups of entities in a software system belong to *mathematical concept analysis* and *data mining* domains. In the following subsection, we compare these two techniques using the same example for both techniques, shown in Figures 2 and 3.

[1] The adopted graph formalism has been discussed in [16].

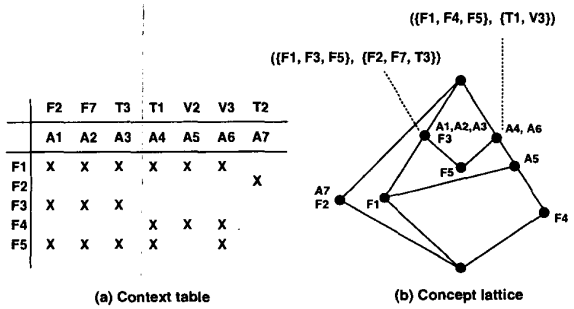


Figure 2. Application of concept lattice analysis in extracting maximal association.

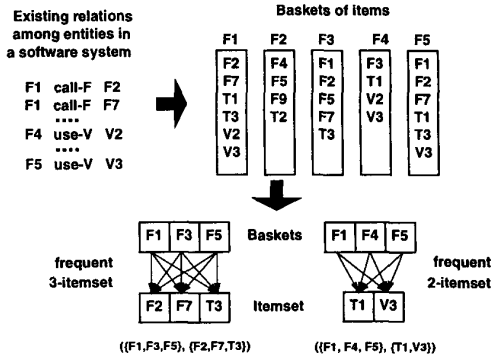


Figure 3. Application of data mining in extracting maximal association, in [18, 19].

4.1 Mathematical concept analysis

Recently, the application of mathematical concept analysis in reverse engineering has been investigated [21, 9]. In this formalism, a binary relation between *objects*[2] and their *attributes* is represented as a lattice which provides significant insight into the structure of the relation. A *concept* is a maximal collection of objects all sharing a maximal group of attributes. In a software system, the *objects* are functions, the *attributes* are functions, types, and variables, and the *binary relation* is a data or control flow dependency. The main idea is then to partition the lattice so that small sets of concepts with overlapped attributes are detected. The difficulty in partitioning lies in resolving the overlapped concepts among different clusters which has resulted in unsuccessful approaches [22].

Figure 2(a) illustrates the table representation (*context table*) of a relation between five functions (F1-F5) and seven attributes (A1-A7), where , the relation represents an aggregation of function call and data-type/variable use. Every two rows (or two columns) can be swapped without change in the concepts. In this table, a concept corresponds to a maximal rectangle consisting of particular rows and arrows. Figure 2(b) illustrates the concept lattice of the context table in part (a). Each node of the lattice corresponds to a concept. A concept lattice has the following characteristics:

- Each lattice node (i.e., a concept) is labeled with objects (functions) and attributes, except for the top and bottom nodes that may be unlabeled.
- Every object has all attributes that are above it in the lattice (directly above or separated by some links).
- Every attribute exists in all objects that are below it in the lattice (directly below or separated by some links).

[2]Here, the notion of *object* is different from object in object oriented paradigm.

For example, the node labeled $|^{A1,A2,A3}_{F3}$ in the lattice corresponds to the concept $(\{F1, F3, F5\}, \{A1, A2, A3\})$ which means each of the functions $F1, F3, F5$ has all attributes $A1, A2, A3$. Also the node labeled "$A4, A6$" in the lattice corresponds to the concept $(\{F1, F4, F5\}, \{A4, A6\})$ with similar interpretation. Such interesting properties are not easily observable in the context table of a large software system.

An alternative technique from data mining domain (below) also detects the maximal association among entities.

4.2 Data mining

The application of data mining in reverse engineering has already been discussed in our early work [18, 19]. In this subsection, we repeat some definitions from [18] to enable us to compare the advantage of data mining technique over concept lattice technique in producing a similarity metric based on the maximal association property.

Most data mining algorithms operate on a database of market transactions in the form of market baskets and their contained items, and generate a collection of *frequent i-itemsets* ($i \in \{1..k\}$). A frequent itemset has the same interpretation as a concept in the concept lattice analysis. Figure 3 illustrates the process of generating frequent itemsets from the database of system entities and relationships, representing the context table in Figure 2(a). The resulting frequent itemsets $(\{F1, F3, F5\}, \{F2, F7, T3\})$ and $(\{F1, F4, F5\}, \{T1, V3\})$ are in the form of tuple of sets $(\{baskets\}, \{itemsets\})$. These two frequent itemsets are the same as two concepts specified in Figure 2.

The frequent itemsets are generated using the *Apriori algorithm* [1]. The input to the Apriori algorithm is the set of entities and relationships in the software system, as shown in Figure 3. The Apriori algorithm generates all possible

frequent itemsets, so that the number of baskets in each frequent itemset is not less than a user-defined threshold *min-baskets*. The generated frequent itemsets are categorized into large groups, based on the size i of the itemset ($i \in \{1..k\}$). Below, three frequent 5-itemsets (5 items in each itemset) from an experiment with a system are shown:

```
({F396, F403, F816}, {F399, T27, V196, V264, V298})
({F774, F804, F807}, {F397, T27, V259, V312, V361})
({F774, F800, F807}, {F407, F608, T5, V259, V361})
```

The prefixes F, T, V correspond to the function, aggregate type, and global variable, respectively.

The Apriori data mining technique has some advantages in generating groups of entities with maximal associations in comparison with the concept analysis technique:

- The quantity of the generated frequent itemsets is controlled by the user-defined *min-baskets* value. As a simple guidance, the user can start from the *min-baskets* value of 2 and increase it so that for a middle-size system (≈50 KLOC), the maximum size of the generated itemsets is ≈10, and the total number of the generated frequent itemsets is ≈20K.
- The resulting frequent itemsets provide a listing of all maximal associated groups which facilitate the encoding of association structure of the groups into a similarly measure, whereas this information is scattered throughout the neighboring nodes in a concept lattice.

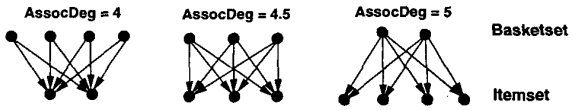
In the following subsection, we introduce a new similarity metric (namely *ent-assoc*) between two system entities such as functions, aggregate types, or global variables to be used for clustering[3]. However, the focus of this paper is component clustering, and we use *ent-assoc* to define the similarity measure between two components.

4.3 Similarity measure between two entities

Association in a graph is a property "*assoc*" among two or more source nodes that share one or more sink nodes (through graph edges). In analogy with data mining terminology, we refer to the sink nodes as "*itemset*" and the source nodes as "*basketset*". In this sense, the whole group of itemset and basketset are denoted as an *associated group*. The association $assoc(e_i, e_j, g_x)$ between two entities e_i and e_j in an associated group g_x is an undirected relation which is defined as:

$$assoc(e_i, e_j, g_x) = |itemset(g_x)| + \frac{|basketset(g_x)|}{2}$$

[3]We have compared the properties of this metric with the Jaccard metric and have used it to cluster the system entities into cohesive modules. However, the space limitation prevents us to discuss them in this paper.



Associated groups with the same number of entities

Figure 4. The notion of entity association.

Where, the value or degree of association between e_i and e_j is a positive *real* number. In general, the number of shared entities (itemsets) contributes more on closeness of the entities in an associated group than the number of sharing entities (baskets). This is also implied from the discussion on cohesion in section 4. In Figure 4, three associated groups with the same number of entities but different association degrees (*assocDeg*) are shown. The properties of such association measure are as follows:

- It is meant for identifying the members of a group of highly associated entities in a system.
- It is not normalized, i.e., its value is not restricted between 0 and 1, instead it depends on the size and form of the group of entities.
- It considers the data types and variables as members of a group with functions, as opposed to considering them as attributes of functions which cause only the functions to be grouped.

The similarity between two entities e_i and e_j, denoted as $ent\text{-}assoc(e_i, e_j)$, is defined as the *maximum association degree* between e_i and e_j, considering that e_i and e_j may belong to more than one associated group g_x with different association degree in each g_x. Formally:

$$ent\text{-}assoc(e_i, e_j) = max|_{g_x}\ assoc(e_i, e_j, g_x)$$

Domain of a node

We represent a software system as a graph $G = (N, R)$, where $N = \{n_1, n_2, ..., n_n\}$ is the set of system entities and $R = \{r_1, r_2, ..., r_m\}$ is the set of relationships between entities. The *domain* of a node n_j in graph G, denoted as D_j, is defined as: *a collection of the graph nodes n_d's that are associated with node n_j along with their similarity values $ent\text{-}assoc(n_j, n_d)$ with respect to n_j*. The node n_j is called the *main-node* of the domain D_j. Formally:

$$D_j = \{(n_j, n_d, a) \mid n_j, n_d \in N \wedge a = ent\text{-}assoc(n_j, n_d)\}$$

Where, the node n_d satisfies the association property $assoc$ with respect to n_j. The entities in D_j are also ordered according to higher similarity values with respect to n_j. The *domain database* of the graph $G = (N, R)$, denoted as $D(G)$, is a database of all graph node domains D_j.

4.4 Similarity measure between two components

The notion of *component association* between two components C_i and C_j, *comp-assoc*(C_i, C_j), has been used to define a software evaluation model based on the component association views [16]. In the current paper, we use the notion of component association to define a new similarity metric, namely *mutual-assoc*(C_i, C_j), between two components. For the continuity of the discussion, the definition of *comp-assoc*(C_i, C_j) has been repeated here.

Component

A system component is a named grouping of the system entities, such as functions, types, and variables, with the relation *contains* or *defines* to those entities. A component can also be viewed as a composite entity. Each system entity is contained in only one component. Each user defined entity in an *include file* (i.e., global variable or aggregate type) is contained in one component based on the frequency of usage by the functions in that component. A component can be a *file*, a *module* of entities, or a *subsystem* of files where the files are replaced by their contained entities.

Component association

A component C_i represents a subgraph $G^{c_i} = (N^{c_i}, R^{c_i})$ of the system graph $G = (N, R)$. The *domain of a component* C_i, denoted as D^{c_i}, is *a collection of the system entities that exist in the domain of each entity* n_j, *where* n_j *is contained in component* C_i, Formally:

$$D^{c_i} = \{(n_j, n_d, a) \in D(G) \mid n_j \in N^{c_i} \wedge n_d \in N\}$$

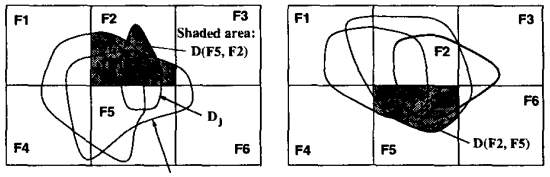
In our discussion, we use *file* as component and *function* as entity, where each file contains (defines) a number of functions. In Figure 5(a), the domain of each entity (function) in file $F5$ is shown as the area in a closed curve. The domain of file $F5$ (i.e., D^{c_5}) is represented as the whole area covered by all closed curves.

The *component association* of component C_i onto component C_j, denoted as *comp-assoc*(C_i, C_j), is defined as the degree of dependency of the entities in C_i onto the entities in C_j. Formally:

$$\textit{comp-assoc}(C_i, C_j) = \frac{\sum_{k=1}^{|D(C_i,C_j)|} totalAssoc(e_k)}{|C_j|}$$

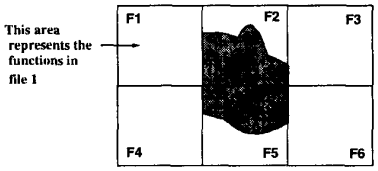
Where, $D(C_i, C_j)$ is the set of entities in the intersection of the "domain of component C_i" (i.e., D^{c_i}) and "component C_j"; $totalAssoc(e_k)$ is the sum of all similarity values for the k^{th} entity in $D(C_i, C_j)$[4] with respect to entities in C_i; and $|C_j|$ is the number of entities in component C_j.

[4]Note that each node in $D(C_i, C_j)$ has different association degrees, if it is a member of different overlapped domains, as in Figure 5(a).



Each curve is a domain of a function in fle 5.

(a) Component association of file 5 to file 2. (b) Component association of file 2 to file 5.



(c) Mutual component association of file 5 and file 2.

Figure 5. A system of six files representing the component association and mutual association of files 2 and 5.

Therefore, *comp-assoc*(C_i, C_j) is the *average of the total similarity values of an entity in* $D(C_i, C_j)$. In Figure 5(a) the overlapped part $D(F_5, F_2)$ is shown.

Similarity measure

Having defined the directed relation *comp-assoc*(C_i, C_j), we define the similarity measure between two components C_i and C_j as the *mutual association* of components C_i and C_j, denoted as *mutual-assoc*(C_i, C_j). The mutual association is defined as a weighted average of the component associations between those two components, as:

$$\textit{mutual-assoc}(C_i, C_j) = \frac{K_{max} * \textit{comp-assoc}(C_i, C_j) + K_{min} * \textit{comp- assoc}(C_j, C_i)}{K_{max} + K_{min}}$$

Where, K_{max} and K_{min} are the weights for the larger and smaller component association degrees, respectively. We use $K_{max} = 2$ and $K_{min} = 1$ to give more value to the larger component association. The unit for both *comp-assoc*(C_i, C_j) and *mutual-assoc*(C_i, C_j) is *"association-degree per entity"* (abbreviated as APE). Figure 5 illustrates the notions of *component association* and *mutual component association* between two files 2 and 5.

5 Supervised clustering scenario

We propose a supervised clustering technique based on *main seeds* that correspond to a decomposition of the whole search space into sub-spaces. The clustering process is *incremental* and consists of n iterations, each corresponds to

generating a new cluster using an optimization search algorithm. The *incremental* characteristic of the technique does not give privilege to the earlier generated clusters: i) there is no commitment on the early clusters until the whole set of clusters is generated; and ii) each succeeding cluster is allowed to overlap with the previous clusters. Therefore, each cluster has an equal chance to possess a given file in the system. However, based on the previous clusters, a *main-seed selection* algorithm provides a ranking list of *top-m* sub-spaces to be selected for the next clustering iteration.

The role of the user in the clustering process is flexible. The minimum interaction with the tool is to select the most appropriate sub-space for the next clustering iteration from the ranking list of the sub-spaces. The role of the user increases by defining fixed files (known as *seeds*) in the clusters which imposes the algorithm to restrict the search space. Finally, the user may completely bypass one iteration by defining a whole cluster as a *manual cluster*. Figure 6 demonstrates a two-phase scenario for the clustering process as following:

In the *clustering phase* (*loop-1* in Figure 6), the system of files are grouped into subsystems based on the similarity metric *mutual-assoc* (C_i, C_j). This phase is further discussed in the next section.

In the *distribution* phase (*loop-2* in Figure 6), the user controls the clustering process. In this phase the tool examines all those files that have not been yet assigned to any cluster (known as *rest-of-system*), against the existing clusters. The result is a list of the files in the rest-of-system and their overall closeness to one or more clusters, which is shown to the user. The list is ordered according to the higher closeness values so that the user can easily view and choose a group of files from the top of the list and then the tool distributes them among the clusters. After each distribution operation, a new list of *rest-of-system* is generated for inspection. The user stops this distribution once the files in the *rest-of-system* show trivial closeness to any of the clusters. At this point the *rest-of-system* itself constitutes a separate manual cluster. The last operation in distribution phase is the *relocation* of files among the clusters. This operation allows the user to overrule the tool's decision on clustering, in order to adjust the sizes of the clusters, improve the average closeness of the clusters, or balance the *import/export* of entities among the clusters. The user may select the files to relocate according to the knowledge obtained from the system documents or from a low overall closeness of one or more files in a cluster to the other files in that cluster.

The result of each iteration, along with additional metrics about the properties of the results, are presented to the user in the forms of HTML pages and graph representations to be viewed by graph visualization tools (block 4 in Figure 6).

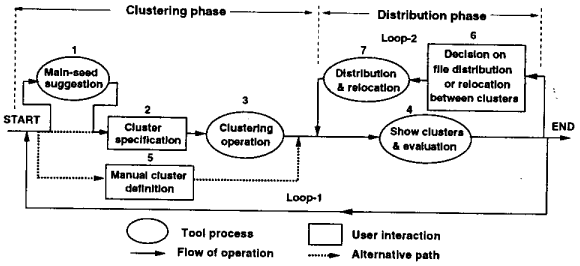


Figure 6. Scenario for supervised incremental clustering.

6 Clustering phase

In order to address the search complexity of clustering components into subsystems, we restrict the search space for each cluster by decomposing the whole search space into sub-spaces and collect them into a space database $D^f(G)$.

The generation of sub-spaces in $D^f(G)$ is similar to the generation of domains in $D(G)$, as discussed in section 4.3. The sub-spaces are generated by applying the Apriori algorithm on the database of baskets and items in Figure 3, where the baskets are files and the items are functions, types, and variables that are called/used by the functions inside the corresponding files.

Each sub-space consists of loosely associated files which qualifies them as a smaller search space for the clustering algorithm. The characteristics of the sub-spaces in $D^f(G)$ are as follows:

- The neighboring sub-spaces in $D^f(G)$ are highly overlapped, therefore, finding an optimal set of sub-spaces as the basis for the clusters requires further analysis of the sub-spaces.

- The files in a sub-space are already ranked according to their association with the main-node of the sub-space. This allows us to select the highly associated files by simply truncating a sub-space to a certain size k. A *truncated sub-space* refers to the first k files of that sub-space and is denoted as *"t-space"*. We truncate all sub-spaces to a particular size k and represent them as a group of overlapping subsystems whose sizes are at most k. However, a truncated sub-space only guarantees the high association of each file to the main-node not between two non main-node files.

The *t-spaces* are analyzed in order to find qualified sub-spaces for the clustering operation.

6.1 Main-seed selection

The *main-seed selection* algorithm resembles the actual clustering algorithm, assuming that a *t-space* is an actual cluster (subsystem). This assumption is not unrealistic since the files in sub-spaces are already ranked according to their association values with the sub-space's main-node. This allows us to analyze individual sub-spaces in the $D^f(G)$ according to the criteria which are formulated as a *score function*. The score function assigns a single score to each sub-space and provides a ranking of *top-m* sub-spaces to the user to select from. The selection algorithm computes *t-spaces* which are large, highly correlated, and sufficiently distinct using the following score function:

$$score_{t\text{-}space} = (AvgAssoc * k1) + (Size * k2) - (Overlap * k3)$$

Where, *AvgAssoc* and *size* are the average association degrees and size of the *t-space*; and *overlap* is the ratio of the total similarity degrees of the shared files[5], to the total similarity degrees of the files in the *t-space*. This ratio demonstrates the degree of overlap between the already clustered subsystems and the candidate *t-spaces*. The first *t-space* is freely selected since the overlap ratio is zero. However, at each subsequent iteration the overlap ratio increases. The coefficients $K1$ to $K3$ are empirically defined by the user so that the terms of the score function yield almost comparable numbers on average. These coefficients are adjustable rather independently.

The seed-selection algorithm scans the whole database $D^f(G)$ m times to produce the *top-m* main-seeds for the user to select from.

6.2 Clustering operation

We use a form of *branch and bound* search algorithm with *bounded path queue* which produces a sub-optimal solution at each clustering iteration, Figure 7(a).

In a branch and bound algorithm a search tree with *incomplete paths* is built and the paths are stored in a sorted path queue. At each step, the algorithm expands an incomplete path with the highest score from the head of the path queue. Upon expansion, new incomplete paths are generated, added to the previous paths in the queue, and the queue is sorted.

The procedure continues until a *complete path* which is an optimal solution is found. A valuation function allocates a score to each node of the branch and bound search tree to guide the search process. This general approach in most cases restricts the search space to a small subset of all tree

[5] Shared files between the candidate *t-space* and the previously selected *t-spaces* or the accumulated clusters.

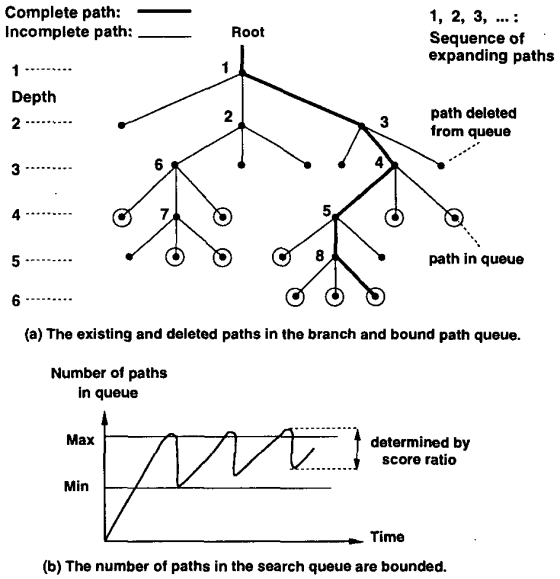


Figure 7. Branch and bound search tree with bounded path queue.

paths, preventing the exponential complexity inherent to the searching problems.

A new cluster is specified by its *main-seed* n_j which corresponds to a sub-space D_j from the database $D^f(G)$. The sub-space D_j determines all potential files that can be clustered in a subsystem. The search tree has a root which corresponds to the main-seed[6] of the subsystem and zero or more nodes that correspond to the *seeds* which are fixed files in a subsystem. At each depth of the tree, a new file is selected and added to the group of files in the subsystem. In Figure 7(a), a thick line from the root of the tree to a leaf node represents a complete path which produces an optimal solution.

Bounded path-queue heuristic

During the search process in a system with large sub-spaces, the branch and bound algorithm accumulates a large number of incomplete paths in the path queue which make the process of storing and sorting the paths in the queue as a bottleneck for the algorithm. Since the path queue is sorted, all of the eligible paths to be expanded are located toward the head of the queue with high scores. Therefore, most of the paths with low scores at the end of a large path queue will never get a chance to be expanded, and remain at the bottom of the path queue until the end of a successful

[6] Here, we assume that each subsystem has a single main-seed, however a subsystem can have more than one main-seed.

search. This property allows us to restrict the size of the path queue within a reasonable range (e.g., multiple hundreds of paths) and still get the same result as we had kept all the paths in the path queue. Figure 7(b) illustrates the oscillation of the number of paths in the path queue. Once the size of queue passes the maximum threshold, it is truncated to the minimum size. However, we only delete the paths from the bottom of the path queue whose scores are much less than those on the head of the queue. Therefore, when we collect paths whose scores are close to each other, the size of queue is kept around the maximum size, as shown in Figure 7(b). As a result this bounded path-queue heuristic yields a sub-optimal version of the branch and bound search algorithm in trade of increasing the performance. In practice, for a medium size system we use a (max, min) queue size threshold of (400, 200) with ratio $\frac{maxScore}{minScore} = 2.0$.

In Figure 7(a), an example of a sub-optimal search with the sequence of path expansion is shown. In this example, each incomplete path is expanded with three paths and the (max, min) threshold is (16, 10).

Score evaluation

We use *group average similarity*[7] $Gav(e, c)$, which determines the average closeness of an entity e to a group of entities in a cluster c. The branch and bound score function $score_{bb}(e,\ c)$ evaluates the score of a candidate entity e against the cluster c using the group average similarity Gav and an estimation of the remaining distance to the destination, as:

$$score_{bb}(e, c) = Gav(e, c) * (1 + k * \frac{|c|}{l})$$

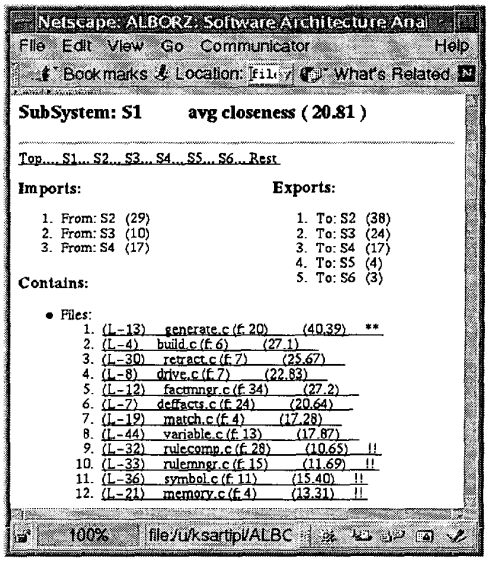
Where, k is an empirical coefficient, $|c|$ is the size of incomplete cluster, and l is the size of complete cluster.

The rational for such a score function is as follows: the value of the group average similarity Gav is decreasing by accumulating more entities in the cluster c. Therefore, the search algorithm tends to explore most of the short incomplete paths in the search tree (i.e., smaller clusters), hence, it is reduced to a breath first search algorithm. However, the second part of score function, i.e., $k * \frac{|c|}{l}$ favors the longer incomplete paths to expand. The combination of these two cases with a proper empirical coefficient k would give chance to the longer incomplete paths to be expanded.

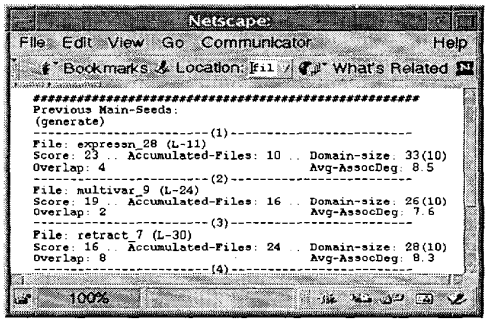
7 Experiments

In this section, the application of the proposed clustering technique on decomposing two software systems CLIPS 4.3 and Xfig.3.2.3 are presented. We have implemented a reverse engineering tool (Alborz [15]), as a user assistant, to recover the architecture of a software system as cohesive components (i.e., subsystems or modules). The Alborz tool has been built using the Refine re-engineering toolkit [14] and uses the built-in parsers to parse the software systems. The experiments have been performed on a Sun Ultra 10 platform.

[7] Jain provides the results of a number of studies on comparing different *group similarity* techniques [6].



(a) Result of recovery for first cluster (subsystem S1).



(b) The top-4 main-seeds to be selected for second cluster.

Figure 8. Main-seed selection for CLIPS.

7.1 CLIPS system

In the first experiment, we apply the proposed technique to recover the architecture of the CLIPS system. The CLIPS system provides an environment for building rule based expert systems. Our experimented results are evaluated against the CLIPS architecture manual [20] published by the NASA Software Technology Support Center. The CLIPS 4.3 consists of 40 KLOC, 46 source files, 736 func-

S1	S2	S3	S4
1) generate.c **	1) expressn.c **	4) math.c **	3) utility.c **
1) build.c	1) rulepars.c	1) evaluatn.c	x) textpro.c !!
3) retract.c	1) lhsparse.c	4) multivar.c	2) sysdep.c !!
3) drive.c	3) engine.c	4) sysio.c	
3) factmngr.c	1) commline.c !!	4) syssecnd.c	
x) deffacts.c	1) analysis.c !!	5) intrbrws.c	S5
3) match.c	1) scanner.c !!	5) entrexec.c	
1) variable.c	2) router.c !!	5) intrfile.c	6) object.c **
x) rulecomp.c !!	1) reorder.c !!	4) syspred.c !!	6) method.c
x) rulemngr.c !!		4) sysprime.c !!	6) bc.c !!
2) symbol.c !!			x) NeXTcall.c !!
2) memory.c !!	Main-seed: **		
	Distributed: !!		S6 (Manual)
CLIPS Sub-systems from document	1: Parsing modules: 11 files 2: System function modules: 4 files 3: Inference engine: 7 files 4: Rule manipulation: 6 files 5: Rules interface: 3 files 6: Object modules: 3 files x: Other modules or not-reported		x) main.c x) my-methods3.c x) compile.c x) methodsFile.c x) my_source3.c x) my_source4.c

(a) Recovered subsysems of the CLIPS system

Recovered subsystem	No. files	Related CLIPS subsystems	Prec.	Recall
S1	12	2 & 3	50%	54%
S2	9	1	78%	64%
S3	10	4 & 5	90%	100%
S5	4	6	75%	100%

(b) Accuracy of the recovered subsystems

Figure 9. The result of CLIPS decomposition.

tions, 161 global variables, and 54 aggregate types. We decompose the CLIPS system files into 6 subsystems in accordance with the supervised clustering scenario in section 5. The coefficient values for the CLIPS system include: *t-space* size = 10, $K1 = 0.5$, $K2 = 1.0$, $K3 = 10.0$ for $score_{t\text{-}space}$ function, and $k = 4.0$ for $score_{bb}$ function.

Figure 8(a) corresponds to the final result of the clustering scenario consisting of both analysis and distribution phases, where only the first cluster (subsystem S1) is shown. The subsystem S1 consists of 12 files each in a separate line, including a main-seed file *generate.c* (marked with "**") and 4 distributed files (marked with "!!"). Each file has a hypertext link to its source code, and is annotated with the number of functions in the file (e.g., f:20 means 20 functions), and the average similarity value with other files in the subsystem (e.g, 40.39 APE in the first line). The *import/export* parts illustrate a summary of the function interaction with other subsystems, as the *"subsystems and interconnections"* representation of the decomposed system. The user can switch to the detailed version of the subsystem interaction to view each individual entity with hypertext links to source code.

Figure 8(b) presents the score ranking of *top-4* subspaces and their corresponding main-seeds, generated by the main-seed selection algorithm after clustering subsystem S1. Each entry includes the information about a truncated sub-space (*t-space*) such as: average of association to the main-seed; size of both sub-space and *t-space*; number of overlapped entities; and its overall score. Therefore, the user obtains enough information about the eligible subspaces to select the best main-seed for the second clustering iteration.

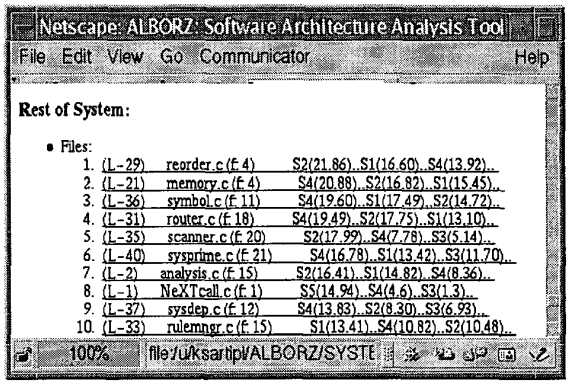


Figure 10. The *rest of system* files in CLIPS.

Figure 9 presents a detailed result of the CLIPS system decomposition, and a measure of accuracy based on *Precision* and *Recall* metrics, compared to the CLIPS system documents. The result of clustering is very promising.

Figure 10 demonstrates the files in the *rest-of-system* after the clustering phase of the clustering scenario. Each line corresponds to a file and is annotated with the closeness values to three different subsystems. The user can easily select a group of files which indicate high closeness values to some subsystem and let the tool distribute them among the clustered subsystems. The distribution is performed in sequence, that is after each allocation of a single file to a subsystem the closeness values are reassessed for the next file. The result of distribution is shown as files that are marked with *"!!"* in Figure 9. The overall computation time for both main-seed selection and incremental clustering in this experiment is 10 seconds

7.2 Xfig system

The system in the second experiment is the Xfig drawing tool which runs under X-Windows. The Xfig.3.2.3 consists of 75 KLOC of source code written in C, distributed over 100 source files, 75 include files, 1662 functions, 1356 global variables, and 37 aggregate types. According to the maintainers of the Xfig system, Xfig lacks any documentation on the structure or implementation, and only usage documents are provided. However, a consistent naming convention is used throughout the system files, such as: d_* files relate to drawing shapes; e_* files relate to edit-

S1: 24 files	S2: 21 files
e_movept.c (f: 27) (49.72) **	e_edit.c (f: 111) (92.28) **
e_joinsplit.c (f: 16) (48.92)	w_grid.c (f: 2) (55.93)
u_search.c (f: 31) (53.77)	w_export.c (f: 25) (47.8)
e_move.c (f: 4) (31.45)	w_print.c (f: 19) (41.57)
e_copy.c (f: 5) (32.50)	w_rulers.c (f: 35) (46.12)
e_rotate.c (f: 23) (39.88)	main.c (f: 2) (52.20)
e_convert.c (f: 8) (41.65)	w_indpanel.c (f: 106) (50.87)
u_list.c (f: 67) (60.76)	w_cmdpanel.c (f: 38) (41.39)
u_markers.c (f: 34) (41.52)	w_color.c (f: 59) (42.53)
u_redraw.c (f: 28) (15.95) !!	w_util.c (f: 51) (38.47)
f_util.c (f: 36) (21.14) !!	w_browse.c (f: 5) (33.56)
u_create.c (f: 26) (23.19) !!	w_fontpanel.c (f: 6) (32.57)
f_load.c (f: 4) (20.64) !!	w_mousefun.c (f: 27) (32.2)
u_undo.c (f: 37) (22.19) !!	w_library.c (f: 24) (32.15)
u_scale.c (f: 13) (21.48) !!	w_file.c (f: 24) (31.72)
e_compound.c (f: 6) (25.41) !!	w_srchrepl.c (f: 19) (29.91)
f_read.c (f: 29) (34.34) !!	w_msgpanel.c (f: 13) (27.66)
f_save.c (f: 14) (30.86) !!	w_modepanel.c (f: 52) (14.19) !!
d_subspline.c (f: 5) (26.75) !!	w_drawprim.c (f: 22) (18.98) !!
e_align.c (f: 21) (32.13) !!	w_dir.c (f: 18) (19.42) !!
e_arrow.c (f: 9) (28.56) !!	w_layers.c (f: 21) (24.61) !!
e_flip.c (f: 18) (34.43) !!	
u_bound.c (f: 11) (37.19) !!	
e_deletept.c (f: 4) (35.40) !!	

S3: 27 files	Rest of System: 26 files
u_elastic.c (f: 52) (126.83) **	f_picobj.c (f: 3) S1(14.76)..S3(13.16)..
d_box.c (f: 4) (48.12)	w_help.c (f: 7) S2(13.76)..S3(5.8)..
e_scale.c (f: 43) (101.95)	f_readjpg.c (f: 4) S3(13.49)..S1(11.44)..
u_drag.c (f: 30) (93.79)	f_readpcx.c (f: 5) S3(11.99)..S1(9.90)..
d_ellipse.c (f: 16) (62.49)	w_cursor.c (f: 5) S2(11.43)..S3(6.75)..
d_regpoly.c (f: 4) (48.48)	u_free.c (f: 13) S1(10.83)..S2(5.20)..
d_arcbox.c (f: 4) (48.13)	u_print.c (f: 6) S2(10.70)..S1(3.21)..
d_arc.c (f: 7) (50.38)	f_readgif.c (f: 5) S2(10.66)..S1(3.49)..
d_line.c (f: 8) (50.4)	w_setup.c (f: 1) S2(8.3)..S1(1.37)..
d_picobj.c (f: 4) (39.31)	f_readeps.c (f: 5) S1(7.45)..S3(6.33)..
u_draw.c (f: 38) (61.52)	f_readxbm.c (f: 4) S3(7.32)..S1(6.80)..
e_addpt.c (f: 12) (41.23)	mode.c (f: 4) S3(7.8)..S1(3.90)..
f_readold.c (f: 7) (40.56)	w_menuentry.c (f: 2) S1(6.55)..S3(6.27)
w_canvas.c (f: 17) (13.30) !!	f_neuclrtab.c (f: 17) S2(6.9)..S3(4.32)..
w_capture.c (f: 6) (15.26) !!	f_readtif.c (f: 1) S2(5.12)..S1(2.22)..
u_geom.c (f: 10) (18.24) !!	f_readppm.c (f: 1) S2(4.86)..S1(2.10)..
w_rottext.c (f: 18) (21.75) !!	u_fonts.c (f: 3) S2(4.81)..S3(1.61)..
u_translate.c (f: 12) (21.20) !!	u_pan.c (f: 5) S2(4.55)..S1(2.0)..
e_update.c (f: 22) (23.64) !!	f_wrpcx.c (f: 5) S2(4.3)..S3(1.32)..
w_zoom.c (f: 12) (25.48) !!	u_error.c (f: 5) S2(3.29)..S1(2.71)..
d_text.c (f: 24) (25.21) !!	object.c (f: 0) S1(0.0)..S2(0.0)..
f_readfigure.c (f: 1) (28.93) !!	resources.c (f: 0) S1(0.0)..S2(0.0)..
e_glue.c (f: 20) (31.6) !!	w_fontbits.c (f: 0) S1(0.0)..S2(0.0)..
e_break.c (f: 4) (29.71) !!	w_i18n.c (f: 0) S1(0.0)..S2(0.0)..
e_placelib.c (f: 10) (33.83) !!	w_icons.c (f: 0) S1(0.0)..S2(0.0)..
d_spline.c (f: 6) (33.7) !!	w_listwidget.c (f: 7) S1(0.0)..S2(0.0)..
e_delete.c (f: 8) (33.92) !!	

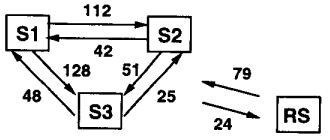


Figure 11. Decomposition of the *Xfig* drawing system and the function interaction diagram.

ing shapes; u_* files are utilities for drawing and editing shapes; f_* files have file-related functions; and w_* files have X-window calls in them to do all of the window-related functions.

These naming conventions provide a logical structure for the system which does not necessary relate to the cohesive clustering of the files into subsystems. Therefore, the generated clusters may overlap with two or more logical subsystems. For example, three groups of files *drawing* shapes, *editing* shapes, and their *utilities* are highly related through a large number of global variables, therefore, a low-coupled decomposition of these three groups is almost impossible.

Figure 11 presents the decomposition of the Xfig system into four equally sized subsystems. Each line corresponds to a file along with the number of functions and overall similarity to other files in the subsystem. Subsystems S1 corresponds to the files for editing shapes and their utilities with 80% Precision and 51% Recall. Subsystem S2 corresponds to the X-window related files with 90% Precision and 70% Recall. Subsystem S3 contains 90% of the drawing shapes; and the *rest of system* contains the files that present trivial closeness to either of the subsystems. A *subsystems and interconnections* diagram in Figure 11 illustrates the function interaction among the subsystems with high interaction between S1 and S3. The overall computation time for both main-seed selection and incremental clustering in this experiment is 50 seconds

8 Conclusion

In this paper, we defined two similarity metrics based on an extension to the conventional coupling and cohesion metrics. The idea is based on providing a metric for measuring the interaction and correlation among the system components at the architectural level of a system. In this approach, the software system is represented as a graph. The data mining technique *Apriori* extracts the maximum association measures among the system entities. A domain analysis technique measures the component associations which are then represented as a new similarity metric mutual component association between components. Experiments with two middle size systems and the accuracy evaluation using Precision and Recall indicate that the proposed technique provides promising results on decomposing a monolithic software system into highly cohesive components. Finally, the user incorporates the knowledge about the system domain and documents into the clustering process.

References

[1] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *Proceedings of the 20th Interna-*

tional Conference on Very Large Databases, Santiago, Chile, 1994.

[2] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, 1999.

[3] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[4] J. Davey and E. Burd. Evaluating the suitability of data clustering for software remodularisation. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 268–276, 2000.

[5] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

[6] A. K. Jain. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, N.J., 1988.

[7] R. Koschke. An incremental semi-automatic method for component recovery. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 256–267, October 1999.

[8] T. Kunz and J. P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527, June 1995.

[9] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, 1997.

[10] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of IWPC'98*, pages 45–53, Ischia, Italy, 1998.

[11] V. B. Misic. Coherence equals cohesion-or does it? In *Proceedings of the Seventh Conference on Asia-Pacific Software Engineering*, pages 465–469, December 2000.

[12] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *International Conference on Software Engineering*, pages 38–48, 1992.

[13] R. S. Pressman. *Software Engineering, A Practitioner Approach*. McGraw-Hill, third edition, 1992.

[14] Reasoning Systems Inc., Palo Alto, CA. *Refine User's Guide*, version 3.0 edition, May 1990.

[15] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 115–116, Toronto, Canada, May 2001.

[16] K. Sartipi. A software evaluation model using component association views. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 259–268, Toronto, Canada, May 2001.

[17] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, Florence, Italy, November 2001. (to appear).

[18] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architectural design recovery using data mining techniques. In *Proceedings of IEEE CSMR 2000*, pages 129–139, Zurich, Switzerland, Feb 29 - March 3 2000.

[19] K. Sartipi, K. Kontogiannis, and F. Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of IEEE IWPC 2000*, pages 37–47, Limerick, Ireland, June 10-11 2000.

[20] A. I. Section. *CLIPS Architectural Manual Version 4.3*. Lyndon B. Johnson Space Center, jsc-23047 edition, May 1989.

[21] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, Nov./Dec. 1999.

[22] G. Snelting. Software reengineering based on concept lattices. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pages 1–8, Zurich, Switzerland, March 2000.

[23] V. Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *Proceedings of the International Workshop on Large-Scale Software Composition*, Vienna, August 1998.

[24] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

[25] T. A. Wiggerts. Using clustering algorithms in legacy systems modularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE Computer Society Press, October 1997.