Clustering Software Systems to Identify Subsystem Structures

Brian Mitchell
Department of Mathematics & Computer Science
Drexel University, Philadelphia, PA, USA
bmitchel@mcs.drexel.edu

Abstract

As the size of software systems continues to grow, understanding the structure of these systems gets harder. This coupled with associated problems such as of lack of current documentation, and the limited or nonexistent availability of the original designers of the system, adds further difficulty to the job of software professionals trying to understand the structure of large and complex systems. The application of clustering techniques and tools to software systems helps software designers, developers, and maintenance programmers by recovering high-level views of system designs.

In this paper we survey clustering approaches that have been developed by software engineering researchers. We also examine classical clustering techniques that have been applied in mathematics, science, and engineering, and investigate how these techniques have been adapted to work in the software domain. We conclude with a discussion of open research challenges related to software clustering.

1. Introduction

People look for abstractions to organize and categorize data so that similar data is grouped together and treated as a single logical entity. This grouping (clustering) reduces the amount of data that must be understood, which makes the original problem easier to understand.

Clustering has been applied in the fields of mathematics, social sciences, engineering and biology for many years. However, only within the last 25 years have researchers investigated and applied clustering techniques to the software domain. This work has resulted in new clustering techniques, as well as the adaptation of classical clustering techniques to the particularities of the software domain. This work has furthermore influenced the development of new programming languages (*e.g.,* classes/methods as abstractions of data and functions), which have, in turn, changed how we design systems.

1.1. The Complexity of Understanding the Structure of Software Systems

Most interesting software systems are large and complex, and as a consequence, understanding the structure of these systems is difficult. One of the primary reasons for this complexity is that these software systems are composed of many entities (*e.g.,* classes, modules, variables), which depend on each other in intricate ways (*e.g.,* procedure calls, inheritance relationships, variable references). Additionally, once an understanding of the system structure is formed, it is difficult to maintain, because software systems tend to evolve during maintenance.

Software Engineering textbooks advocate the use of documentation as an essential tool for describing a system's intended behavior, and for capturing the system's structure. Clearly, in order to be useful to future software maintainers, a system's documentation must be current with respect to any software changes. In practice, however, we often find that accurate and current design documentation does not exist. This problem is exacerbated because the original designers and developers of the system are often no longer available for consultation. Often the designers have found a new job, or are working on another project with deadlines that preclude them from providing advice to the current maintainers of the system. While documentation tools such as javadoc make the task of searching for key documentation in source code easier, these tools are of little value to software practitioners who are trying to understand the design structure of a software system.

In the absence of advice or documentation about a system's structure, software maintainers are left with several choices. First, they can manually inspect the source code to develop a mental model of the system organization. This approach is often not practical because of the large number of dependencies between the source code components. Another alternative that is now becoming available to software maintainers is to use automated tools to produce useful information about the system structure. A primary goal of these tools is to analyze the low-level dependencies in the source code, and cluster them into meaningful subsystems.

A subsystem is a collection of source code resources that closely collaborate with each other to implement a high-level feature, or provide a high-level service to the rest of the system. Typical source code resources that are found in subsystems include modules, classes, and in some cases, even other subsystems. Subsystems facilitate program understanding by logically treating many low-level source code resources as a single high-level entity. Interesting enough, subsystems are not specified in the source code explicitly, as most programming languages do not support this kind of structuring yet.

We have, however, seen some developments in programming language design to allow programmers to group low-level source code entities into common structures. While these language features can be used to represent subsystems, the actual placement of source code entitles into commonly named structures relies on programmer convention. For example, consider Java *packages* and C++ *namespaces*. These programming language features associate different source code entities with a common name (*e.g.,* `java.io.print()`, `java.io.println()`).

Because source code lacks the descriptiveness necessary to specify subsystems, researchers have been investigating other ways to represent subsystems. Several researchers have developed module interconnection languages (MIL)[9, 23, 6, 15], which are structured text specifications of a software design that typically include the ability to specify subsystems.

Mancoridis[15] presents a novel approach to specifying system components (including subsystems) and dependencies that is based on the Interconnection Style Formalism (ISF). ISF is a visual formalism for specifying high-level system designs and constraints on the designs. One problem with MILs and ISF is that they are documentation that must be maintained independent of the source code. A well-known problem in software engineering is keeping external documentation consistent with the source code as the system undergoes maintenance.

Even without programming language support for subsystems, developers often organize their source code into a set of subsystems. For example, consider the `FileSystem` subsystem of an operating system. We would expect this subsystem to include modules that support local files, remote files, mounting file systems, and so on. The question is, in the absence of this knowledge from designers, can the key subsystems of a software system be recovered directly from the source code?

We think the answer to the above question is yes. Thus far, however, creating tools to do so has proven to be difficult because automatically clustering software systems into subsystems is a hard problem. To illustrate this point consider representing the structure of a software system as a directed graph. The graph is formed by representing the modules of the system as nodes, and the dependencies between the modules as edges. We refer to this graph as the Module Dependency Graph (*MDG*) of a software system. Each partition of the *MDG* consists of a set of non-overlapping clusters that cover all of the nodes of the graph. The goal is to partition the *MDG* into meaningful subsystems.

Given an *MDG* with n components that are partitioned into k distinct clusters (subsystems), the number of ways to cluster the *MDG* satisfies the recurrence relation:

$$S_{n,k} = \begin{cases} 1 & if\ k = 1\ or\ k = n \\ S_{n-1,k-1} + S_{n-1,k} & otherwise \end{cases}$$

The entries $S_{n,k}$ are called *Stirling Numbers.* As the recurrence relation shows, the number of ways to partition the *MDG* into subsystems grows exponentially with respect to the number of modules in the system. We are only interested in partitions of the *MDG* that represent the subsystem structure of the source code, which are few relative to the total possible number of partitions. Hence, we need a technique to quickly filter out the "uninteresting" partitions and "discover" partitions that are representative of the subsystem structure of the system being analyzed.

Clustering systems using only the source code components and relationships is a hard problem. Other research has shown that most graph partitioning algorithms are NP complete or NP hard. We expect that the general problem of clustering software systems is NP hard, however, the formal proof of this remains an open problem. Most researchers have addressed the software clustering problem by using heuristics to reduce the execution complexity to a polynomial upper bound.

1.2. Overview of the Paper

The goal of this paper is to survey the research area of software clustering. In particular, we will investigate the established work in the area of software clustering. We will also examine research conducted in the area of classical clustering from the perspective of adopting these techniques to the software domain.

The remainder of this paper is organized as follows. In Section 2 we survey both software and classical clustering research. In Section 3 we investigate representation, similarity and algorithms that have been used for implementing clustering systems. In Section 4 we present some common observations found in the clustering research. In Section refSourceCodeVisualization we examine research in complementary fields including source code analysis and visualization. In Section 6 we conclude with a presentation of open research challenges in software clustering.

2. Historical Perspectives

Over the past few years there has been significant research activity in the field of reverse engineering. There are several reasons for this. First, with the year 2000 approaching, information technology professionals have spent significant amounts of time verifying that their software will work beyond the year 2000. Much of this software was remediated to work after the year 2000 by individuals who had no access to the original system designers. Second, there have been rapid changes in the software industry development processes. Just in the past few years we have seen information technology systems migrate from a two-tiered, to a three-tiered, to an n-tiered [1] (browser-based) client/server architectural model. Each time the system architecture changes, developers must understand how to adopt the previous version of their software to fit the new architectural model. The above situations are forcing software professionals to understand, and in some cases, remodularize vast amounts of source code. Software clustering tools can help these individuals by providing automated assistance for recovering the source code structure of large and complex systems.

[1]Industry literature commonly refers to n-tiered architectures as systems that are logically decomposed into presentation (user-interface), application logic and data access components. Each tier exposes an interface that may be deployed on a different computer and accessed over a computer network. Two-tiered architectures separate the data access from the presentation and application logic, which are combined into a single tier. Three-tiered architectures are designed with logical independence between the presentation, application logic, and data access components. With n-tiered architectures, the application logic and data access components are typically distributed across many physical hardware systems, and the presentation layer is often constructed using browser technology. Middleware frameworks such as Microsoft's DCOM, Javasoft's EJB, and CORBA are typically used to facilitate the distribution between the application components.

Given the timely importance of helping software professionals understand the structure of source code, the remainder of this section will review work performed by researchers in the area of software clustering.

In the earliest days of computing, the need for clustering low-level software entities, into higher-level structures (called modules) was identified. The landmark work of David Parnas[22] first suggested that the "secrets" of a program should be hidden behind module interfaces. The resultant information hiding principle advocated that modules should be designed so that design decisions are hidden from all other modules. Parnas suggested that interfaces to modules should be created to supply a well-defined mechanism for interacting with the modules' internal algorithms. Parnas suggested a programmer discipline whereby procedures that act on common data structures should be grouped (clustered) into common modules. Much of the basis for object-oriented design techniques evolved from Parnas's ideas.

Objected oriented techniques provide an initial level of clustering by grouping related data, and functions that operates on the data, into classes. Booch[3] suggests that, during the design process, a system should be decomposed into collaborating autonomous agents (*a.k.a.* objects) to perform some higher-level system behavior. Booch stresses the importance of using abstraction in design to focus on the similarities between related objects, encapsulation to promote information hiding, organizing design abstractions into hierarchies to simplify understanding, and using modularity to promote strong cohesion and loose coupling between classes. [2] Almost all research in software clustering concentrates on one or more of these concepts.

Information hiding and object oriented design embody some of the fundamental concepts of software clustering into the design process. However, without the availability of this design information, how can a programmer gain insight into the structure of the system? Furthermore, as a program undergoes maintenance, how can we ensure that the structure of the source code still agrees with the original design? We

[2]Cohesion and coupling measure the degree of component independence. Cohesion is a measurement of the internal strength of a module or subsystem. Coupling measures the degree of dependence between distinct modules or subsystems. According to Sommerville[28], well designed systems should exhibit high cohesion and low coupling.

will address these questions in the subsequent sections of this paper. The remainder of this section examines important research contributions in software clustering.

2.1. Software Clustering Research

In an early paper on software clustering, Hutchens and Basili[11] introduce the concept of a *data binding*. A data binding classifies the similarity of two procedures based on the common variables in the static scope of the procedures. Data bindings are very useful for modularizing software systems because they cluster procedures and variables into modules (*e.g.,* helpful for migrating a program from COBOL to C++). The authors of this paper discuss several interesting aspects of software clustering. First, they identify the importance of having consistency between the reverse engineered modules and the designer's mental model of the system's structure. The authors also claim that software systems are best viewed as a hierarchy of modules, and as such, they focus on clustering methods that exhibit their results in this fashion. Finally, the paper addresses the software maintenance problem, and identifies the benefit of cluster analysis to verify how the structure of a software system changes or deteriorates over time.

Robert Schwanke's tool Arch[25], provides a semi-automatic approach to software clustering. Arch is designed to help system architects understand the current system structure, reorganize the system structure, integrate advice from the system architect, document the system structure, and monitor compliance with the recovered structure. Schwanke's clustering heuristics are based on the principle of maximizing the cohesion of procedures placed in the same module, while at the same time, minimizing the coupling between procedures that reside in different modules. Arch also supports an innovative feature called maverick analysis. With maverick analysis, modules are refined by locating misplaced procedures. These procedures are then prioritized and relocated to more appropriate modules. Schwanke also investigated the use of neural networks and classifier systems to modularize software systems[26].

In Hausi Müller's work, we start to see the basic building block of a cluster change from a module to a more abstract software structure called a subsystem. Also, Müller's tool, Rigi[19], implements many heuristics that guide software professionals during the clustering process. Many of the heuristics discussed by Müller focus on measuring the "strength" of the interfaces between subsystems. A unique aspect of Müller's work is the concept of an omnipresent module. Often, when examining the structure of a system we find modules that act like libraries because they provide services to many of the other subsystems, or modules that behave like drivers because they use many of the other subsystems. Müller defines these modules as omnipresent and contends that they should not be considered when performing cluster analysis because they obscure the system's structure. One last interesting aspect of Müller's work is the notion that the module names themselves can be used to make clustering decisions. Later in this section we will discuss recent work by Anquetil and Lethbridge[2] who investigated this technique in more detail.

The work of Schwanke and Müller resulted in semi-automatic clustering tools where user direction is required to obtain meaningful results. Choi and Scacchi's[6] paper describes a fully automatic clustering technique that is based on maximizing cohesion. Their clustering algorithm starts with a directed graph called a resource flow diagram (RFD) and uses the articulation points of the RFD as the basis for forming a hierarchy of subsystems. The RFD is constructed by placing an arc from A to B if module A provides one or more resources to module B. Their clustering algorithm searches for articulation points, which are nodes in the RFD that divide the RFD graph into two or more connected components. Each articulation point and connected component of the RFD is used as the starting point for forming subsystems. Choi and Scacchi also used the NuMIL architectural description language to specify the resultant design of the system. They argue that NuMIL provides a hierarchical description of modules and subsystems, which is not directly specified in the flat structure of the system's source code.

Lindig and Snelting[14] propose a software modularization technique based on mathematical concept analysis. In their paper, they present a formalism for module structures, which includes formal definitions for an abstract data object (module), cohesion, coupling, and a mathematical concept. Their work is conceptually similar to Hutchens and Basili[11] where the goal is to cluster procedures and variables into modules based on shared global variable dependencies between procedures.

The first step in their software modularization technique is to generate a variable usage table. This table captures the shared global variables that are used by each procedure in the system. The authors then present a technique for converting the table into a concept lattice. A concept lattice is a convenient way to visualize the variable relationships between the procedures in the system. The next step in their modularization technique involves systematically updating procedure interfaces to remove dependencies on global data (the variable can be passed via the procedures interface instead). The goal is to transform the concept lattice into a tree-like structure. Once this transformation is accomplished, the modules are easily formed. The authors describe two techniques for accomplishing this transformation: *modularization by interface resolution* and *modularization via block relations*. Unfortunately, the authors were not able to modularize two large systems as part of their case study. Furthermore, their technique would probably only offer value in programming languages that rely on global data for information sharing (*i.e.,* COBOL, FORTRAN), but would not be well suited for object-oriented programming languages where data is typically encapsulated behind class interfaces.

Montes de Oca and Carver[7] present a formal visual representation model for deriving and presenting subsystem structures. Their work utilizes data mining techniques as the basis for forming subsystems. We agree with the authors that data mining techniques are complementary to the software clustering problem. Specifically:

- Data mining has been used to find non-trivial relationships between elements in a database. Software clustering tries to solve a similar problem by forming subsystem relationships based on non-obvious relationships between the source code entities.

- Data mining can discover interesting relationships in databases without upfront knowledge of the objects being studied. One of the significant benefits of software clustering is that it can be used to promote program understanding. As mentioned in Section 1.1, developers who work with source code are often unfamiliar with the intended structure of the system.

- Data mining techniques are designed to work with a large amount of information. Thus, the study of data mining techniques may advance the state of current software clustering tools. These tools typically suffer from performance problems due to the large amount of data that needs to be processed to develop the subsystem structures.

The authors use the dependencies of procedures on shared files as basis for forming subsystems. However, they do not describe in detail how their similarity mechanism works, as the primary goal of their research was to develop a formal visualization model for describing subsystems, modules, and relationships. We believe that the authors did a good job in capturing the requirements for a software visualization tool. For example, they support hierarchical nesting, and *singular programs*, which are programs that do not belong to a subsystem. Singular programs are very similar in definition to Müller's omnipresent modules[19].

Mancoridis, Mitchell et. al.[17, 16, 10, 18] treat the software clustering problem as an optimization problem. One key aspect of our clustering technique is that we do not attempt to cluster the native source code entities directly into subsystems. Instead, we start by generating a random subsystem decomposition of the software entities. We then use optimization techniques to move software entities between the randomly generated clusters, and in some cases we even create new clusters, to produce an "improved" subsystem decomposition. This process is repeated until no further improvement is possible. The optimization process is guided by an objective function that rewards high cohesion and low coupling. We have also researched several popular optimization search techniques that are based on traditional hill climbing and genetic algorithms. The capabilities described above, have been implemented in a Java tool called Bunch, which we made available for download (see http://serg.mcs.drexel.edu)

The original version of our tool clustered source code into subsystems automatically. In fact, it is the fully automatic clustering capability of Bunch that distinguishes it from related tools that require user intervention to guide the clustering process. However, we have extended Bunch over the past year to incorporate useful features that have been described, and in some cases implemented, by other researchers. For example, we incorporated Orphan Adoption techniques[30] for incremental maintenance, Omnipresent Module support[19], and user-directed clustering to complement Bunch's automatic clustering engine. These features are described in one of our papers[16]. We are currently working on other enhancements to Bunch. Most notably, we have created a distributed version of Bunch, and replaced our objective function with a faster version[18]. These changes enable Bunch to cluster larger systems faster.

Most approaches to clustering software systems are bottom-up. These techniques produce high-level structural views of a software system by starting with the system's source code. Murphy's work with Software Reflexion Models[20] takes a different approach by working top-down. The goal of the Software Reflexion Model is to identify differences that exist between a designers model of the system's structure, and the actual organization of the source code. By understanding these differences, the designer can update the model, or the source code can be modified to better adhere to the organization of the system that is understood by the designer. This technique is particularly useful to prevent the systems structure from "drifting" away from the intended system structure as a system undergoes maintenance.

In a recent paper, Anqetil, Fourrier and Lethbridge[1] conduced experiments on several hierarchal clustering algorithms. The goal of their research was to measure the impact of altering various clustering parameters when applying classical clustering algorithms to software remodularization problems. The authors define 3 quality metrics that allow the results of their experiments to be compared. Of particular interest, is the expert decomposition quality metric. This measurement quantifies the difference between the results of an automated clustering experiment, and the decomposition of the same system that is performed by an expert. The measurement has two parts: *precision* (agreement between the clustering method and the expert) and *recall* (agreement between the expert and the clustering method). The authors claim that many clustering methods have good precision and poor recall.

Anqetil et. al., also claim that a clustering method does not "discover" the system structure, but instead, imposes one. Furthermore, they state that it is important to choose a clustering method that is most suited to a particular system. We feel that this assessment is technically correct, however, imposing a structure consistent with the information hiding principle is often desired. As such, we find that most similarity measures are based on maximizing cohesion and minimizing coupling.

The authors make another strong claim, namely clustering based on naming conventions often produce better results than using source code features. The naming convention feature is based on clustering entities with similar names such as source code file names and procedure names. Anquetil and Lethbridge[2] have presented several case studies that show very promising results (high precision and high recall) using the file name feature to cluster software systems.

We feel that for some systems, using a naming convention feature as the basis for making clustering decisions will produce good results. One potential reason for this is that by convention some programmers spent significant time organizing their source files into different directories, and to name source code files that perform a similar function in a similar way. However, for some systems, this approach may not work well, especially when a system has undergone maintenance by programmers that did not thoroughly understand the system's structure. Using source code features as the basis for clustering will always provide accurate information to a clustering method, as this information is extracted directly from the relationships that exist in the source code.

In this section we presented a historical overview of the important research performed in software clustering. In the next section we examine the components necessary for constructing software clustering tools.

3. Clustering Techniques

In Section 1.1, we contend that one of the daunting challenges facing programmers is trying to understand the structure of the source code in the absence of formal design documentation. In this section, we examine clustering techniques that use the most up-to-date documentation of a system that is available to programmers - the source code. As a starting point, the survey paper by Wiggerts[32] provides an excellent overview of clustering approaches that, for the most part, have been used in other disciplines. Wiggerts argues that these classical clustering techniques can be adapted to cluster software systems.

Wiggerts contends that there are three fundamental issues that need to be addressed when applying cluster analysis to software systems:

1. Representation of the entities to be clustered.
2. Measurement of the similarity between the entities that are grouped into common subsystems.
3. Clustering algorithms that use the similarity measurement.

3.1. Representation of Source Code Entities

When clustering source code components, several things must be considered in order to determine how to represent the entities in the software system. There are many choices depending on the desired granularity of the recovered system design. First, should the entitles be procedures and methods, or modules and classes? Next, the type of relationships between the entities needs to be established. Should the relationships be weighted as to provide significance to special types of dependencies between the entities? For example, should two entities be more related if they use a common global variable? How does this weight compare to a pair of entities that have a dependency that is based on one entity using the public interface of another entity? As an example, in the Rigi tool, the user sets the criteria used to calculate the weight of the dependency between two entities manually. In Section 6, we identify that the identification of system entities, and determining the weight of the dependencies between two entities, warrants additional study.

3.2. Similarity Measurements

Once the type of entities and relationships are determined, "similarity" criteria between a pair of entities must be established. Similarity measures are designed so that larger values indicate a stronger similarity between the entities. For example, the Rigi tool establishes a "weight" to represent the collective "strength" of the relationships between two entities. The Arch tool uses a different similarity measurement whereby a similarity weight is calculated by considering common features that exist, or do not exist, between a pair of two entities. Wiggerts classifies the approach used by the abovementioned tools as *association coefficients*. Additional similarity measurements that have been classified are *distance measures*, which determine the dissimilarity between two entities (thus, if two entities do not share a common feature, they can be considered similar), *correlation coefficients*, which use the classical statistical notion of correlation to establish the degree of similarity between two entities, and *probabilistic measures*, which assign additional significance to rare features that are shared between entities (*i.e.,* entities which share rare features are more important then entities which share common features).

Example Similarity Measurements

1. **Rigi:** Rigi's primary similarity measurement, *Interconnection Strength* (*IS*), represents the exact number of syntactic objects that are exchanged or shared between two components. The *IS* measurement is based on information that is contained within a system's resource flow graph (RFG). The *RFG* is a directed graph $G = (V, E)$, where V is the set of nameable system components, and $E \subseteq V \times V$ is a set of tuples $\langle u, v \rangle$ which indicates that component u provides a set of syntatic objects to component v. Each edge $\langle u, v \rangle$ in the *RFG* contains an edge weight (*EW*) that is labeled with the actual set of syntactic objects that component u provides to component v. For example, if class v calls a method in class u named `getCurrentTime()`, then `getCurrentTime()` would be included in the *EW* set that labels the edge from u to v in the *RFG*. Given the *RFG*, and the *EW* set for each edge in the *RFG*, we can now show the *IS* similarity measurement.

$$Prv(s) = \bigcup_{x \epsilon V} EW(s,x) \quad Req(s) = \bigcup_{x \epsilon V} EW(x,s)$$
$$ER(s,x) = Req(s) \cup Prv(x)$$
$$EP(s,x) = Prv(s) \cup Req(x)$$
$$IS(u,v) = |ER(u,v)| + |EP(u,v)|$$

   Thus, the interconnection strength *IS* between a pair of source code entities is calculated by considering the number of shared exact requisitions (*ER*) and exact provisions (*EP*) between two modules. *ER* is the set of syntactic objects that are needed by a module and provided by another module. *EP* is the set of syntactic objects that are provided by a module and needed by another module. Both *ER* and *EP* are determined by examining the set of objects that are provided by a module, *Prv(s)*, and the set of objects that are needed (requisitioned) by a module *Req(s)*.

2. **Arch:** The Arch similarity measurement[26] is formed by normalizing an expression based on features that are shared between two software entities, and features that are distinct to each of the entities:

$$Sim(A,B) = \frac{W(a \cap b)}{1 + c \times W(a \cap b) + d \times (W(a-b) + W(b-a))}$$

   In Arch's similarity measurement, c and d are user-defined constants that are used to add or subtract significance to source code features. $W(a \cap b)$ represents the weight assigned to features that are common to entity A and entity B. $W(a-b)$ and $W(b-a)$ represent the weights assigned to features that are distinctive to entity A or entity B, respectively.

3. **Other classical similarity measurements:** Wiggerts presents some classical similarity measurements. The measurements are based on classifying how a particular software feature is shared, or not shared between two software entities. Consider the following table:

		Entity j	
		1	**0**
Entity i	**1**	a	b
	0	c	d

a: Number of common features in entity i and entity j
b: Number of features unique to entity j
c: Number of features unique to entity i
d: Number of features absent in both entity i and entity j

   Given the above table, similarity measurements are formed by considering the relative importance of how features are shared between two software entities. An interesting example is d, which represents the number of 0-0 matches, or the number of features that are absent in both entities. According to Wiggerts, the most commonly used similarity measurements are:

$$simple(i,j) = \frac{a+d}{a+b+c+d} \quad and \quad Jaccard(i,j) = \frac{a}{a+b+c}$$

Thus, the *simple* measurement treats 0-0 (type d) and 1-1 (type a) matches equally. The *Jaccard* measurement does not consider 0-0 matches. In a recent paper, Antquil et. al.[1] applied the *simple* and *Jaccard* similarity measurements to cluster Linux, Mosaic, gcc, and a large proprietary legacy telecommunication system. His results show that that *Jaccard* gives better results then the *simple* measurement.

3.3. Clustering Algorithms

Now that representation and similarity measures have been discussed, we turn our attention to describing algorithms for software clustering. Although many algorithms have been used for clustering, the most popular choice seems to be hierarchical algorithms.

Most hierarchical clustering algorithms start with the low-level source code entities (*i.e.,* modules and classes). These entities are then clustered into subsystems, which in turn, get clustered into larger subsystems (*i.e.,* subsystems that contain other subsystems). Eventually, everything will coalesce into a single cluster, resulting in a tree of clusters, where the leafs of the tree are the source code entities, and the interior nodes of the tree are the subsystems.

Hierarchies are found in many domains because they allow the original problem to be studied at different levels of detail by navigating up and down the different levels of the hierarchy. This property is particularly useful when applied to software understanding because of the large amount of data that needs to be considered.

Below, we present the clustering algorithms used by several known clustering tools:

- One of the simplest clustering algorithms is the one supported by the Rigi tool. In Rigi, two software entities are merged into a common subsystem if the similarity measure (*i.e.,* Interconnection Strength) exceeds the value of a user-defined threshold.

- Arch uses a form of the classical, hierarchical agglomerative clustering algorithm:

```
1. Place each entity in a subsystem by itself
2. Repeat
3.   Identify the two most similar entities
4.   Combine them into a common subsystem
5. Until the results are "satisfactory"
```

- The clustering algorithm described Anquetil and Lethbridge[2] is based on using the names of the source code files to form subsystems:

  1. Split each file name into a set of abbreviations formed from substrings in the file names.
  2. For each abbreviation found, create a cluster.
  3. For each file name, put it into each of the clusters that correspond to its abbreviations.

  The authors present several techniques for decomposing file names into abbreviations.

- The Bunch clustering algorithm works as follows:

```
1. Create a module dependency graph (MDG). $MDG = (M, R)$, where $M$ is the set of
   named modules in the system, and $R \subseteq M \times M$ is a set of ordered pairs
   $\langle u, v \rangle$ which represents the source-level relationships that exist
   between module $u$ and module $v$.
2. Generate a random set of partitions of the MDG. We refer to the entire set of
   partitioned MDG's as the population.
3. For each partition p in the population
4.   Repeat
5.     Let partition $p' = p$
6.     Use the modularization quality objective function, MQ, to measure the
       "quality" of $p$
7.     Let $q$ be a partition of the MDG found by applying one of our clustering
       algorithms to $p$
8.     if $MQ(q) > MQ(p)$ let $p = q$
9.   Until $MQ(p') = MQ(q)$
10. Return $p$
```

In the original version of Bunch[17], the *MQ* measurement was slow and computationally intensive. In a recent paper, we describe an *MQ*[18] measurement that can be computed more efficiently:

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{\mu_i}{\mu_i + \varepsilon_i} & otherwise \end{cases}$$

Given an *MDG* partitioned into k clusters, *MQ* is calculated by summing the Cluster Factors (*CF*) for each cluster. CF_i, for cluster i, is a normalized ratio for the number of edges (μ_i) that are contained within cluster i, and the number of edges (ε_i) that exit cluster i (these edges represents a dependency with a entity in another cluster).

In this section we investigated bottom-up clustering algorithms that use source code entities, and the dependencies between these entities, as the input to the clustering process. These approaches are well suited to software engineering problems because they only rely on the systems source code.

4. Observations

In the previous sections of this paper we reviewed significant software clustering research and presented various clustering techniques. In examining this work, several interesting observations can be made:

- Much of the clustering research makes mention to coupling and cohesion. Low coupling and high cohesion[28] are generally recognized properties of well-designed software systems. However, there does not seem to be any consistent, formal definitions for these properties.

- Many clustering algorithms are computationally intensive. In Section 1.1, we claim that clustering software systems is a hard problem. Many researchers address this complexity by creating heuristics to reduce the computation complexity so that results can be obtained in a reasonable time. Researchers such as Montes de Oca and Carver[7] are addressing computational complexity by looking into other areas such as Data Mining.

- Much of the clustering research uses experts to validate the results of their approach. However, how can one tell if expert support in a research case study is applicable to other systems? Thus, a more formal method of validating the results of a clustering technique is needed. Recent work by Antquetil and Lethbridge[2, 1] starts to address this problem. Specifically, they define metrics for precision and recall (see Section 2) which quantify the differences between an experts view and a recovered view of a systems structure. Clearly, this is better than an experts "opinion" about how good or bad the clustered result complies with the expert's mental model. A recent paper by Tzerpos and Holt[31] takes this concept one step further. In their paper, they define a distance metric called *MoJo* that can be used to evaluate the similarity of two different decompositions of a software system. Thus, using *MoJo*, the distance between an expert's decomposition and a clustering tool's decomposition of a software system can be measured.

- Many researchers recognize that a fundamental software engineering problem is that the structure of systems deteriorates as they undergo maintenance. To address this problem Tzerpos and Holt[30] present an orphan adoption technique to incrementally update an existing system decomposition by investigating how a change to the system impacts the structure. One problem with orphan adoption, however, is that it only looks at the impact of the changed (or new) module on the existing system structure. Without considering the remaining (existing) system components and relationships, the structure of the system can decay over time by repeatedly applying the orphan adoption technique unless a complete remodularization is performed periodically. Tzerpos and Holt's[31] work on *MoJo* can also help explain the impact of maintenance on a software system. By applying *MoJo* to two versions of a software system, a measurement for the differences between their structures can be obtained. Thus, if *MoJo* produces a large result, one should expect that there are significant structural differences between the two subsequent versions of the system.

- Early clustering research focused on clustering procedures into modules and classes. As software systems continued to grow, current research switched focus to clustering modules and classes into higher-level abstractions such as subsystems.

In the next section of this paper we introduce the importance of two bodies of work that are related to software clustering. First, source code analysis tools and techniques are required to provide input for clustering tools. Second, visualization tools are needed to present meaningful and understandable results to users of clustering tools.

5. Source Code Analysis and Visualization

Cluster analysis tools typically rely on several complementary technologies. In Section 3 we presented a framework for a clustering environment. This framework requires the establishment of the entities to be clustered (representation), the determination of how "similar" two entities are with respect to each other (similarity measurement), and clustering algorithms.

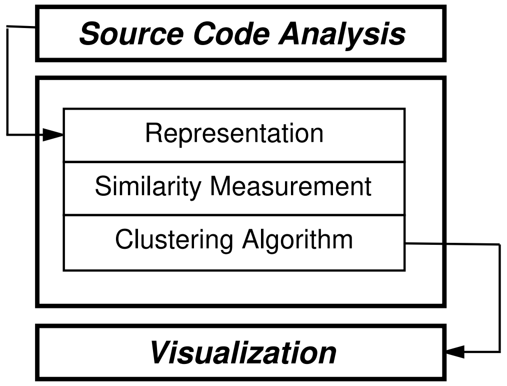

Figure 1. The Relationship Between Source Code Analysis and Visualization

In order to increase it's effectiveness to users, cluster analysis tools should enable a user to provide source code as input to the clustering process. Because typical systems of interest are large and complex, we want to avoid the user from having to transform the source code into the representation (*e.g., MDG*) required by the clustering tool manually. Not only is this manual task tedious and highly error prone, but we often want to iteratively cluster the same software system by varying the parameters supported by the clustering tool without having to revisit the original source code.

Another complementary technology to cluster analysis is visualization. The output of the clustering process often contains a large amount of data that needs to be presented to users. Without effective visualization techniques, it would be difficult to view the results of the clustering process. In Figure 1 we show the relationship between source code analysis, clustering, and visualization tools.

5.1. Source Code Analysis

Source code analysis has been researched for many years because it is desirable to consolidate a system's source code into a single repository that can be used for a variety of program understanding and reverse engineering activities. Repositories are useful to developers because they allow them to investigate the program structure and navigate through the many intricate relationships that exist between the source code components. In addition to clustering, source code analysis tools have been applied to other activities such as performing dead code detection, and reachability analysis.

The first step in source code analysis is to parse the source code and store the parsed artifacts into a repository. Source code analysis tools support a variety or programming languages such as COBOL, C, C++, Java and Smalltalk.

Once the repository is populated with information obtained from the source code, queries can be made against the repository to extract structural information about the source code. Two popular approaches for organizing the internal structure of the repository include storing the system's abstract syntax tree, or structuring the repository as a relational database based on the entities, and the relationships between the entities, that are found in the source code.

AT&T research has created a family of source code analysis tools for the C, C++ and Java languages. Their approach is based on an entity relationship model[4]. For each supported language syntax, the source code is scanned and a relational database is constructed capturing the entities in the program (*e.g.,* file names, variables, functions, macros), and the relationships between the entities (*e.g.,* variable reference, function call, inheritance). In addition to the entities and relationships, various attributes are recorded about the entities and relationships (*e.g.,* data types, declaration/definition).

Once the relational database is constructed, a series of tools can be used to query the database and produce information about the structure of the system being studied. The output of the query operation can be saved into a delimited text file. The query tools provided by AT&T can be combined with other standard Unix tools such as AWK to produce a consolidated representation of the systems entities and relationships.

In Table 1, we illustrate the result of running an AWK script which queries and filters the repository for the `Boxer` graph drawing system. Each row in the table consists of a tuple which represents a dependency between two modules in the `Boxer` system. For example, the tuple (main, boxParserClass) indicates that a dependency exists between the main module and the boxParserClass module. Table 1 only shows the dependencies between pairs of modules. Using the query tools provided by AT&T, one could capture additional information such as the dependency type (*e.g.,* global variable reference), a weighted average for the relative "strength" of the dependency, and so on.

The data presented in Table 1 can visualized to show the structure of the `Boxer` system. Figure 2 presents a graph formed from the data in Table 1. The figure illustrates the structure of the `Boxer` system. Because this is a relatively small system consisting of 18 modules and relatively few dependencies, the overall structure

Module 1	Module 2
main	boxParserClass
main	boxClass
main	edgeClass
main	Fonts
main	Globals
main	Event
main	hashedBoxes
main	GenerateProlog
main	colorTable
boxParserClass	Globals
boxParserClass	error
boxParserClass	boxScannerClass
boxParserClass	boxClass
boxParserClass	edgeClass
boxParserClass	stackOfAnyWithTop
boxParserClass	colorTable
boxParserClass	hashedBoxes
boxClass	edgeClass
boxClass	Fonts
boxClass	colorTable
boxClass	MathLib
boxClass	hashedBoxes
Globals	boxClass
hashedBoxes	hashGlobals
GenerateProlog	boxClass
GenerateProlog	Globals
stackOfAny	nodeOfAny
boxScannerClass	Lexer
stackOfAnyWithTop	stackOfAny

Table 1. **Dependencies of the Boxer System**

of `Boxer` should be understandable by inspecting the figure.

Sometimes, however, even small systems can be difficult to understand. In Figure 4, we show the graph of the dependencies recovered by performing source code analysis on `rcs`, an open source version control tool. Clearly, because of the number of dependencies, it is impractical if not impossible to develop a mental model for the structure of `rcs` by simply inspecting the dependencies between the systems modules.

When we applied our clustering tool, Bunch, to the dependency relationships for the `Boxer` and `rcs` systems, we obtain the graphs illustrated in Figure 3 and Figure 5, respectively. Figures 3 and Figure 5 present a clustered view of the `Boxer` and `rcs` systems, which show the native source code components clustered into subsystems. The clustered view of the source code conveys additional information to the user that is not obvious in the unclustered view of the same software system.

Source code analysis tools traditionally require access to the source code of the system in order to build a repository. The popularity of the Java language, along with its platform independent bytecode, enables analysis to be performed on the compiled bytecode, eliminating the need for the actual source code. Using bytecode, instead of source code, can also be used to perform dynamic analysis of a program when it is executing. This capability enables the user to capture dependencies between classes that would not be evident if static analysis was performed on the source code.

Several recent software engineering tools have been constructed that require only Java bytecode. Womble[12], extracts a systems object model directly from Java bytecode. Rayside, Kerr and Kontogannis[24] constructed a tool to detect changes between different versions of Java software systems using bytecode. Chava[13] complements the suite of source code analysis tools developed by AT&T for languages such as C[4] and C++[5]. Chava works with both the Java source code and the compiled bytecode.

As indicated earlier, cluster analysis tools require source code analysis utilities to package the native source code into a representation where the dependencies between the source code entities can be analyzed. Another important requirement for a clustering tool is the ability to present the results of the clustering process to the user. In the next section we discuss the importance of visualization tools. In particular, we will discuss tools such as dotty which are extensively used by researchers for graph visualization. We will also present some open problems associated with using standard graph tools for presenting software clustering results.

5.2. Visualization

The goal of a graph visualization tool is to present a graph clearly. Unfortunately, because most systems of interest are large and complex, providing a visualization of large system structures that is easy to interpret by a user, has proven to be difficult.

There are several reasons for this difficulty:

Navigation

- Most visualization tools only provide simple navigation and zooming facilities. This limitation often makes it difficult to see enough of the clustered result at a sufficient level of detail.

- When presenting clustering results to the user, it is desirable to first present a high-level view of

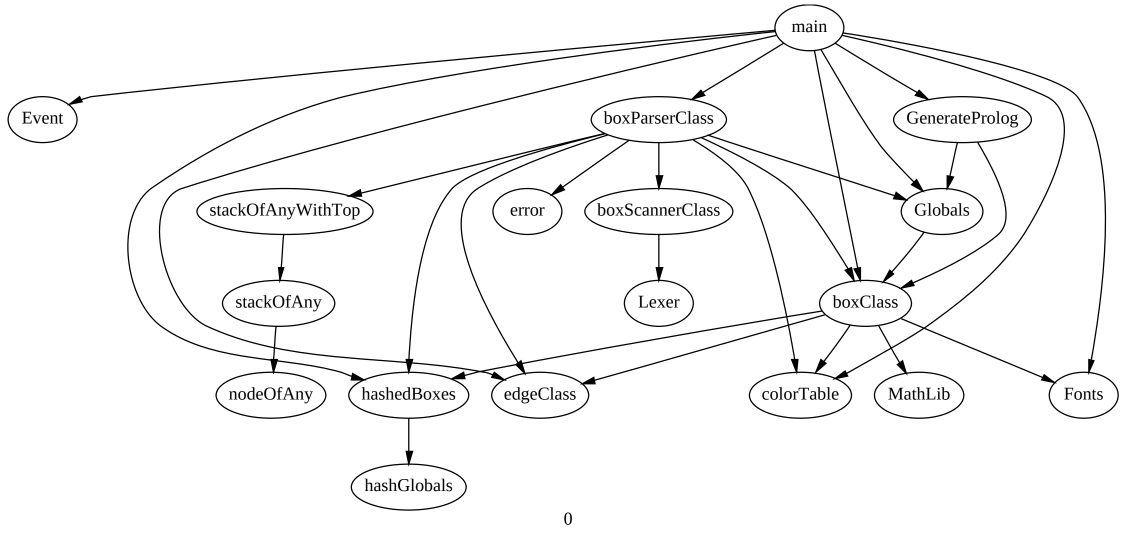

Figure 2. **The Structure of the Boxer System**

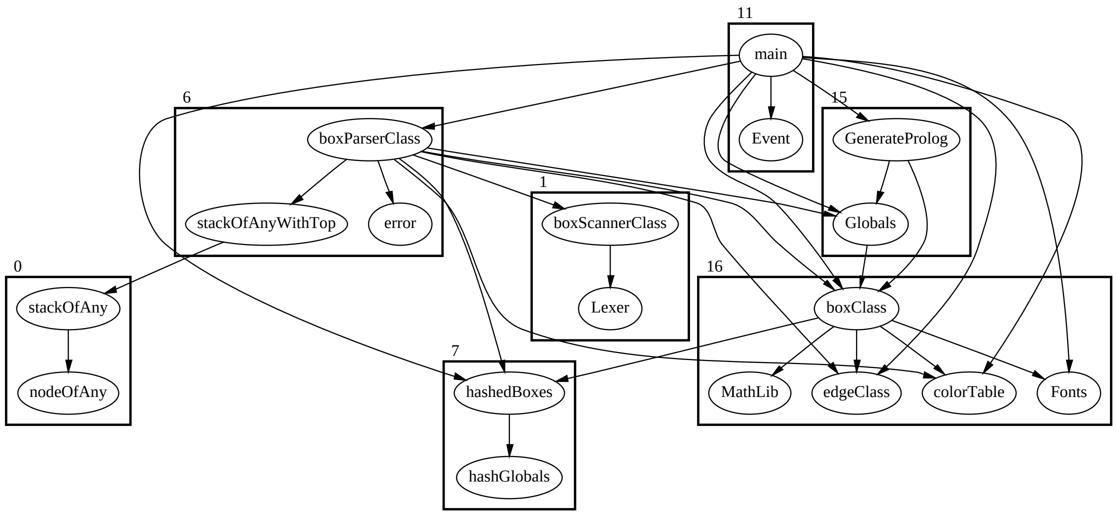

Figure 3. **The Clustered Structure of the Boxer System**

the systems structure, and then allow the user to drill-down into more detailed views showing the internal structure of the selected high-level component. Most visualization tools do not support this capability.

Integration

- Most visualization tools present results using traditional directed, or undirected graphs. While graphs effectively show the relationships between source code entities, clustering results have additional dimensions that need to be included in the visualization. For example, it is highly desirable to clearly show the entities that are clustered into common subsystems, and the relative strength of the relationships between the entities. To address some of these problems, some visualization utilities provide the capability to place labels on the nodes and edges in the graph. Although these features are helpful on small graphs, they only add additional clutter to large and highly interconnected graphs.

Scalability

- Many visualization tools are intended to be used standalone, or are tightly integrated into CASE environments. This makes it very difficult to tightly integrate them into other tools.
- Some clustering environments produce multiple views of the systems structure. For example, one

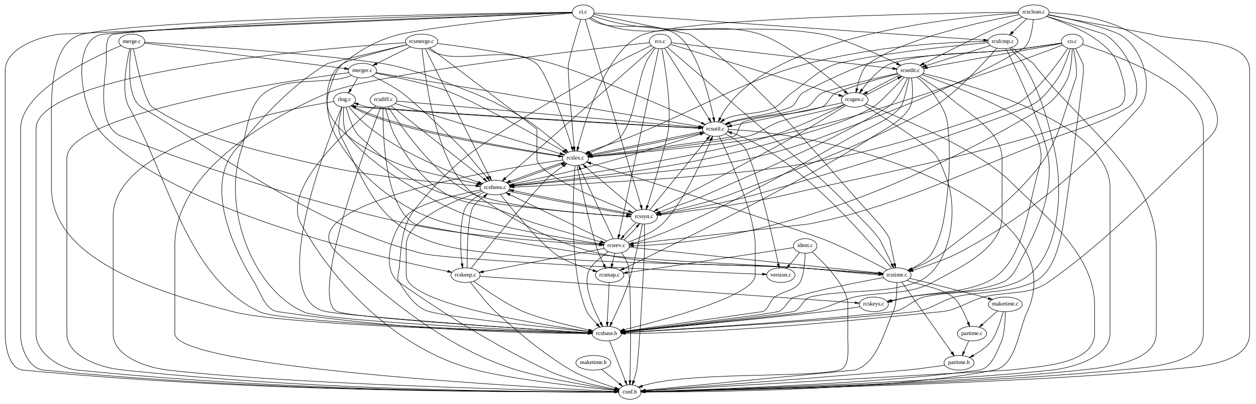

Figure 4. **The Structure of the `rcs` System**

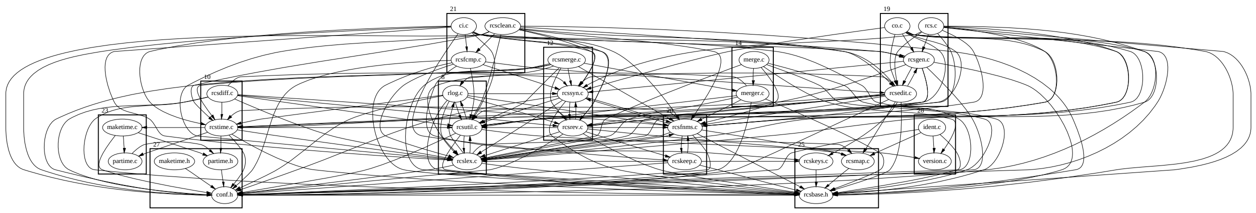

Figure 5. **The Clustered Structure of the `rcs` System**

view may contain the clustered subsystems, another view may describe the systems inheritance hierarchies, and so on. It would be desirable to have a visualization tool that organizes and relates these different views of the system.

The above list details some common problems with visualization tools. Not all tools exhibit all of the problems described above, but many are not capable of addressing all of the described issues. The remainder of this section will review some visualization tools and approaches that have been used in clustering systems.

Figures 2, 3, 4, and 5 were created using the AT&T graph layout tool dot[21]. Dot is very versatile and has been used by researchers for many years. One of the most notable features of dot is its graph description language. Users specify the nodes and edges of the graph in a plain text file, and may add any number of optional attributes for the nodes and edges to control the appearance of the resultant graph. For example, users can set attributes to control the font, font size, edge style, edge labels, colors of the nodes and edges, fill types, scaling, and so on. The dot documentation[21] contains a complete description of the plethora of options that can be included in a dot input file.

Dot is a command line utility that accepts a dot description file as input, and produces a graphical representation of the graph as output. The current version of dot supports 12 output file formats including GIF and PostScript. The dot layout engine uses advanced techniques to maximize the clarity of the produced graph by performing operations such as edge routing and edge crossing minimization.

While the goal of dot is to produce an output file, a complementary research tool by AT&T called dotty[21], is an online graph viewer. Dotty can be used to visualize and edit a graph that is specified in a dot description file. Dotty supports many common visualization functions such as birdseye views and zooming.

The simplicity of the dot description language, along with the power of the dot engine, allows the dot and dotty tools to be integrated into other tools easily.

Tom Sawyer[29] is a commercial graph visualization tool that we have integrated with our clustering tools. Tom Sawyer supports a text file description language that is not as robust as the dotty graph description file, however, Tom Sawyer supports many different layouts that make it easier to view the relationships between source code entities and subsystems.

Both dotty and Tom Sawyer allow users to specify the size, location, color, style, and label of the objects in the graph. These features are particularly useful when visualizing the results of a clustered system, because these attributes can be used to represent additional aspects of the clustered system. For example, we can alter the size of the nodes in the graph so that larger modules are "bigger" than smaller modules. Also, we can modify the thickness, style, grayscale or color of the edges in the graph to represent the "strength" of the relationship between two modules in the graph.

In a recent paper by Demeyer, Ducasse, and Lanza[8], the authors describe many visualization conventions based on node size, node position, node color, and edge color that represent different aspects of the structure of a software system. These techniques allow many dimensions of the software system's structure to be visualized at the same time, however, the user must constantly keep in mind what each layout convention means.

Figure 6 illustrates an example showing some of the conventions described in [8]. In this figure, a small systems inheritance tree is shown. The nodes represent the classes, and the edges represent the inheritance relationships. The width of each node represents the number of instance variables in a class, and the height of each node represents the number of methods in the class. Finally, the color tone of the node represents the number of class variables. Clearly, once the different conventions of the figure are understood, the two dimensional graph presents a lot of information to the user.

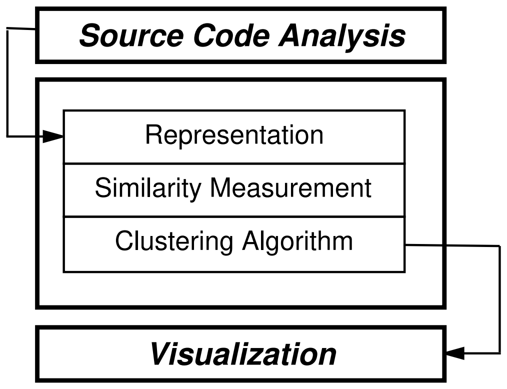

Figure 6. Inheritance Tree Showing Notational Conventions

The conventions specified in [8] are implemented in a tool called CodeCrawler. It is unclear how difficult it would be to integrate CodeCrawler into another utility because the authors do not describe an application programming interface, or a file format.

Another graphical modeling approach designed to represent structural design information is presented in [7]. The authors present a visual model called the Representation Model (RM), which consists of simple graphical objects and connectors. Each object represents a structural aspect of the system (*e.g.,* program, module, file, subsystem), and each connector represents a dependency within the system (*e.g.,* a program uses a file). RM uses a hierarchical visualization approach where the initial diagram presented to the user represents the high-level view of the system. Each high level RM object is linked to another RM diagram that visualizes additional detail about the internals of the initial RM object. This layering continues, forming a hierarchy of RM diagrams, until a primitive level is reached. The authors of the paper describe the model and plan to create an automated tool for creating, viewing and managing RM documents. Currently, RM documents are manually created in an external CASE or diagramming tool.

6. Future Work and Open Research Challenges

In this survey we presented research results from the area of software clustering. Specifically, we overviewed clustering techniques that that suggest module-level and subsystem-level system decompositions.

Over the past few years we have seen an increasing interest and activity in software clustering research. Through our study of this research we hope to provide an overview of what has been achieved and to inspire new research directions. This final section summarizes some of the open problems we intend to investigate in the near future:

- **Improving Performance.** In order to cluster a software system, a large amount of data must be analyzed. Since many clustering tools are computationally intensive, they do not perform well when applied to large systems. We have been investigating distributed techniques to scale our clustering approach, but improvement is still needed to cluster software systems with over 1 million lines of source code. To address this problem, we are currently working on extending the distributed clustering approach described in[18] to improve the scalability of Bunch. For environments where high-performance computers are not available, the distributed computation approach used by the SETI project[27] should be investigated. Data Mining techniques, as described by Montes de Oca and Carver[7], may also help address the performance problem.

- **Dynamic Analysis.** Many of the systems being developed today do not have all of their dependencies specified in the source code. Instead, these systems use Application Programming Interfaces (API's) provided by the operating system or programming language to dynamically load their classes (or modules) at runtime. Once loaded, the functionality provided by these components is indirectly invoked via a reference or proxy to the dynamically loaded component. Thus, to gain a good understanding of the structure of these systems, the runtime dependencies must be captured and included in the model of the system structure to be clustered. Technology provided by runtime debuggers and the Java Virtual Machine may provide a starting point for capturing these dynamic dependencies.

- **Representing Software Structure.** In Section 3.1 we discuss the importance of representing the entities and dependencies of a software system. Many software clustering algorithms (Section 3.3) and similarity measurements (Section 3.2) rely on the weight of the dependencies to make clustering decisions. The type of dependencies in software systems may be function calls, variable references, macro invocations, and so on. An open problem is to establish the collective weight of the dependencies that exist between the modules and classes in the system. Typically, all dependencies are treated equally, however, biasing the dependencies - so that certain types of dependencies are more significant than others - may be a good idea for improving clustering algorithms.

  Since most clustering techniques use graphs to represent software systems. It would be interesting to investigate the similarities and difference between software structure graphs and other types of graphs such as random, bipartite, connected, and directed acyclic graphs.

- **Improving Optimization Techniques.** In [10] we describe a Genetic Algorithm (GA) for clustering. GA's have been shown to produce good results in optimizing large problems. Although our initial results with GA's are promising, we feel that additional study is needed. We are especially interested in alternative encoding schemes, as the performance and results of applying GA's to optimization problems is highly dependent on the encoding of the problem.

- **Incremental Clustering.** As a system undergoes maintenance, there is a risk that the structure of the system will migrate from its intended design. One would expect that minor changes to the source code would not alter the system structure drastically. Thus, if an investment has already been made in decomposing a system into subsystems, it would be desirable to preserve this information as a starting point for updating, and in some cases, creating new subsystems. The Orphan Adoption[30] technique is a first step in addressing this problem. Due to the large computational requirements necessary to re-cluster a system each time it changes, we feel that the topic of incremental clustering warrants additional study.

- **Comparative Evaluation.** Most of the case studies reported in the clustering research are conducted on small and medium sized systems. There are enough clustering techniques to warrant a comparative study on how these algorithms perform against a set of systems of different sizes and complexity.

The above list confirms that there is a significant amount of important work that should be investigated by researchers. Also, it is timely to identify and start to work on some of the abovementioned research challenges, as the primary benefit of software clustering is to assist software professionals who are working on very large systems.

References

[1] N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with hierarchical clustering algorithms as software remodularization methods. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[2] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[3] G. Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company Incorporated, 2nd edition, 1994.

[4] Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[5] Y. Chen, E. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proc. 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1997.

[6] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.

[7] C. M. de Oca and D. Carver. A visual representation model for software subsystem decomposition. In *Proc. Working Conf. on Reverse Engineering*, Oct. 1998.

[8] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[9] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, pages 80–86, June 1976.

[10] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, 1999.

[11] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, Aug. 1995.

[12] D. Jackson and A. Waingold. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, 1997.

[13] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, 1999.

[14] C. Lindig and G. Snelting. Lightweight extraction of object models from bytecode. In *Proc. International Conference on Software Engineering*, Aug. 1999.

[15] S. Mancoridis. ISF: A Visual Formalism for Specifying Interconnection Styles for Software Design. *International Journal of Software Engineering and Knowledge Engineering*, 8(4):517–540, 1998.

[16] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, Aug. 1999.

[17] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, 1998.

[18] B. S. Mitchell, M. Traverso, and S. Mancoridis. Using networked general-purpose workstations to improve the performance of software clustering algorithms. In *Submitted for Publication*.

[19] H. Müller, M. Orgun, S. Tilley, and J. Uhl. Discovering and reconstructing subsystem structures through reverse engineering. Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, Aug. 1992.

[20] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. ACM SIGSOFT Symp. Foundations of Software Engineering*, 1995.

[21] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, 1994.

[22] D. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, pages 1053–1058, 1972.

[23] R. Prieto-Diaz and J. Neighbors. Module Interconnection Languages. *IEEE Transactions on Software Engineering*, 15(11), 1986.

[24] D. Rayside, S. Kerr, and K. Kontogiannis. Change and adaptive maintenance detectionin java software systems. In *Proc. Working Conf. on Reverse Engineering*, Oct. 1998.

[25] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[26] R. Schwanke and S. Hanson. ISF: A Visual Formalism for Specifying Interconnection Styles for Software Design. *Using Neural Networks to Modularize Software*, 15:137–168, 1998.

[27] The SETI Project. http://setiathome.ssl.berkeley.edu.

[28] I. Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1995.

[29] Tom Sawyer. Graph Drawing and Visualization Tool. http://www.tomsawyer.com.

[30] V. Tzerpos and R. Holt. The orphan adoption problem in architecture maintenance. In *Proc. Working Conf. on Reverse Engineering*, 1997.

[31] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clustering. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[32] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proc. Working Conference on Reverse Engineering*, 1997.