Using Automatic Clustering to Produce High-Level System Organizations of Source Code

S. Mancoridis, B. S. Mitchell, C. Rorres
Department of Mathematics & Computer Science
Drexel University, Philadelphia, PA, USA
{smancori, bmitchel, crorres}@mcs.drexel.edu

Y. Chen, E. R. Gansner
AT&T Labs - Research
Florham Park, NJ, USA
{chen,erg}@research.att.com

Abstract

This paper describes a collection of algorithms that we developed and implemented to facilitate the automatic recovery of the modular structure of a software system from its source code.

We treat automatic modularization as an optimization problem. Our algorithms make use of traditional hill-climbing and genetic algorithms.

Keywords: *Clustering, Reverse Engineering, Software Structure, Optimization, Genetic Algorithms.*

1. Introduction

Understanding the intricate relationships that exist between the source code components of a software system can be an arduous task. Frequently, this problem is exacerbated because the design documentation is out of date and the original system architect is no longer available for consultation.

With no mechanism for gaining insight into the system design and structure, the software maintenance practitioner is often forced to make modifications to the source code without a thorough understanding of its organization. As the requirements of heavily used software systems tend to change over time, it is inevitable that continually adopting an ad hoc approach to maintenance will have a negative effect on the overall modularity of the system. Over time, the system structure may deteriorate to the point where the source code organization is so chaotic that it needs to be radically overhauled or abandoned.

More than ever, software engineers rely on notations and tools to help them cope with the complexity of the structure of large software systems. One way programmers cope with structural complexity is by grouping (clustering) related procedures and their associated data into modules (or classes).

While modules do much to improve software development and maintenance, they are insufficient for supporting the design and ongoing maintenance of large systems. Such systems often contain several hundreds of thousands of lines of code that are packaged into a large number of cooperating modules. Fortunately, we often find that these systems are organized into identifiable clusters of modules, called subsystems, that collaborate to achieve a higher-level system behavior[3].

Unfortunately, the subsystem structure is not obvious from the source code structure. Our research therefore proposes an automatic technique that creates a hierarchical view of the system organization based solely on the components and relationships that exist in the source code. The first step in our technique is to represent the system modules and the module-level relationships as a module dependency graph. We then use our algorithms to partition the graph in a way that derives the high-level subsystem structure from the component-level relationships that are extracted from the source code.

Fully automatic modularization techniques are useful to programmers who lack familiarity with a system. These techniques are also useful to system architects who want to compare documented modularizations with the automatically derived ones, and possibly improve the design by learning from the differences between the modularizations.

Figure 1 shows the architecture of our automatic software modularization environment. The first step in the modularization process is to extract the module-level dependencies from the source code and store the resultant information in a database. We used AT&T's CIA tool[1] (for C) and Acacia[2] (for C++) for this step. After all of the module-level dependencies have been stored in a database, we execute an AWK script to query the database, filter the query results, and produce, as output, a textual representation of the module dependency graph. Our clustering tool, called Bunch,

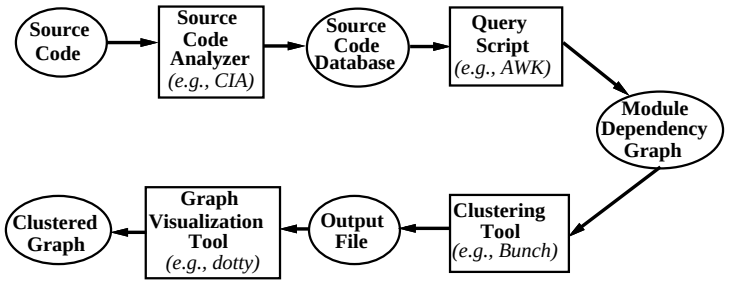

Figure 1. Automatic Software Modularization Environment

applies our clustering algorithms to the module dependency graph and emits a text-based description of the high-level structure of the systems organization. We then use the AT&T dotty visualization tool[9] to read the output file from our clustering tool and produce a visualization of the results.

The structure of the remainder of this paper is as follows: Section 2 presents a case study that illustrates the effectiveness of our automatic software modularization technique. Section 3 develops the pertinent aspects of our technique by formally quantifying inter-connectivity, intra-connectivity and modularization quality. Section 4 presents the algorithms we have implemented for clustering software components. Section 5 is dedicated to describing the operation and performance of our modularization tool. Section 6 presents related research in the area of software modularization. We conclude by outlining the research benefits and limitations of our work along with a discussion of our future plans to improve our technique.

2. An Example

Figure 2 shows the module dependency graph of a C++ program that implements a file system service. It allows users of a new file system `nos` to access files from an old file system `oos` (with different file node structures) mounted under the users' name space. Each edge in the graph represents at least one dependency relationship between program entities in the two corresponding source modules (C++ source files). For example, the edge between `oosfid.c` and `nos.h` is established due to 19 dependency relationships from the former to the latter.

The program consists of 50,830 lines of C++ code, not counting the system library files. The Acacia tool parsed the program and detected 463 C++ program entities and 941 dependency relationships between them. Note that containment relationships between classes/structs and their members are excluded for consideration in the construction of module dependency graphs.

Even with the module dependency graph, it is not clear what major components are in this system. Applying our automatic modularization tool to the graph results in Figure 3 with two large clusters and two smaller ones in each. After discussing the outcome of our experiment with the original designer of the system, several interesting observations were made:

1. It is obvious that there are two major components in this system. The right cluster mainly deals with the old file system while the left cluster deals with the new file system.

2. The clustering tool is effective in putting strongly-coupled modules like `pwdgrp.c` and `pwdgrp.h` in the same cluster even though the algorithm does not get any hints from the file names. Such clustering is consistent with the designer's expectation.

3. On the other hand, just by looking at the module names, a designer might tend to associate `oosfid.c` with the right cluster. Interestingly, the algorithm decided to put it in the left cluster because of its associations with `sysd.h` and `nos.h`, which are mostly used by modules in the left cluster. The designer later confirmed that the partition makes sense because it is the main interface file used by the new file system to talk to the other file system.

4. We cannot quite explain why a small cluster, consisting of `errlst.c`, `erv.c`, and `nosfs.h`, was created on the left. It might have been better to merge that small cluster with its neighbor cluster. A simple explanation is that our algorithm is only sub-optimal and may give a less-than-satisfactory answer in certain cases.

In the next two sections, we examine our algorithm in detail and shed some light on the heuristics used to obtain a sub-optimal solution.

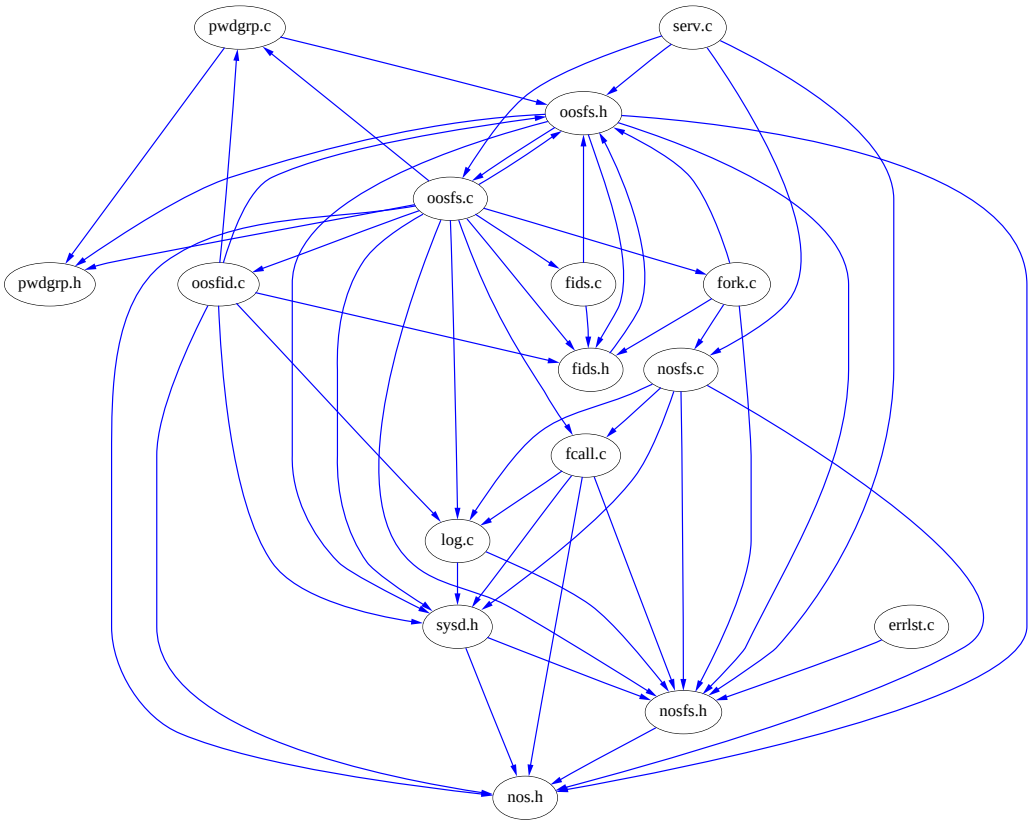

Figure 2. Module Dependency Graph of the File System

3. Automatic Software Modularization

Software systems contain a finite set of software components along with a finite set of relationships that govern how the software components interact with each other. Typical software components include classes, modules, variables, macros and structures; while common relationships include import, export, inherit, procedure invocation, and variable access. The goal of our software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity (*i.e.,* connections between the components of two distinct clusters) while maximizing intra-connectivity (*i.e.,* connections between the components of the same cluster). We accomplish this task by treating clustering as an optimization problem where our goal is to maximize an objective function based on a formal characterization of the trade-off between inter- and intra-connectivity.

The clusters, once discovered, represent higher-level component abstractions of a system's organization. Each subsystem contains a collection of modules that either cooperate to perform some high-level function in the overall system (*e.g.,* scanner, parser, code generator), or provide a set of related services that are used throughout the system (*e.g.,* file manager, memory manager). A fundamental assumption underlying our approach is that well-designed software systems are organized into cohesive clusters that are loosely interconnected.

3.1. Intra-Connectivity

We regard *Intra-Connectivity* (A) to be a measure of the connectivity between the components that are grouped together in the same cluster. A high degree of intra-connectivity indicates good subsystem partitioning because the modules grouped within a common subsystem share many software-level components. A low degree of intra-connectivity indicates poor subsystem partitioning because the modules assigned to a particular subsystem share few software-level components (limited cohesion). By maximizing the intra-connectivity measurement we increase the likelihood that changes made to a module are localized to the subsystem that contains the module.

We define the intra-connectivity measurement A_i of cluster i consisting of N_i components and m_i intra-edge dependencies as:

$$A_i = \frac{\mu_i}{N_i^2}$$

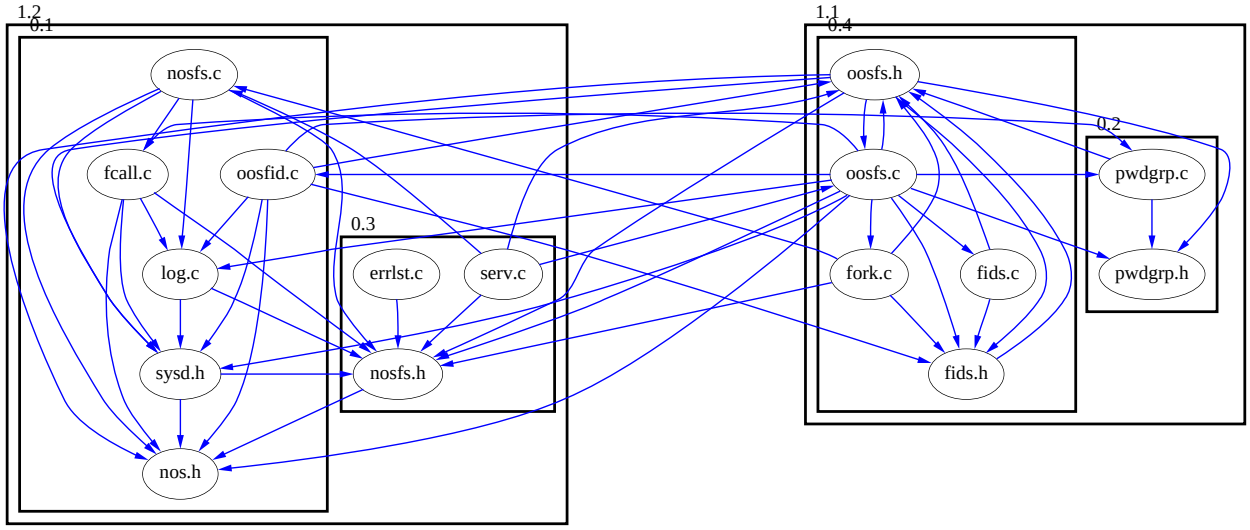

Figure 3. Automatically Produced High-Level System Organization of the Same File System

This measurement is a fraction of the maximum number of intra-edge dependencies that can exist for cluster i, which is N_i^2. The value of A_i is bounded between the values of 0 and 1. A_i is 0 when modules in a cluster do not share any software-level resources; A_i is 1 when every module in a cluster uses a software resource from all of the other modules in its cluster (*i.e.,* the modules and dependencies within a subsystem form a complete graph). In Figure 4 we apply our intra-connectivity measurement to a cluster containing three modules and two dependencies.

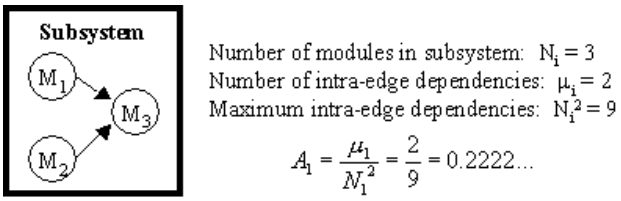

Figure 4. Intra-Connectivity Example

3.2. Inter-Connectivity

We regard *Inter-Connectivity* (E) to be a measurement of the connectivity between two distinct clusters. A high degree of inter-connectivity is an indication of poor subsystem partitioning. Having a large number of inter-dependencies complicates software maintenance because changes to a module may affect many other parts of the system due to the subsystem interrelationships. A low degree of inter-connectivity is a desirable trait of a system organization and is an indicator that the individual clusters of the system are, to a large extent, independent. Therefore, changes applied to a module are likely to be localized to its subsystem, which reduces the likelihood of introducing errors into other parts of the system.

We define the inter-connectivity E_{ij} between clusters i and j consisting of N_i and N_j components, respectively, with ε_{ij} inter-edge dependencies as:

$$E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\varepsilon_{i,j}}{2N_iN_j} & \text{if } i \neq j \end{cases}$$

Our inter-connectivity measurement is a fraction of the maximum number of inter-edge dependencies between clusters i and j ($2N_iN_j$). This measurement is bound between the values of 0 and 1. E_{ij} is 0 when there are no module-level dependencies between subsystem i and subsystem j; E_{ij} is 1 when each module in subsystem i depends on all of the modules in subsystem j and vice-versa. Figure 5 illustrates an example of the application of our inter-connectivity measurement.

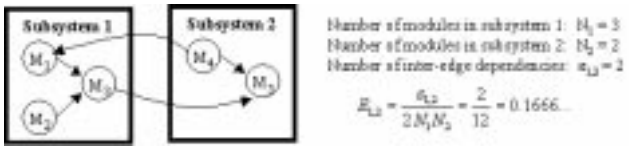

Figure 5. Inter-Connectivity Example

3.3. Modularization Quality

Recall that our goal is to discover a partitioning of the components of a software system that concurrently minimizes inter-connectivity and maximizes intra-connectivity. The Modularization Quality (MQ) measurement, which will be used as the objective function of our optimization process, is therefore defined as a measurement of the "quality" of a particular system modularization. Specifically, we define the MQ of a

module dependency graph partitioned into k clusters, where A_i is the Intra-Connectivity of the i^{th} cluster and E_{ij} is the Inter-Connectivity between the i^{th} and j^{th} clusters as:

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j} & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases}$$

The MQ measurement demonstrates the tradeoff between inter-connectivity and intra-connectivity by rewarding the creation of highly cohesive clusters, while penalizing the creation of too many inter-edges. This tradeoff is established by subtracting the average inter-connectivity from the average intra-connectivity. We use the average values of A and E to ensure unit consistency in the subtraction because the Intra-Connectivity summation is based on the number of subsystems (k), while the Inter-Connectivity summation is based on the number of distinct pairs of subsystems ($\frac{k(k-1)}{2}$). The MQ measurement is bounded between -1 (no cohesion within the subsystems) and 1 (no coupling between the subsystems). Figure 6 illustrates an example calculation of MQ.

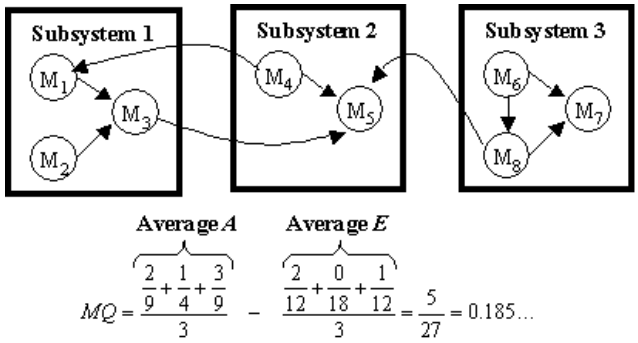

Figure 6. Modularization Quality Example

4. Modularization Algorithms

Now that the MQ measurement has been defined, we turn our attention to developing algorithms that start with the module dependency graph of the source code and produce, as output, a hierarchy of clusters that represents the subsystem structure of a software system. Figure 7 depicts the software modularization algorithms that are supported by Bunch. The optimal algorithm produces the best results, but it only works for small systems. The other two algorithms are much faster, but they may not produce an optimal result. The remainder of this section describes these algorithms in detail.

The first step in our automatic modularization process is to parse the source code and build a module

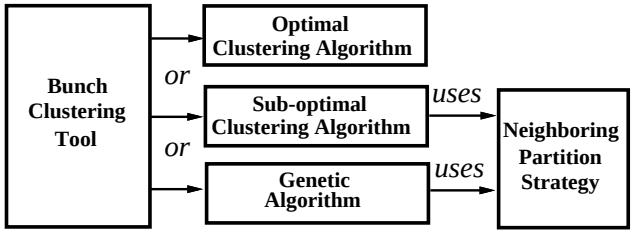

Figure 7. Automatic Clustering Algorithms

dependency graph. Formally, a module dependency graph $MDG = (M, R)$ consists of two components M and R where: M is the set of named modules in the software system, and $R \subseteq M \times M$ is a set of ordered pairs of the form $\langle u, v \rangle$ which represents the source-level relationships that exist between module u and module v. Once the module dependency graph is constructed, we apply our modularization algorithms to it. The remainder of this section develops some additional theory and then presents a collection of algorithms that we have implemented to automatically partition software systems.

4.1. Partitions of a Set

Consider the source code organization of a software system. Let S be a set of modules $\{M_1, M_2, \ldots, M_n\}$ where each module contains source code features (*i.e.*, variables, macros, functions, procedures, constants). Let $\pi = A_1, A_2, \ldots, A_k$ be a set of non-empty subsets of S. We call π a *partition* of set S if:

1. $\bigcup_{i=1}^{n} A_i = S$
2. $A_i \bigcap A_j = \emptyset, \forall 1 \leq i, j \leq n \cdot i \neq j$

If π is a partition, we call each subset A_i a *cluster* of S. Also, a partition of S into k non-empty clusters is called a *k-partition* of S.

Given a set S that contains n elements, the number $S_{n,k}$ of distinct *k-partitions* of the set satisfies the recurrence equation:

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

The entries $S_{n,k}$ are called *Stirling numbers* and grow exponentially with respect to the size of S. For example, a 5-node module dependency graph would have 52 distinct partitions, while a 15-node module dependency graph would have 1,382,958,545 distinct partitions.

4.2. The Optimal Clustering Algorithm

We now present our algorithm for determining the optimal clustering of a software system.

1. Let $S = \{M_1, M_2, \ldots, M_n\}$, where each M_i is a module in the software system.
2. Let MDG be the graph representing the relationships between the modules in S.
3. Generate every partition of set S.
4. Evaluate MQ for each partition.
5. The partition with the largest MQ is the optimal solution.

We have successfully applied the Optimal Clustering Algorithm to systems of up to 15 modules. Beyond that, the search space (number of k-partitions of S) becomes so large that it cannot be explored in a reasonable time-frame. Clearly, sub-optimal techniques must be employed for systems with a large number of modules.

4.3. Neighboring Partitions

Our sub-optimal clustering technique relies on moving modules between the clusters of the partition so as to improve the MQ. This task is accomplished by generating a set of neighboring partitions (NP) for a partition.

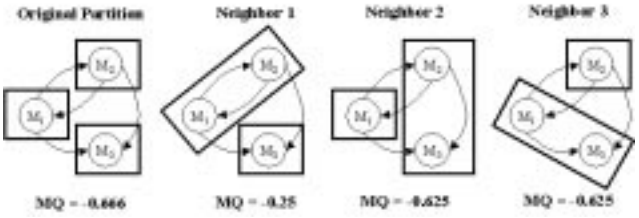

Figure 8. Neighboring Partitions

We define a partition NP to be a neighbor of a partition P if and only if NP is exactly the same as P except that a single element of a cluster in partition P is in a different cluster in partition NP. Figure 8 illustrates the process of determining all of the neighboring partitions of P.

Although there are many other ways to define a neighboring partition, this one is simple to understand and implement and offers good execution performance.

In other automated software modularization techniques[10], a poor module movement decision early on can negatively bias the final results because there is no facility for moving a module once it has been placed into a cluster. A useful property of our neighboring partition approach is that the assignment of a module to a cluster is not permanent.

4.4. Sub-Optimal Clustering Algorithm

The search space required to enumerate all possible partitions of a software system becomes prohibitively large as the number of modules in the system increases. Therefore, we directed our attention to developing a search strategy, based on traditional hill-climbing optimization techniques, that quickly discovers an acceptable sub-optimal clustering result.

In summary, our sub-optimal clustering algorithm starts with a random partition and repeatedly finds better neighboring partitions until no neighboring partition can be found with a higher MQ.

Sub-Optimal Clustering Algorithm

1. Let $S = \{M_1, M_2, \ldots, M_n\}$, where each M_i is a module in the software system.
2. Let MDG be the graph representing the relationships between the modules in S.
3. Generate an initial random[8] partition P of set S.
4. Repeat
   - Randomly select a better neighboring partition bNP (*i.e.,* one that has $MQ(bNP) > MQ(P)$).
   - If a bNP is found, then let $P = bNP$.

   Until no further "improved" neighboring partitions can be found.
5. Partition P is the sub-optimal solution.

In our sub-optimal clustering algorithm, a better neighboring partition (bNP) is discovered by going through the set of neighboring partitions of P, one-by-one, until a partition with a higher MQ is found.

4.5. A Genetic Algorithm Implementation

Our experimentation with the sub-optimal clustering algorithm has shown that, given an initial random starting partition, the algorithm will always converge to a local maximum. However, not all randomly generated initial partitions improve to an acceptable sub-optimal result. One approach to solving this problem is to run the experiment many times using different initial partitions and pick the experiment that results in the largest MQ as the sub-optimal solution. As the number of experiments increases, the probability of finding the globally optimal partition (based on the MQ) also increases.

Another more systematic approach to solving our optimization problem is based on Genetic Algorithms. Discovering an acceptable sub-optimal solution based on Genetic Algorithms involves starting with a population of randomly generated initial partitions and systematically improving them until all of the initial samples converge. In this approach, the resultant partition with the largest MQ is used as the sub-optimal solution.

Genetic Algorithms[4] have been successfully applied to many problems that involve exploring large search spaces. They combine a survival-of-the-fittest technique with a structured and randomized information exchange to facilitate innovative search algorithms that parallel the theory of natural selection. Genetic Algorithms are more than a randomized search; instead, they exploit historical data to speculate new information that is expected to yield improved results.

We now present our genetic search algorithm for finding a sub-optimal partition of a software system:

Genetic Sub-Optimal Clustering Algorithm

1. Let $S = \{M_1, M_2, \ldots, M_n\}$, where each M_i is a module in the software system.
2. Let MDG be the graph representing the relationships between the modules in S.
3. Generate a population of N random partitions of set S.
4. Repeat
   - Randomly select a percentage of N partitions from the population and improve each one by finding a better neighboring partition, bNP.
   - Generate a new population of N partitions by making N selections, with replacement, from the existing population of N partitions. These selections are to be random and biased in favor of partitions with larger MQs.

   Until no improvement is seen for t generations, or until all of the partitions in the population have converged to their maximum MQ, or until the maximum number of generations ($MaxG$) has been reached.
5. The partition P in the final population with the largest MQ is the sub-optimal solution.

Our genetic clustering implementation has several user-configurable parameters. These include setting the population size (N), the maximum number of generations to execute ($MaxG$) before concluding that the experiment did not converge, and the convergence threshold (which is a percentage of $MaxG$).

4.6. Hierarchical Clustering

The algorithms presented in the previous section generated partitions based of the MDG graph, which was formed by recovering the relationships between source code components. However, when performing analysis on large software systems the number of clusters found in a partition may be large. In this case it makes sense to cluster the clusters, thus creating a hierarchy of subsystems.

Several software modularization techniques[5, 10] support hierarchical clustering. Our technique does so as well in the following way:

The first step in the hierarchical clustering process is to apply our standard software modularization algorithms to the MDG graph. This activity discovers a partition, P_{lm}, which represents a partition that has converged to a local maximum (lm). We then build a new higher-level graph by treating each cluster in P_{lm} as a single component. Furthermore, if there exists at least one edge between any two clusters in P_{lm} then there is an edge between their representative nodes in the new graph. We then apply our clustering algorithms to the new graph in order to discover the next higher-level graph, and so on. This process is applied iteratively until all of the components have coalesced into a single cluster (*i.e.,* the root of the subsystem decomposition hierarchy).

5. The Bunch Clustering Tool

We have implemented the algorithms described above and applied them to many example software systems. Table 1 presents performance measurements for some common systems that were processed by Bunch. The computation environment used for these experiments was a Pentium 166 computer with 80 Mb of RAM, running the WindowsNT 4.0 operating system. The execution times shown in Table 1 were collected running Bunch under the Microsoft J++ virtual Java machine. We experienced similar performance results using the Java just-in-time (JIT) compiler provided by Sun Microsystems in Solaris 2.6.

Interested readers may download a copy of our software from the Drexel University Software Engineering Research Group home page http://www.mcs.drexel.edu/~serg.

6. Related Work

The problem of automatic modularization (also referred to as automatic clustering) has been extensively researched over the past two decades. A recent paper by Wiggerts[11] provides an excellent introductory survey to the use of clustering in systems remodularization. Two widely referenced clustering tools that have been developed to specifically address the software remodularization problem are Rigi[6] and Arch[10]. These tools, however, employ clustering techniques that rely on the intervention from an architect who understands the system structure in order to produce good results. As a result, these techniques are of little help to someone who is not familiar with a software system, yet is trying to understand its structure.

Hutchens and Basili[5] presented an automatic clustering technique based on data bindings. Unfortunately, the use of data bindings as the basis for performing a software modularization has some shortcomings. Specifically, if the system modules exhibit strong encapsulation (*i.e.,* hide their data), then there

System Name	System Type	Module Count	Module-Level Relationships	Execution Time
ispell	Unix Spell Checker	22	98	22.793 sec.
rcs	Version Control System	27	159	46.256 sec.
mtunis	Small Operating System	20	57	17.876 sec.
lu	Proprietary System	153	103	1 hour 24.924 sec.

Table 1. Bunch Tool Performance

is no way of determining their module-level relationships with data bindings because of the limited number of publicly accessible variables. Additionally, modularization based on data bindings addresses the problem of clustering procedures and variables into classes and modules. Our objective is to cluster related modules and classes into subsystems, which is useful when systems have a large number of modules.

In addition to the bottom-up clustering approaches, which produce high-level structural views starting with the structure of the source code, research emphasis has been placed on top-down approaches. For example, the goal of the Software Reflexion Model[7] is to capture and exploit the differences that exist between the source code organization and the designer's mental model of the high-level system organization. The primary purpose of this technique is to streamline the amount of time it takes for someone unfamiliar with the system to understand its source code structure.

7. Conclusions and Future Work

Experimentation with our clustering technique has shown good results for many of the systems that we have investigated. The primary method that we use to evaluate our results is to present an automatically generated modularization of a software system to the actual system designer(s) and ask for feedback on the quality of the results.

While we were able to produce good results for many of the systems that we examined, one known shortcoming with our current definition of modularization quality (MQ) is that it does not take into account the *Interconnection Strength* (IS)[6] of the relationships that exist between the modules in the software system. According to Müller et. al., IS is a measurement of the exact number of syntactic objects that are exchanged or shared between two modules. Thus, our clustering technique, which is based strictly on the topology of the module dependency graph, might not convey an accurate representation of a systems modularization when the magnitude of the interconnection strengths of the actual module relations differ significantly.

In order to address this shortcoming, we are currently working on an extension to our definitions of inter-connectivity, intra-connectivity and modularization quality that accounts for the weight of the module-level dependencies. We expect this extension to yield better results for systems in which the distribution of interconnection strength values is non-uniform. Our current assumption is that the value of IS for module-level dependencies is equal to one.

References

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

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

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

[4] D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, 1989.

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

[6] 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.

[7] 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.

[8] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms*. Academic Press, 2nd edition, 1978.

[9] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, pages 235–245, 1994.

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

[11] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Working Conference on Reverse Engineering (WCRE97)*, 1997.