# On the Automatic Modularization of Software Systems Using the Bunch Tool

Brian S. Mitchell, *Member*, *IEEE*, and Spiros Mancoridis, *Senior Member*, *IEEE*

**Abstract**—Since modern software systems are large and complex, appropriate abstractions of their structure are needed to make them more understandable and, thus, easier to maintain. Software clustering techniques are useful to support the creation of these abstractions by producing architectural-level views of a system's structure directly from its source code. This paper examines the Bunch clustering system which, unlike other software clustering tools, uses search techniques to perform clustering. Bunch produces a subsystem decomposition by partitioning a graph of the entities (e.g., classes) and relations (e.g., function calls) in the source code. Bunch uses a fitness function to evaluate the quality of graph partitions and uses search algorithms to find a satisfactory solution. This paper presents a case study to demonstrate how Bunch can be used to create views of the structure of significant software systems. This paper also outlines research to evaluate the software clustering results produced by Bunch.

**Index Terms**—Clustering, reverse engineering, reengineering, program comprehension, optimization, maintainability.

## 1 Introduction

Without insight into a system's design, a software maintainer might modify the source code without a thorough understanding of its organization. As the requirements of heavily used software systems change over time, it is inevitable that adopting an undisciplined approach to software maintenance will have a negative effect on the quality of the system structure. Eventually, the system structure may deteriorate to the point where the source code organization is so chaotic that it needs to be overhauled or abandoned.

Automatic design extraction methods create aggregate views—similar to "road maps"—of a system's structure. Such views help software engineers to cope with the complexity of software development and maintenance; they also help with reengineering activities associated with remodularizing a system and may even help project managers assign work to different developers (or development teams) by identifying areas of the system that are loosely coupled to each other.

Design extraction starts by parsing the source code to determine the components and relations of the system. The parsed code is then analyzed to produce views of the software structure, at varying levels of detail. Detailed views of the software structure are appropriate when the software engineer has isolated the subsystems that are relevant to his or her analysis. However, abstract (architectural) views are more appropriate when the software engineer is trying to understand the global structure of the software. Software clustering may be used to produce such abstract views. These views encapsulate source code-level components and relations into subsystems. The source code components and relations can be determined using source code analysis tools. The subsystems, however, are not found in the source code. Rather, they must be inferred from the source code either automatically, using a clustering tool, or manually (e.g., using the package/directory structure to define clusters), when tools are not available.

The reverse engineering research community has been actively investigating techniques to decompose (partition) the structure of software systems into subsystems (clusters) [1], [2], [3], [4], [5], [6], [7]. Subsystems provide developers with structural information about the numerous software components, their interfaces, and their interconnections. Subsystems generally consist of a collection of collaborating source code resources that implement a feature or provide a service to the rest of the system. Typical resources found in subsystems include modules, classes, and, possibly, other subsystems. Subsystems facilitate program understanding by treating sets of source code resources as software abstractions.

Unlike many other software clustering techniques, our approach evaluates the quality of a graph partition that represents the software structure, and it uses heuristics to navigate through the search space [8] of all possible graph partitions. We have developed several promising heuristic approaches to solving this problem, and we outline our findings in the remaining sections of this paper.

This paper is organized as follows: Section 2 describes how the software clustering problem can be modeled as a search problem and presents an example illustrating the clustering process. Section 3 describes the architecture of the Bunch software clustering tool. Section 4 describes the software clustering algorithms that are implemented in the Bunch tool. Section 5 demonstrates an iterative session in which a software engineer uses Bunch to build and incrementally refine useful structural views of a software system. Section 6 describes several qualitative and quantitative approaches that were used to evaluate the results produced by Bunch. Section 7 discusses how Bunch's clustering approach relates to other work performed by the reverse engineering community. Section 8 outlines the contributions this work makes to the software engineering community. Conclusions are presented in Section 9.

• *The authors are with the Department of Computer Science and Software Engineering Research Group at Drexel University, Philadelphia, PA 19104. E-mail: {bmitchell, spiros}@drexel.edu.*

*Manuscript received 5 Mar. 2004; revised 20 Dec. 2005; accepted 28 Dec. 2005; published online DD Mmmm, YYYY.*
*Recommended for acceptance by D. Weiss.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0035-0304.*

## 2 Software Clustering as a Search Problem

Our software clustering approach is independent of any programming language. To accomplish this objective, we rely on source code analysis tools to transform source code into a directed graph. We refer to this graph as the Module Dependency Graph (*MDG*). The *MDG* is a language-independent representation of the structure of the system's source code components and relations. This representation includes all of the modules in the system and the set of dependencies that exist between the modules. The *MDG* is constructed automatically using source code analysis tools such as CIA [9] for C, Acacia [10] for C and C++, and Chava [11] for Java.

We define a module to be a source code entity that encapsulates data and functions that operate on the data (e.g., Java/C++ classes, C source code files). The dependencies between the modules are binary relations that are supported by the programming language used to implement the system (e.g., function/procedure invocation, variable access, inheritance). Although there are many types of software structures, our work focuses on clustering graphs of the static modules and dependencies of software systems. When the modules and relations are captured using source code analysis tools, the clustering results are based on the static structure of the source code. While the focus of this paper will be on clustering *MDG*s created this way, it should be noted that we also clustered *MDG*s that were created by collecting profiling information at runtime [12]. This type of dynamic analysis can capture dependencies that are not specified in the source code (e.g., asynchronous calls, procedures that are linked at runtime).

Our approach to solving the clustering problem using search techniques can be stated informally as "finding a good partition of the *MDG*." We use the term *partition* in the graph-theoretic sense, that is, the decomposition of a set of elements (i.e., all nodes of a graph) into mutually disjoint sets (i.e., clusters). By a "good partition" we mean a partition where highly interdependent modules (nodes) are grouped in the same subsystems (clusters) and, conversely, independent modules are assigned to separate subsystems.

Given an *MDG*, $G = (V, E)$, we define a partition of $G$ into $n$ clusters formally as $\Pi_G = \{G_1, G_2, \ldots, G_n\}$, where each $G_i$ $((1 \leq i \leq n) \wedge (n \leq |V|))$ is a cluster in the partitioned graph. Specifically,

$$
\begin{gathered}
G = (V, E),\ \Pi_G = \bigcup_{i=1}^{n} G_i, \\
G_i = (V_i, E_i), \\
\bigcup_{i=1}^{n} V_i = V, \\
\forall((1 \leq i, j \leq n) \wedge (i \neq j)),\ V_i \cap V_j = \emptyset, \\
E_i = \{\langle u, v\rangle \in E \mid u \in V_i \wedge v \in V\}.
\end{gathered}
$$

If partition $\Pi_G$ is the set of clusters of the *MDG*, each cluster $G_i$ contains a nonoverlapping set of modules from $V$ and edges from $E$. The number of clusters in a partition ranges from 1 (a single cluster containing all of the modules) to $|V|$ (each module in the system is placed into a singleton cluster). A partition of the *MDG* into $k$ $(1 \leq k \leq |V|)$ nonempty clusters is called a *k-partition* of the *MDG*.

Given an *MDG* that contains $n = |V|$ modules, and $k$ clusters, the number $G_{n,k}$ of distinct *k-partitions* of the *MDG* satisfies the recurrence equation:

$$
G_{n,k} = \begin{cases} 1 & \text{if } k = 1 \text{ or } k = n \\ G_{n-1,k-1} + kG_{n-1,k} & \text{otherwise.} \end{cases}
$$

The entries $G_{n,k}$ are called *Stirling numbers* [13] and grow exponentially with respect to the size of set $V$. For example, a 5-node module dependency graph has 52 distinct partitions, while a 15-node module dependency graph has 1,382,958,545 distinct partitions.

### 2.1 Evaluating MDG Partitions

The primary goal of our software clustering algorithms is to propose subsystems that expose useful abstractions of the software structure. Finding good partitions quickly involves navigating through the very large search space of all possible *MDG* partitions in a systematic way. To accomplish this task efficiently, our clustering algorithms treat graph partitioning (clustering) as a search problem. The goal of the search is to maximize the value of an objective function, which we call *Modularization Quality (MQ)*.

*MQ* determines the quality of an *MDG* partition quantitatively as the trade-off between interconnectivity (i.e., dependencies between the modules of two distinct subsystems) and intraconnectivity (i.e., dependencies between the modules of the same subsystem). This trade-off is based on the assumption that well-designed software systems are organized into cohesive subsystems that are loosely interconnected. Hence, *MQ* is designed to reward the creation of highly cohesive clusters that are not coupled excessively. Our premise is that the larger the *MQ*, the closer the partition of the *MDG* is to the desired subsystem structure. It should be noted that criteria other than the ones that *MQ* is based on can be used to create different views of the system structure.

A naive algorithm for finding the optimal partition of an *MDG* is to enumerate through all of its partitions and select the partition with the largest *MQ* value. This algorithm is not practical for most *MDG*s (i.e., systems with more than 15 modules), because the number of partitions of a graph grows exponentially with respect to its number of nodes [13]. Thus, our clustering algorithms use heuristic-search techniques [8] to discover acceptable suboptimal results quickly.

Fig. 1 illustrates the process supported by our search-based clustering algorithms. The algorithms start with the *MDG* (lower left), then process a collection of partitioned *MDG*s (center), and finally produce a partitioned *MDG* (lower right) as the clustering result. Before we describe these algorithms, we next present a small example to illustrate how suboptimal clustering results can be useful to software maintainers.

### 2.2 A Small Software Clustering Example

An example *MDG* for a small compiler developed at the University of Toronto is illustrated in Fig. 2. A sample partition of this *MDG*, as created by our clustering tool called Bunch, is shown in Fig. 3. Notice how Bunch automatically created four subsystems to represent the structure of the compiler. Specifically, the subsystems exhibit the code generation, scope management, type checking, and parsing services of the compiler.

Even though this is a small system, the compiler *MDG* can be partitioned in 27,644,437 distinct ways. One run of Bunch's search algorithm produced the result shown in

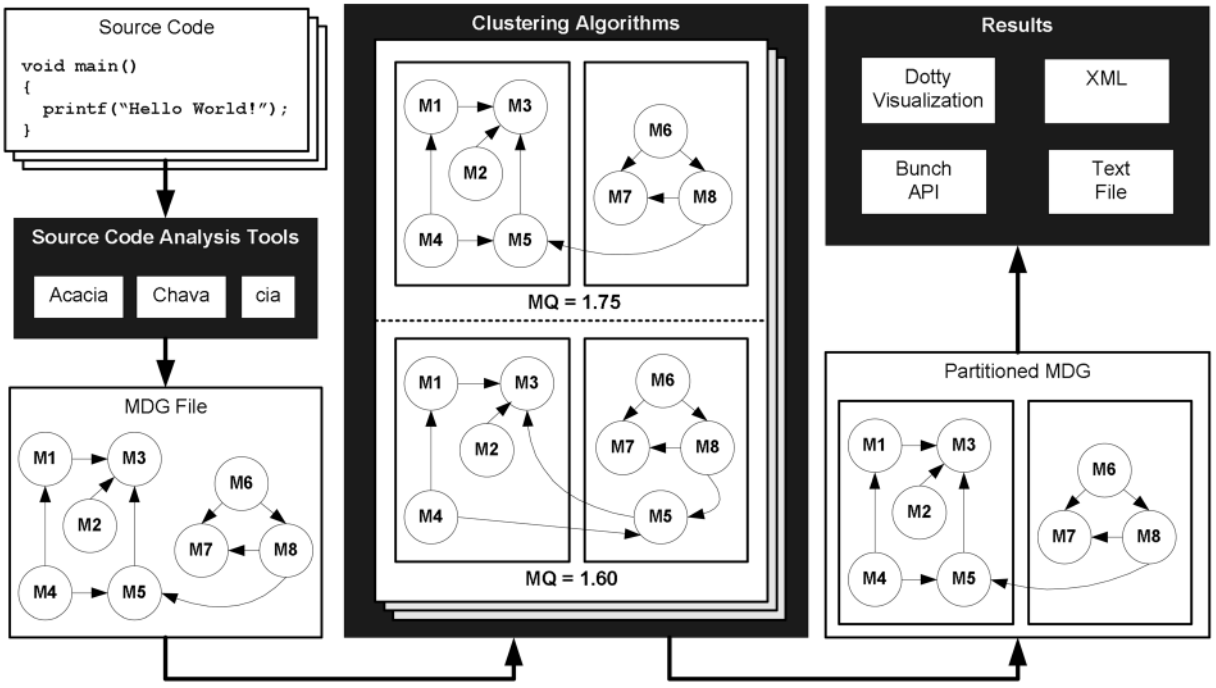


Fig. 1. The software clustering process.

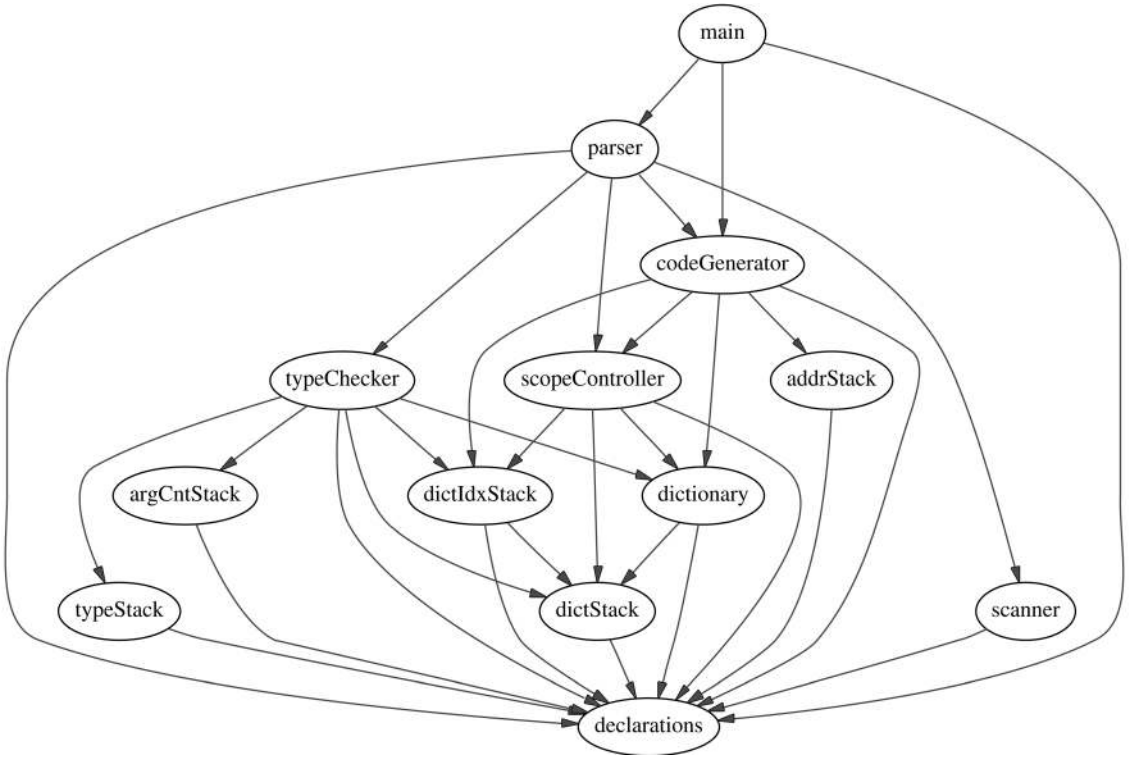


Fig. 2. The *MDG* for a compiler.

Fig. 3 in 0.07 seconds, examining a total of 373 partitions from the search space. An exhaustive search, which executed for 39.57 seconds,[1] validated that the result show in Fig. 3 is optimal. (It has the highest objective function value for this *MDG*.)

The search process used by Bunch selects the initial partition at random. In this example, any one of the valid 27,644,437 partitions is equally likely to be used as the starting point for the search. Since the search space is large, it is unlikely that multiple searches will produce the same result. However, we would like to have confidence that the result produced from one run to the next is similar since the goal is to produce satisfactory suboptimal results quickly. To illustrate this point, we clustered the compiler *MDG* 10 times to observe the variability in the results. Half of the runs produced the result shown in Fig. 3 (the optimal

1. The execution times are based on a 1.7 GHz Pentium M processor.

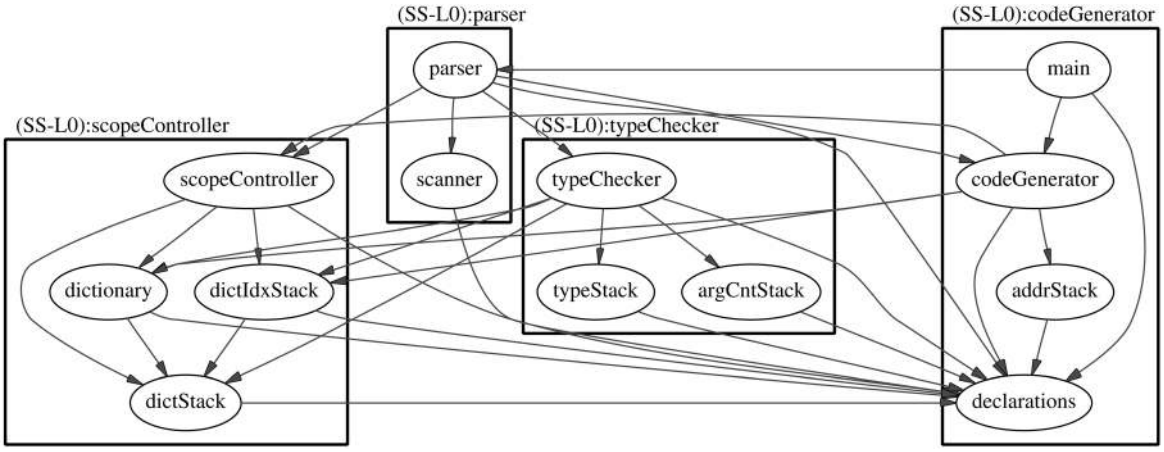


Fig. 3. The partitioned *MDG* for a compiler.

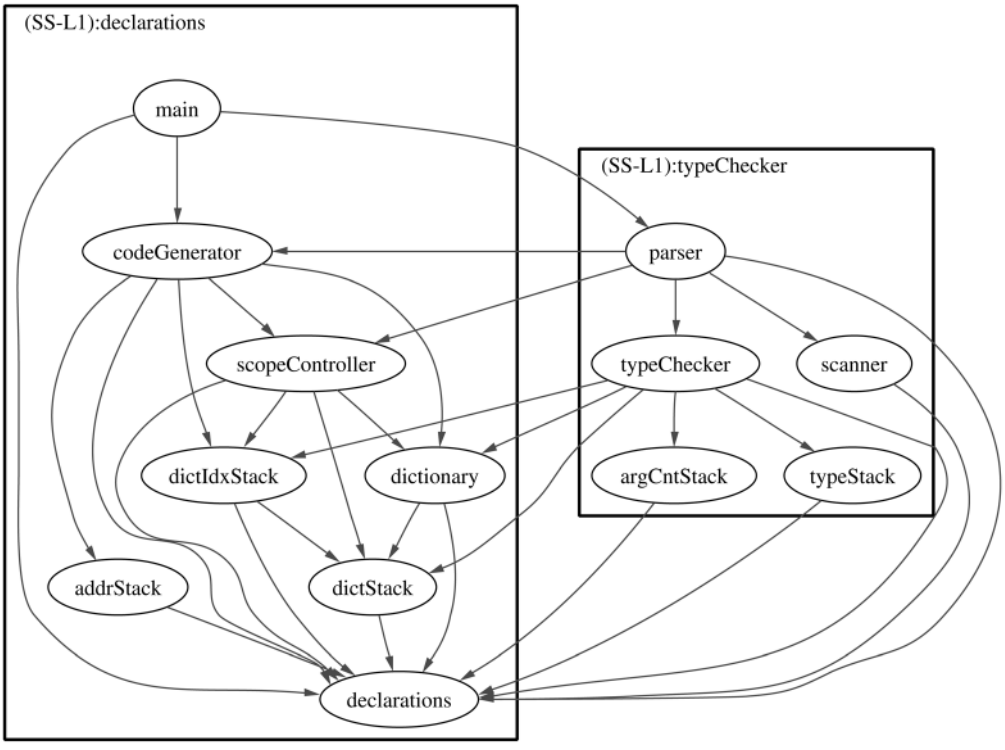


Fig. 4. An alternative partitioned *MDG* for a compiler.

solution) and three of the runs produced the result shown in Fig. 4. This result is very similar to the optimal solution—the two clusters in Fig. 4 are formed by grouping a pair of clusters in Fig. 3. The remaining two results, which are not shown, were also similar, with the main difference being influenced by which cluster contained the `declarations` module, which is highly connected to the other modules in the system.

### 2.3 Measuring MQ

The *MQ* function is designed to balance the tradeoff between the coupling and cohesion of the individual clusters in the partitioned *MDG*. Formally, the *MQ* measurement for an *MDG* partitioned into $k$ clusters is calculated by summing the *Cluster Factor* (*CF*) for each cluster of the partitioned *MDG*. The Cluster Factor, $CF_i$, for cluster $i$ $(1 \leq i \leq k)$ is defined as a normalized ratio between the total weight of the internal edges (edges within the cluster) and half of the total weight of the external edges (edges that exit or enter the cluster). We split the weight of the external edges in half in order to apply an equal penalty to both clusters that are are connected by an external edge.

We refer to the internal edges of a cluster as intraedges ($\mu_i$), and the edges between two distinct clusters $i$ and $j$ as interedges ($\varepsilon_{i,j}$ and $\varepsilon_{j,i}$, respectively). If edge weights are not provided by the *MDG*, we assume that each edge has a weight of 1. Below, we define the *MQ* calculation:

$$MQ = sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise. \end{cases}$$

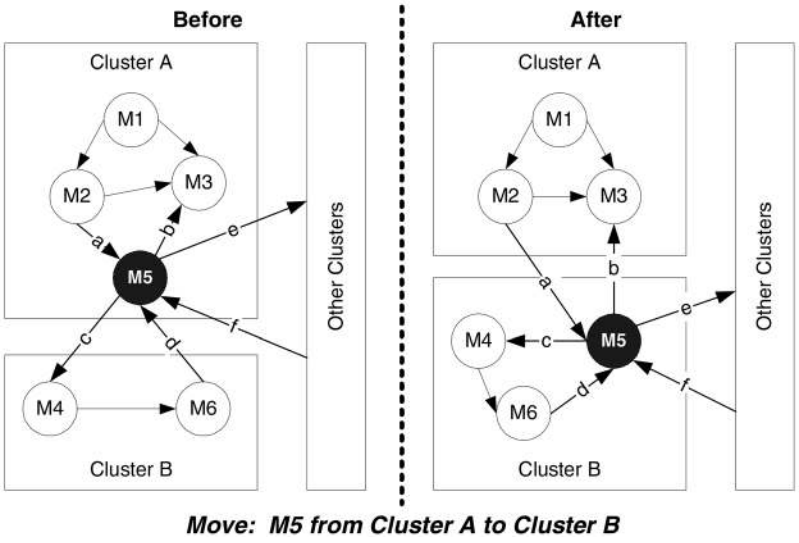


Fig. 5. Two similar partitions of an *MDG*.

### 2.3.1 MQ Performance

The execution complexity of calculating $MQ$ has been shown to be $O(|E|)$, where $|E|$ is proportional to the number of edges in the *MDG* [14]. Applying domain knowledge about our clustering algorithms, we were able to reduce the execution complexity of calculating $MQ$ in practice to $O(\frac{|E|}{|V|})$ [14], where $|V|$ is proportional to the number of modules in the system. This is a dramatic improvement since $|E|$ is closer to $|V|$ than it is to $|V^2|$ for the *MDGs* of most software systems.[2]

The optimization described above is based on the observation that the clustering algorithm used by Bunch improves $MQ$ by continuously moving single modules from one cluster to another. This algorithm is described in Section 4.

An example of a move operation performed by the Bunch clustering algorithm is shown in Fig. 5. Notice that the only difference between the decomposition on the left and the decomposition on the right is the cluster that contains module $M5$. All other nodes and all edges that are not incident to module $M5$ remain unchanged. Thus, the $MQ$ value for the decomposition on the right can be calculated incrementally from the decomposition on the left by only updating $CF_A$ and $CF_B$. Each edge that is incident to module $M5$ can be classified into one of six possible types, labeled $\{a - f\}$ as shown in Fig. 5. Simple update rules can be formulated for each edge type to alter the $\mu$ and $\varepsilon$ weights for $CF_A$ and $CF_B$ [14].

To illustrate the benefit of calculating $MQ$ incrementally, we ran an experiment using the source code from the Java Swing class library, which is provided with Sun's Java Development Kit (JDK) [15]. The first run clustered the class library with the incremental update feature disabled in 31.94 minutes (1,916) seconds. The result contained 24 clusters and the $MQ$ value was 3.89. The second run used the incremental optimization and produced a result in 14.2 seconds. This result also contained 24 clusters and had an $MQ$ value of 4.02. It should also be noted that the first run with the incremental update feature disabled found a solution by evaluating 4.5 million $MQ$ evaluations, while the second run required the calculation of 11.7 million $MQ$ calculations. The difference in the number of required $MQ$ evaluations is not surprising since both runs had a different starting population.

2. A fully connected software graph, which is unlikely to appear in practice, actually has $\frac{|V|(|V|-1)}{2}$ edges since reflexive edges are assumed to be encapsulated inside of the individual modules.

## 3 The Bunch Tool

The Bunch clustering tool was designed to support an ongoing research project; therefore, it needed to satisfy the following requirements:

- **Flexibility.** The source code for our tool is changed often, by different people. Because Bunch is a tool that supports research, changes need to be incorporated into the source code quickly.
- **Installation and Portability.** Bunch was intended to be distributed to students, researchers, and software developers. It is important that our tool is easy to install and can be executed on a variety of operating systems.
- **Performance.** Execution speed is important so that Bunch can be used to cluster large systems.
- **Integration and Extension.** Our research emphasis is on designing and implementing clustering algorithms. We wanted to integrate tools from other researchers into Bunch for related activities, such as source code analysis and visualization.

Fig. 6 highlights how the Bunch tool (center) relates to the overall clustering environment and satisfies the requirements outlined above. Bunch is designed more like a framework than an application. For example, the clustering algorithms themselves are integrated into Bunch via the extension interface (bottom). Although we use this interface to enhance our own clustering algorithms, this design feature also allows other researchers to integrate their own algorithms and objective functions into Bunch. The extension interface supported the creation of many of the features that are described in the subsequent sections of this paper. Another important capability of the Bunch framework is its application programming interface (API). We used the API

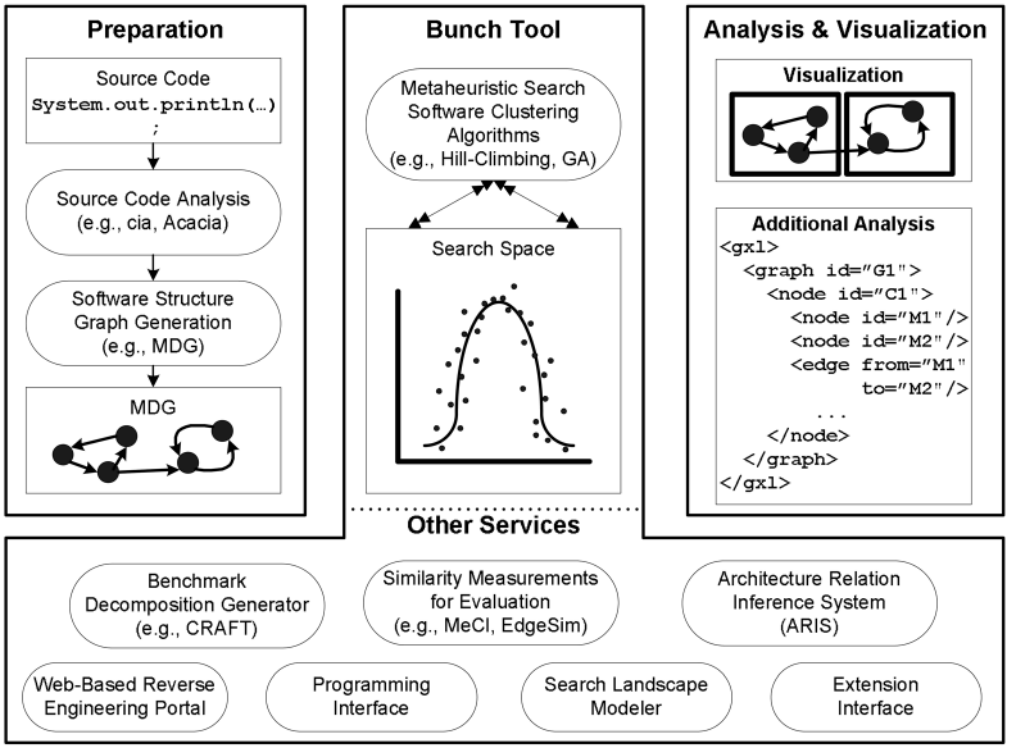


Fig. 6. The Bunch clustering environment.

to integrate Bunch into other tools that we designed to assist software engineers, such as the REportal Web-based reverse engineering portal [16], and a style-specific architecture recovery system [17], [18]. Bunch is developed in Java and has been tested and used on the Windows, Solaris, and Linux platforms.

## 4 Clustering Algorithm

This section examines one of the clustering algorithms that is integrated into the Bunch clustering tool. Bunch supports a hill-climbing algorithm, an exhaustive clustering algorithm that only works well for small systems [14], and a genetic algorithm [19]. We have found the hill-climbing algorithm to work best for most systems and have identified several enhancements that we plan to make to the genetic algorithm in the future [14].

### 4.1 The Hill-Climbing Clustering Algorithm

This section describes a family of hill-climbing search algorithms that have been implemented in Bunch. All of Bunch's hill-climbing clustering algorithms start with a random partition of the *MDG*. Modules from this partition are then systematically rearranged in an attempt to find an improved partition with a higher *MQ*. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. This hill-climbing approach eventually converges when no partitions can be found with a higher *MQ*.

As with traditional hill-climbing algorithms, each randomly generated initial partition of the *MDG* eventually converges to a local maximum. Unfortunately, not every initial partition of the *MDG* produces a satisfactory suboptimal solution. We address this problem by creating an initial *population* (i.e., collection) of random partitions. Our hill-climbing algorithm clusters each of the random partitions in the population and selects the result with the largest *MQ* as the suboptimal solution. As the size of the population increases, the probability of finding a good suboptimal solution also increases.

#### 4.1.1 Neighboring Partitions

Bunch's hill-climbing algorithms move modules between the clusters of a partition in an attempt to improve *MQ*. This task is accomplished by generating a set of neighboring partitions (*NP*).

We define a partition *NP* to be a *neighbor* of a partition *P* 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*. A side effect of creating a neighboring partition is the potential for creating or removing a cluster. In the former case, a node may be moved from one cluster to a new (singleton) cluster. In the latter case, a singleton cluster will be eliminated when its only node is moved into another cluster. Fig. 7 illustrates an example partition, and all of its neighboring partitions.

n other automated software modularization techniques [20], a poor early module movement can bias the final results in a negative way because there is no mechanism for moving a module once it has been placed into a cluster. A useful property of Bunch's neighboring partition strategy is that the approach used to assign a module to a cluster is not necessarily permanent. Rather, future iterations of the clustering algorithm may move a module multiple times, to different clusters.

#### 4.1.2 Simulated Annealing

A well-known problem of hill-climbing algorithms is that certain initial starting points may converge to poor solutions (i.e., local optima). To address this problem, the Bunch hill-climbing algorithm does not rely on a single

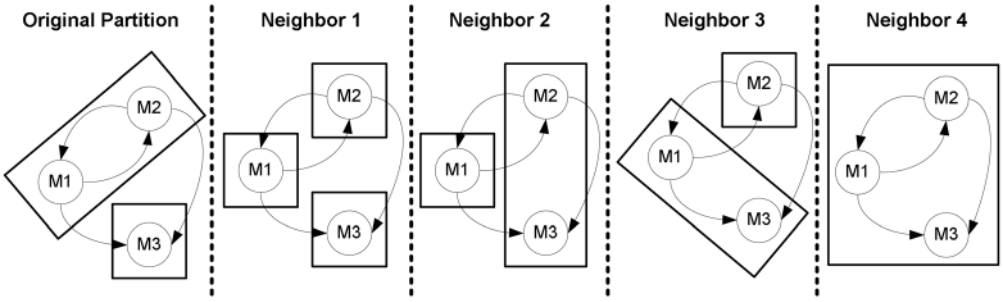


Fig. 7. Neighboring partitions.

random starting point but, instead, uses a collection of random starting points. Bunch also allows the user to alter the hill-climbing algorithm behavior by enabling a simulated annealing [21] feature.

When applied to optimization problems, simulated annealing enables the search algorithm to accept, with some probability, a worse variation as the new solution of the current iteration. As the computation proceeds, the probability of accepting a partition of the *MDG* with a lower *MQ* as the basis for the next iteration of the hill-climbing process diminishes.

#### 4.1.3 Calibrating the Clustering Algorithm

Bunch's hill-climbing algorithm uses a threshold $\eta$ ($0\% \leq \eta \leq 100\%$) to calculate the minimum number of neighbors that must be considered during each iteration of the hill-climbing process. A low value for $\eta$ results in the algorithm taking more small steps prior to converging, and a high value for $\eta$ results in the algorithm taking fewer large steps prior to converging. Our experience has shown that examining many neighbors during each iteration (i.e., using a large threshold, such as $\eta \geq 75\%$) increases the time the algorithm needs to converge to a solution [22]. The increase in execution time, however, is a tradeoff in that a higher $\eta$ value causes the clustering algorithm to explore more of the search space prior to selecting a partition of the *MDG* as the basis of the next iteration of the search. Examining more of the search space increases the probability of finding a better solution.

### 4.2 Other Features of Bunch

Bunch includes features that preprocess the *MDG* to extract special modules that might bias the clustering results in a negative way. Bunch can automatically detect and remove *omnipresent modules*, which are modules that have an unusually large number of connections to the other modules in the system. According to Müller et al. [3], omnipresent modules obscure important underlying structural features and should be removed, along with all of their incident dependencies, prior to the clustering process. Bunch also can detect and remove *library modules* from the *MDG* prior to clustering. These modules have a large number of in-degree edges and no out-degree edges. All of the special modules are collected into special clusters, and the user has the option to show or hide the edges from the *MDG* that are incident to these modules.

Although automatic clustering with Bunch has been shown to produce valuable information, some information about a system design typically exists; thus, we wanted to integrate this knowledge into the automatic clustering process. To accomplish this goal, Bunch allows users to specify sets of modules that belong to the same cluster. Bunch respects the information provided by the user and then focuses on clustering the remaining modules.

We have found these additional features helpful when we work with Bunch users on their own systems. Our position is that design recovery is an incremental process where users first use the automatic clustering capabilities of Bunch to establish a baseline result. This result can then be further analyzed and updated by the software engineer since not all modules belong to clusters based solely on cohesion and coupling criteria. The recommended process for using Bunch is outlined in Section 5.

## 5 Case Study

This section describes a case study where Bunch was used to analyze the structure of a software system. It is worth noting that we had access to one of the primary developers of the software that was analyzed in this case study. In particular, we used Bunch on two consecutive versions of the graph drawing tool *dot* [23]. (Note that *dot* was used to draw Figs. 8, 9, and 10.)

In what follows, it is useful to have some background on the *dot* program and its structure. The *dot* system is based on a pipe-and-filter architecture [24] where a series of processing steps are applied to lay out and draw a graph. The program starts by reading in a description of a directed graph and, based on user options as well as the structure and attributes of the graph, it draws the graph using an automatic layout algorithm. The automatic graph layout technique is a pipeline of graph transformations, through which the graph is filtered, in which each step adds more-specific layout information: First, cycles in the graph are broken. If necessary, information about node clusters is added. Nodes are then optimally assigned to discrete levels. Edges are routed to reduce edge crossings. Nodes are then assigned positions to shorten the total length of all the edges. Finally, edges are specified as Bezier [25] curves. Once the layout is done, *dot* writes to a file using a user-specified output format, such as PostScript or GIF.

The model we employed to create the *MDG* of *dot* uses files as modules and defines a directed edge between two files if the code in one file refers to a type, variable, function, or macro defined in the other file. To generate the *MDG*, we first used the Acacia [10] system to create a source code database for *dot*. Then, using Acacia and standard Unix tools, we generated the MDG for input to Bunch. Fig. 8 presents the original MDG for the first version of *dot*. Little of the structure of *dot* is evident from this diagram.

When we apply Bunch to this *MDG*, we arrive at the clustering shown in Fig. 9. The partition shown is reasonably close to the software structure described above. Cluster 1 contains most of the layout algorithm. Cluster 2

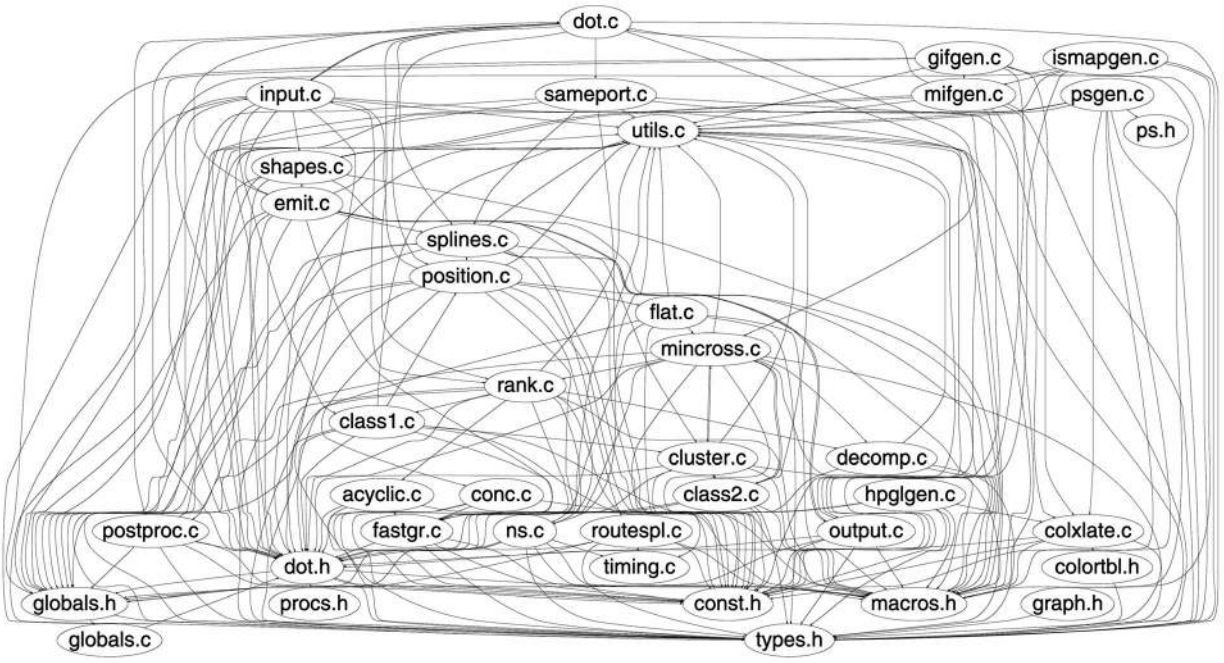


Fig. 8. The Module Dependency Graph (MDG) of *dot*.

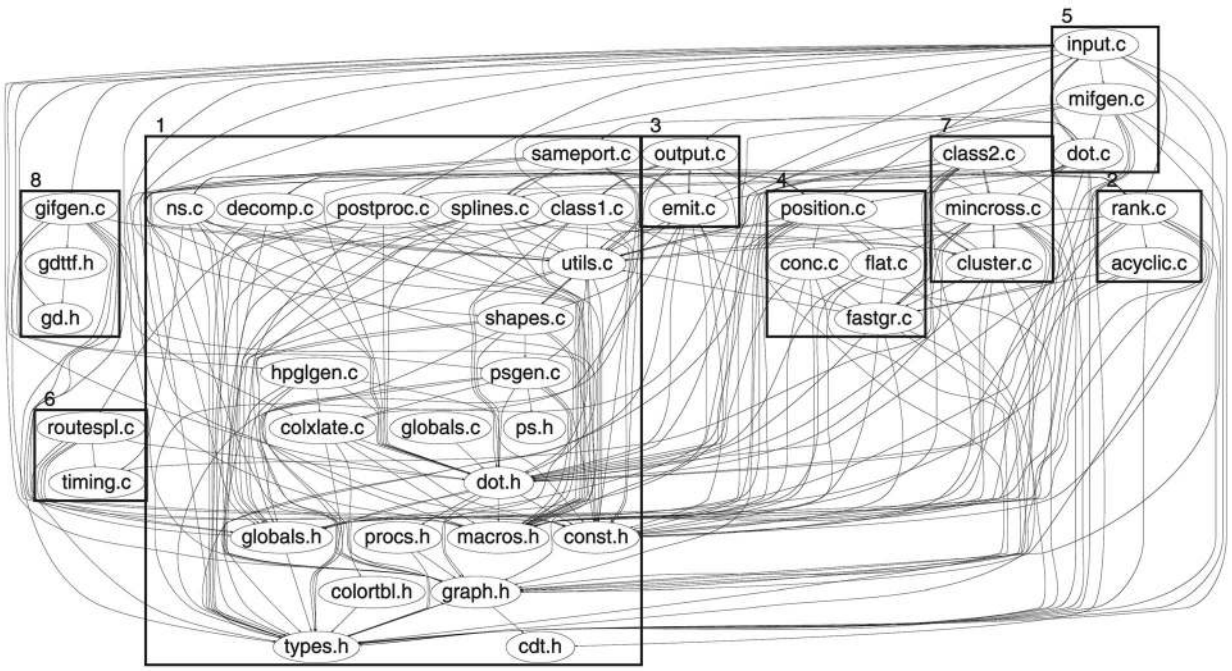


Fig. 9. The automatically produced MDG partition of *dot*.

handles the initial processing of the graph. Cluster 3 captures the output framework. Cluster 5 represents the `main` function and program input. Cluster 7 contains modules for drawing composite nodes (clusters) and performing edge-crossing minimization. Finally, Cluster 8 concerns GIF-related output.

However, there are some anomalies in the partition shown in Fig. 9. Knowing the program, it is hard to make sense of Clusters 4 and 6, or why the module `mifgen.c` is put into Cluster 5. This latter observation, in fact, exposed a flaw in the program structure. There is a global variable defined in `dot.c` that is used to represent the version of the software. This information is passed to each of the output drivers as a parameter; the driver in `mifgen.c`, however, accesses the variable directly rather than using the value passed to it. This explains why `mifgen.c` finds itself in the same subsystem as `dot.c`.

In addition, we note very large fan-ins for some of the modules in Cluster 1, which greatly adds to the clutter of the graph. Using Bunch's omnipresent module calculator, we see that these modules are identified as potential omnipresent modules. Based on program semantics, we recognized these modules as *include* files that define data types that are common to many of the other program

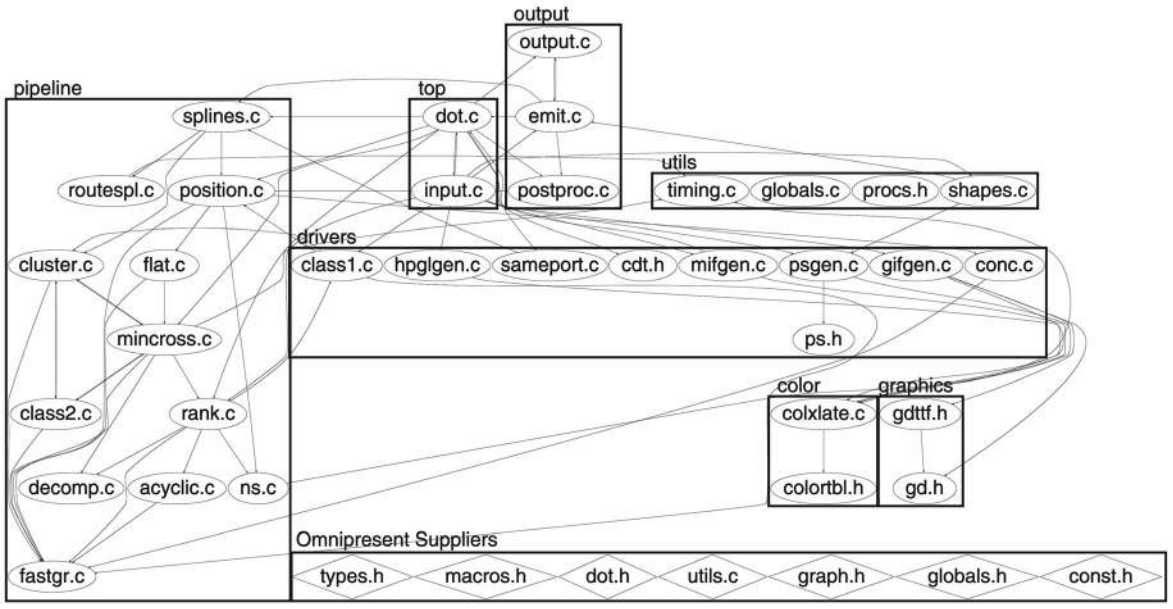


Fig. 10. The *dot* partition (with user-defined cluster names) after the omnipresent modules have been identified and isolated.

modules. It, therefore, makes sense to instruct Bunch to consider these modules as omnipresent suppliers, which are treated separately for the purposes of cluster analysis. We also instruct Bunch not to draw the edges to these omnipresent suppliers, in order to simplify the graph.

After the omnipresent modules are extracted, the next step is to use the Bunch user-directed clustering mechanism to fine-tune the cluster structure by "locking in" certain cluster associations. The designer of the system relocated a few modules from the result that Bunch produced automatically and also provided meaningful names for the subsystems. The resulting clustered graph is shown in Fig. 10.

## 6 Evaluation

The approaches used to evaluate software clustering tools fall into two major categories. Qualitative evaluation approaches often involve soliciting feedback on the clustering results from one or more of the system developers [26], [27]. Quantitative evaluation approaches measure the distance from the results produced by a software clustering algorithm to an authoritative reference solution [28]. Bunch has been evaluated using both of these approaches.

### 6.1 Qualitative Evaluation of Bunch

The results produced by Bunch have been qualitatively evaluated for several popular systems for which we had direct access to the developers. Specifically, we reviewed the results of the recovered structure of the Korn shell program with David Korn and the structure of the dotty graph drawing program (Section 5) with one of its authors, Gansner [27]. We also integrated Bunch's clustering engine into the Enterprise Navigator tool [29], which is used at AT&T to visualize and analyze the interactions between their enterprise information systems. The experts provided positive feedback on the quality of the structural views produced by Bunch. To illustrate this type of evaluation further, the remainder of this section presents a case study where the results of Bunch were reviewed by an expert on a proprietary file system developed by AT&T research.

Fig. 11 shows the *MDG* 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 relation between program entities in the two corresponding source modules (C++ source files). For example, the edge between `oosfid.c` and `nos.h` represents 19 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 [10] parsed the program and detected 463 C++ program entities and 941 relations between them. Note that containment relations between classes/structs and their members are excluded from consideration in the construction of the *MDG*.

Even with the *MDG*, it is not clear what the major components of this system are. Bunch produced the results shown in Fig. 12 containing 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:

- 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.
- The clustering tool is effective in placing 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.
- On the other hand, just by looking at the module names, a designer might 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 old file system.

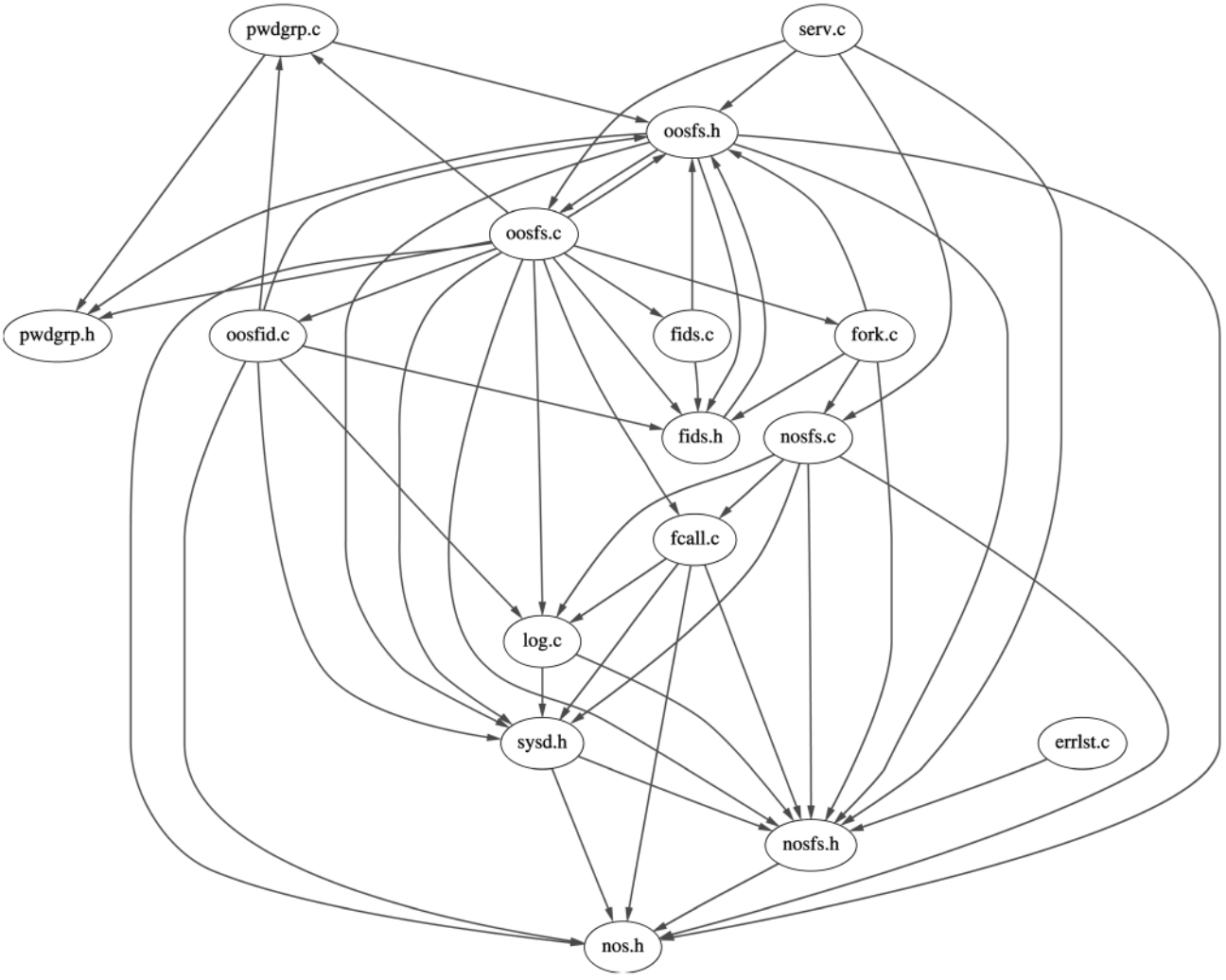


Fig. 11. Module dependency graph of the file system.

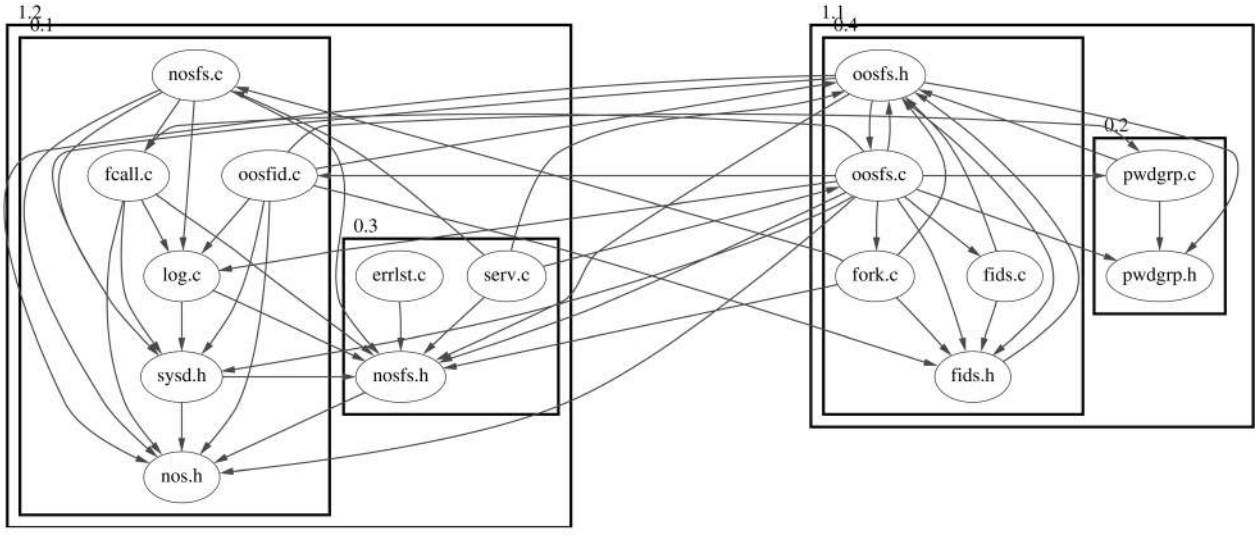


Fig. 12. Clustered view of the file system from Fig. 11.

- It is not obvious why a small cluster, consisting of `errlst.c`, `erv.c`, and `nosfs.h`, was created on the left. It appears that it would have been better to merge that small cluster with its neighbor cluster. In examining the topology of the clustered *MDG* (Fig. 12), Bunch probably created this additional cluster because the interface file `nosfs.h` is used by many modules in both the old and new file systems. Merging this cluster with its neighbor would increase its coupling to the new file system. It

should be noted that the clustering algorithm did, however, eventually combine the two clusters on the left into a single cluster. This observation also shows one of the benefits of creating a hierarchy of clusters, namely, that a user can examine a partitioned *MDG* at varying levels of detail.

### 6.2 Quantitative Evaluation of the Stability of Bunch's Clustering Algorithm

We define a clustering algorithm to be stable if it consistently produces results that are close to a local optimum. We distinguish algorithm stability from the orthogonal concept of solution stability, which is related to the number of distinct solutions that are close to the global optimum. An algorithm that exhibits an unstable behavior is not indicative of an unstable solution to the problem. Likewise, an algorithm that exhibits a stable behavior is not indicative of a stable solution.

In this section, we conduct a quantitative evaluation of the stability of the Bunch algorithm. Quantitative evaluation approaches of software clustering results often require a benchmark standard (i.e., reference decomposition) and a distance measurement. The distance measurement determines the similarity (or dissimilarity) between the result produced by the clustering algorithm and the benchmark standard. Note that the benchmark need not be the optimal solution in a theoretical sense. Rather, it is a solution that is perceived by several people to be satisfactory. Wen, Tzerpos, and Andritsos used this evaluation approach for a number of clustering algorithms (including Bunch) on the Linux [28] and TOBEY[3] systems. They used the MoJo [30], [31], [32] measurement to determine the similarity of the results produced by several clustering algorithms to the benchmark. MoJo measures similarity between two clusters by calculating the minimum number of *module move* and *cluster join* operations required to transform one cluster into the other. Bunch performed well in their case studies.

Evaluating software clustering results against a benchmark is useful; however, this kind of evaluation is not always possible, because benchmarks are often not documented, and the developers are not always accessible for consultation. To address this issue, we wanted to explore if the variations in the clustering results produced by Bunch's heuristic search algorithms could be useful for evaluation. When Bunch is run against a system many times, we observe that certain modules appear in the same clusters constantly (high similarity), certain modules appear in different clusters constantly (high dissimilarity), and a small number of modules tend to drift between different clusters. This observation led us to believe that we could generate an approximation to the de facto reference decomposition in the cases where one does not exist. The underlying assumption is that if a large sample of clustering results agree on certain clusters, these clusters become part of the generated reference decomposition.

The first step in this evaluation process is to cluster a system many times and record the result of each clustering run. Let $\mathcal{S} = \{M_1, M_2, \ldots M_n\}$ be the set of the modules from the system being clustered. For each distinct pair of modules $(M_i, M_j)$, where $1 \leq i, j \leq |\mathcal{S}|$ and $M_i \neq M_j$, we calculate the frequency $(\alpha_{i,j})$ that this pair appears in the same cluster. Given that the system was clustered $\Pi$ times, the number of times that a pair of modules can appear in the same cluster is bounded by $0 \leq \alpha \leq \Pi$. The principle that guides our analysis is that the closer $\alpha$ is to $\Pi$, the more likely that the pair of modules belongs to the same cluster in the generated reference decomposition. An $\alpha$ that is closer to 0 is an indicator of high dissimilarity, meaning that the pair of modules should appear in different clusters in the reference decomposition.

After the clustering step, the set $\mathcal{D}$ is constructed containing all of the $\{M_i, M_j, \alpha_{i,j}\}$ triples sorted in decreasing order by $\alpha_{i,j}$. The next step in forming the generated reference decomposition involves creating an initial cluster by taking the relation from $\mathcal{D}$ with the largest $\alpha$ value. The closure of the modules in this relation is then calculated and new modules are added to the cluster by searching for additional relations in $\mathcal{D}$ such that one of the modules in the triple is already contained in the newly formed cluster, and the $\alpha$ value exceeds a user defined threshold $\beta$ ($0\% \leq \beta \leq 100\%$). The $\beta$ threshold balances the size and number of clusters in the reference decomposition. A $\beta$ value of 0 percent will create a single cluster containing all modules in $\mathcal{S}$, while a $\beta$ value of 100 percent will only group modules that appear in the same cluster all of the time. We have found that using a larger $\beta$ value ($\beta \geq 75\%$) works well in creating a balance between the size and number of clusters.

When adding modules to clusters in the generated reference decomposition, care is taken so that each module in set $\mathcal{S}$ is assigned to no more than one cluster; otherwise, we would not have a valid partition of the *MDG*. After the closure of the cluster is calculated, a new cluster is created and the process repeats. Eventually, all modules that appear in relations in $\mathcal{D}$ that have an $\alpha$ value that exceeds the user defined threshold $\beta$ will be assigned to a cluster. In some cases, however, there may be modules for which no relation in $\mathcal{D}$ exists with a large enough $\alpha$ value to be assigned to a cluster. In these instances, for each remaining unassigned module in $\mathcal{S}$, a new cluster is created containing a single module. This condition is not an indication that the module belongs to a singleton cluster but, instead, is meant to indicate that the module is somewhat unstable in its placement as it tends not to get assigned to any particular cluster.

From an evaluation perspective, we also wanted to measure the similarity of the individual clustering results to the reference decomposition. If the similarity of the individual runs varied dramatically, then Bunch users would not have confidence in an individual clustering run. We calculated the the average, standard deviation, minimum, and maximum values using two different similarity measurements: EdgeSim and MeCl [3].

The EdgeSim similarity measurement normalizes the number of intra and intercluster edges that agree between two partitions. The MeCl similarity measurement determines the distance between a pair of partitions by first calculating the **Cl**umps, which are the largest subset of modules from both partitions that agree with respect to their placement into clusters. Once the "clumps" are determined, a series of **Me**rge operations are performed to convert the first partition into the second one. The actual MeCl distance is determined by normalizing the number of Merge operations [33].

To illustrate this approach for evaluating Bunch clustering results, we analyze the Java Swing [34] class library. The Swing library consists of 413 classes that have

3. The TOBEY system is a proprietary compiler optimization back-end developed by IBM.

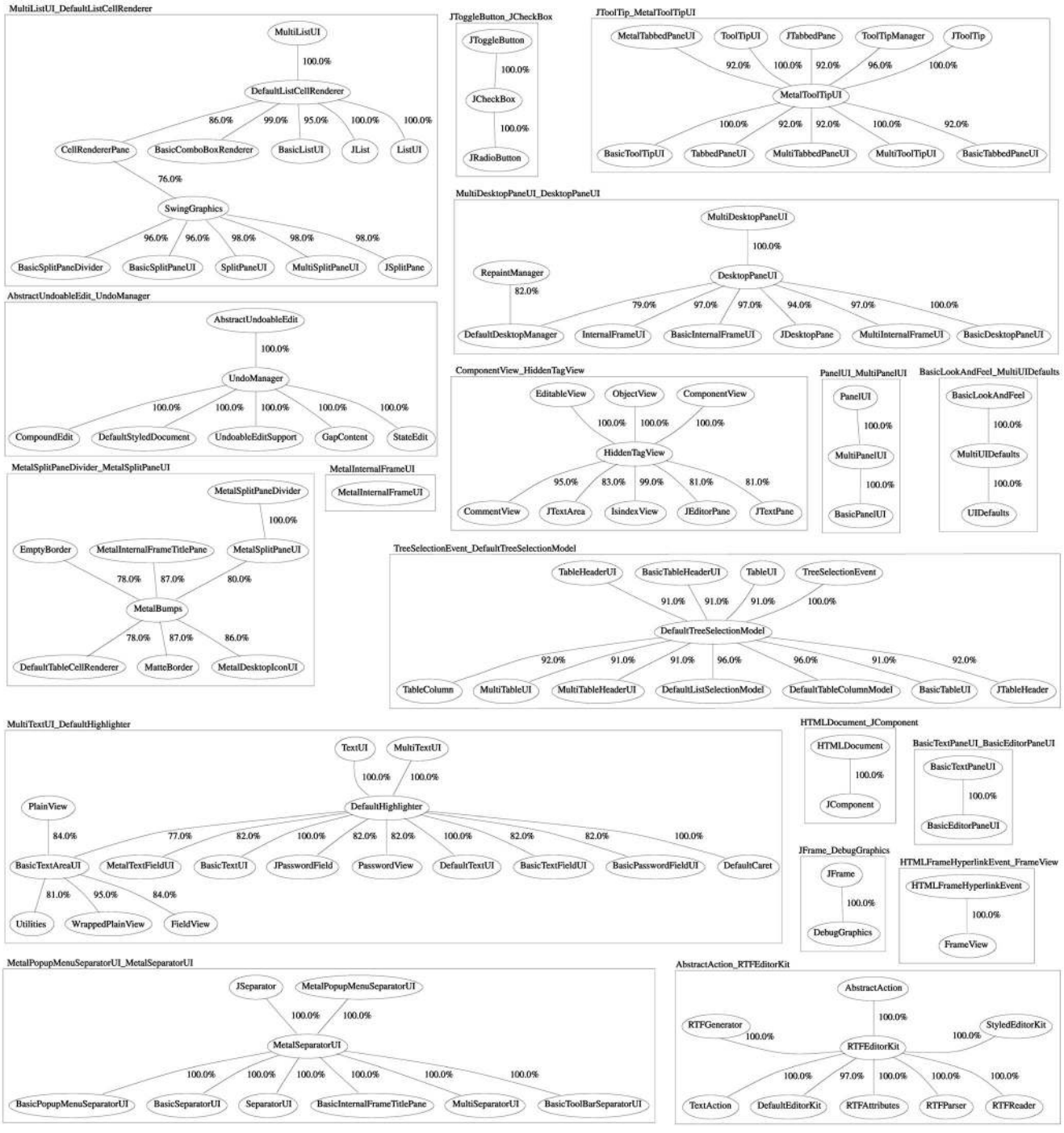


Fig. 13. A partial view of swing's reference decomposition.

1,513 dependencies between them. A partial view[4] of the result produced by using Bunch to generate a reference decomposition is illustrated in Fig. 13. The nodes represent the classes in the Swing framework and the edges show the $\alpha$ value for the relation between the nodes. The Swing system was clustered 100 times and the default $\beta$ value of 75 percent was used to create the visualization.

Table 1 details how the results of the individual clustering runs compare to the generated reference decomposition. Since the average of both similarity measurements is high, coupled with a low standard deviation, we can conclude that the individual clustering results do not deviate far from the

TABLE 1
Similarity Results for the Swing Class Library

| Similarity Measurement | Avg | Min | Max | Stdev |
|---|---|---|---|---|
| MeCl | 96.5% | 92.1% | 100.0% | 0.17 |
| EdgeSim | 93.1% | 89.3% | 98.6% | 0.80 |

4. The entire view was too large to visualize for layout on a printed sheet of paper.

generated reference decomposition. It should also be noted that the results shown in Fig. 13 are highly consistent with the Java package structure of the Swing library, which provides additional confidence that the individual clustering results produced by Bunch are useful.

It is worth noting that measuring the similarity between the individual clustering runs and the generated reference decomposition can be used to assess the quality of the generated reference decomposition. Recall that the reference decomposition is constructed by grouping modules that appear in the same subsystem across many of the individual clustering runs. If the results produced by the individual clustering runs are stable, this approach will result in a reference decomposition that is similar to each of the individual clustering runs. If the results produced by the individual clustering runs vary significantly, then the generated reference decomposition will contain many small clusters and will not be similar to the individual clustering results. Thus, by determining the average similarity between the reference decomposition and the individual clustering runs, we can assess if the reference decomposition is representative of the results produced by clustering algorithms. Namely, a high similarity would indicate that the reference decomposition represents a useful solution, where a low similarity would suggest the opposite.

### 6.3 Evaluation Summary

Evaluation based on both qualitative and quantitative methods is useful for gaining confidence that a clustering algorithm produces good results. Both of these evaluation methods require an expert either to give feedback on a result produced by a clustering algorithm or to define a benchmark that can be used to compare to the results produced by a clustering algorithm.

Bunch is valuable because it provides useful views of the software structure that are helpful to practitioners performing maintenance activities. For example, in Section 2.3 we described our *MQ* measurement, which was designed to reward the creation of cohesive clusters. Since *MQ* guides Bunch's search algorithms, the results produced by Bunch provide structural views that embody a generally accepted software engineering principle that well-designed systems are organized into cohesive clusters that are loosely interconnected [35].

Unfortunately, determining the partition of the *MDG* with the largest *MQ* value is only possible using an exhaustive search. However, we have demonstrated elsewhere [36] that we can bound the solutions produced by Bunch within a known factor $\log^2 N$ of the optimal solution.

The views produced by Bunch should not be perceived as the definitive solution to the software clustering problem. We advocate that users follow an incremental approach in which the results produced by Bunch are refined as additional information about the systems structure is discovered. We demonstrated this approach in Section 5 on the *dot* system. It is also important to note that there is not a single way to assign modules into clusters, as some of these modules may be placed better using other criteria (e.g., directory names, file names, package names). The case study in Section 5 demonstrated several automated facilities to automatically identify candidate libraries, drivers, and omnipresent modules [3]. When appropriate, users can bypass the automatic placement of certain modules by assigning them to clusters manually. The Bunch tool also has an extensive API [14] so that users can extend the framework by adding new clustering algorithms, objective functions, simulated annealing cooling schedules, output formats, etc.

The papers and software produced by the Bunch project are widely cited [37], [38], and have been used in advanced software engineering classes. Bunch has been integrated into an industrial-strength software system at AT&T [39] as well as a number of other software engineering tools used by researchers [16], [18], [17]. We also compared the results produced by Bunch to a spectral clustering technique, which enabled us to discover an upper bound for how far the results created by Bunch are from the optimal solution.

## 7 Related Work

Early work by Belady and Evangelisti [40] identified automatic clustering as a means to produce views of the structure of software systems. Much of the initial work on software clustering, like that of Hutchens and Basili [41], focused on techniques for grouping related procedures and variables into modules. Progressively, as software systems began to grow in size, the new problem of grouping sets of modules into hierarchies of subsystems became pertinent.

Schwanke's ARCH tool [20] determined clusters using coupling and cohesion measurements. The Rigi system [42], by Müller et al. pioneered the concepts of isolating omnipresent modules, grouping modules with common clients and suppliers, and grouping modules that had similar names. The last idea was followed up by Anquetil and Lethbridge [43], who used common patterns in file names as a clustering criterion.

Lindig and Snelting [4] proposed a software modularization technique based on mathematical concept analysis. Eisenbarth et al. [44] also used concept analysis to map a system's externally visible behavior to relevant parts of the source code.

Tzerpos and Holt's ACDC clustering algorithm [45] uses patterns that have been shown to have good program comprehension properties to determine the system decomposition. The authors define seven *subsystem patterns* and, then, describe their clustering algorithm that systematically applies these patterns to the software structure. After applying the subsystem patterns of the ACDC clustering algorithm, most, but not all, of the modules are placed into subsystems. ACDC then uses orphan adoption [46] to assign the remaining modules to appropriate subsystems.

Koschke [47] examined 23 different clustering techniques and classified them into connection-, metric-, graph-, and concept-based categories. Of the 23 clustering techniques examined, 16 are fully automatic and seven are semiautomatic. One of Koschke's metric-based algorithms that is closest to our work is called similarity clustering, which is an extension of earlier work done by Schwanke [20]. The challenge with this clustering approach is to calibrate a large number of parameters properly. Koschke used search techniques, including simulated annealing, to optimize the clustering algorithm parameters. Koschke also created a semiautomatic clustering framework based on modified versions of the fully automatic techniques he investigated. The goal of Kochke's framework is to enable a collaborative session between his clustering framework and the user. The clustering algorithm does the processing, and the user validates the results.

Mockus and Weiss [48] identified the value of assigning independent parts of the code base, or "chunks," to

maintenance teams that are globally distributed. They apply a search technique based on simulated annealing to identify candidates for chunksrdquo; of code that can be assigned to different development teams. The premise for their research is that minimizing the need for coordination and synchronization across globally distributed development teams has a measurable positive impact on the quality and development time to implement software changes.

Although most clustering approaches are bottom-up, which produce structural views starting from the source code, some promising top-down approaches have demonstrated their effectiveness for helping with software maintenance problems. Murphy et al. developed Software Reflexion Models [49] to capture and exploit differences that exist between the actual source code organization and the designer's mental model of the system organization.

Bunch includes several features that were inspired by other software clustering research such as orphan adoption [46] and the automatic detection of candidate omnipresent modules [42]. Bunch has also influenced the work of other researchers and distinguishes itself from other software clustering approaches in several ways:

- The Bunch framework provides integrated manual, semiautomatic, and fully automatic clustering facilities because the process of design extraction is often iterative. Users can automatically generate partitioned views of a system's structure and then refine these views using Bunch's semiautomatic clustering features. This process was shown in Section 5.
- Bunch uses search techniques and treats the clustering activity as a graph partitioning problem. Other early work investigating search strategies for software clustering problems was performed by Koschke [47], in which he used simulated annealing to calibrate the parameters necessary to guide his clustering algorithm. More recently, Mahdavi et al. [50] agglomerated results from multiple hill-climbing runs to identify software clusters, an approach similar to the one that we used for evaluating software clustering results [51].
- Most software clustering algorithms are deterministic, and as such, always produce the same result for a given input. Since our approach is based on randomized heuristic search techniques, our clustering algorithms often do not produce the same result, but a family of similar results. We examined some useful properties associated with this behavior to model the search space, investigate the stability of the clustering results [22], [52], and to create reference decompositions [51]. Mahdavi et al. recently investigated this property to improve their genetic clustering algorithm [50]. As mentioned earlier, Mockus and Weiss [48] used simulated annealing to partition the code base ownership to different geographically separated maintenance teams.
- Bunch's algorithms and tools were designed to be fast. We often work on large systems, like Swing [34] and Linux, and are able to produce results in several minutes.
- Bunch is available for download over the Internet [53]. We encourage users to download, use, and extend our tool through its published programming interface. We also integrated Bunch into an online software engineering portal called REportal [16].

Now that a broad range of approaches to software clustering exist, the validation of clustering results is starting to attract the interest of the reverse engineering research community. Early work in this area was performed by Koschke and Eisenbarth [54], who described a quantitative method to evaluate software clustering results. The authors state that a reference decomposition is necessary in order to compare the strengths and weaknesses of individual clustering algorithms. Early work by Tzerpos and Holt [55] used this approach for evaluation, which led to the development of a distance measurement called MoJo to compare clustering results with a reference decomposition. We outlined some problems with the initial version of MoJo and other similarity measurements used to evaluate software clustering results. These measurements only considered the placement of modules into clusters and not the relations between the modules. We proposed two new measurements called EdgeSim and MeCl [33] that use the strength of the module relations to measure similarity. Wen and Tzerpos [31], [32] have since extended MoJo to consider both the placement of modules and the strength of the relations between the modules when measuring the similarity of software clustering results.

## 8 Contribution to the Software Engineering Community

Using Bunch to create views of the system structure directly from its implementation has several useful applications:

- Software engineers who manage maintenance activities often try to identify the points of loose coupling in the system implementation so that work can be distributed to maintainers efficiently. Ensuring a high degree of independence in the code touched by maintainers reduces the need for multideveloper coordination [48]. This approach simplifies activities such as integration testing, placing more emphasis on fixing defects in earlier phases of the software development lifecycle (i.e., during unit testing).
- The clustering results produced by Bunch can be used by project managers and enterprise planners who manage software maintenance activities or are responsible for a large portfolio of integrated systems. For example, we integrated Bunch's clustering engine into the Enterprise Navigator tool [29], which is used at AT&T to visualize and analyze the interactions between their enterprise information systems. Specifically, Bunch was used to partition a large graph where the nodes represented information systems and the edges represented data flows between them. The goal of this work was to help planners understand and manage the integration points between a large number of legacy systems.
- The design of the Bunch tool and its associated clustering algorithms has been extensively documented [14]. The Bunch tool, along with the documentation of its programming interface, is available for download [53] over the Internet. Although a significant amount of recent research on clustering focuses on comparing the results of other clustering algorithms to those produced by

Bunch [37], [38], we have also been contacted by researchers and students who are integrating Bunch into their own work. By keeping the programming interface public and the design of our tool flexible, we hope that others will not only continue to integrate Bunch into their own work but also to extend Bunch by creating new clustering algorithms and objective functions. Since Bunch is designed to be extended quickly, the Bunch framework can be used to expedite new research into software clustering algorithms and alternative objective functions [56].

## 9 Conclusions

The Bunch framework was designed to help maintainers who are trying to understand large and complex software systems. Using search techniques, Bunch partitions the structure of a software system using the entities and relations that are specified in the source code. Bunch is fast and scalable, as systems like the Linux kernel (10,000 modules and 100,000 relations) can be clustered in about 30 seconds [33]. Over the past several years, we have applied Bunch to over 50 systems, obtained from both industry and the open source community. These systems range in size from a few thousand lines of code to upwards of a million lines of code.

This paper places a significant emphasis on how Bunch can help software engineers to perform a variety of program understanding and maintenance activities. It also illustrates how the results produced by Bunch can be evaluated, since software engineers must have confidence in the tools that they use to analyze systems. This paper explains how Bunch was evaluated using qualitative and quantitative methods.

In addition to the clustering algorithms described in this paper, Bunch was designed so that it can be extended to include other clustering algorithms, *MQ* measurement functions, and simulated annealing cooling schedules. Bunch also includes an API that makes it easy to integrate its clustering services into other software engineering tools.

## Acknowledgments

The authors would like to thank the US National Science Foundation (grant numbers CCR-9733569 and CISE-9986105), AT&T, and Sun Microsystems for their generous support of this research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, US government, AT&T, or Sun Microsystems. We would also like to thank Emden Gansner and Yih-Farn Chen for their help with the qualitative evaluation of Bunch.

## References

[1] R. Schwanke and S. Hanson, "Using Neural Networks to Modularize Software," *Machine Learning,* vol. 15, pp. 137-168, 1998.

[2] S. Choi and W. Scacchi, "Extracting and Restructuring the Design of Large Systems," *IEEE Software,* pp. 66-71, 1999.

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

[4] C. Lindig and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis," *Proc. Int'l Conf. on Software Eng.,* May 1997.

[5] A.V. Deursen and T. Kuipers, "Identifying Objects Using Cluster and Concept Analysis," *Int'l Conf. Software Eng.,* pp. 246-255, May 1999, http://citeseer.nj.nec.com/vandeursen99building.html.

[6] N. Anquetil, "A Comparison of Graphis of Concept for Reverse Eng.," *Proc. Int'l Workshop Program Comprehension,* June 2000.

[7] N. Anquetil and T. Lethbridge, "Recovering Software Architecture from the Names of Source Files," *Proc. Working Conf. Reverse Eng.,* Oct. 1999.

[8] J. Clark, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B.S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating Software Engineering as a Search Problem," *J. IEE Proc. Software,* vol. 150, no. 3, pp. 161-175, 2003.

[9] Y. Chen, "Reverse Engineering," *Practical Reusable Unix Software,* chapter 6, pp. 177-208, B. Krishnamurthy, ed. New York: John Wiley & Sons, 1995.

[10] Y. Chen, E. Gansner, and E. Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection," *Proc. Sixth European Software Eng. Conf. and Fifth ACM SIGSOFT Symp. Foundations of Software Eng.,* Sept. 1997.

[11] J. Korn, Y. Chen, and E. Koutsofios, "Chava: Reverse Engineering and Tracking of Java Applets," *Proc. Working Conf. Reverse Eng.,* Oct. 1999.

[12] M. Salah and S. Mancoridis, "Reverse Engineering of a Hierarchy of Dynamic Software Views: From Object Interactions to Feature Dependencies," *Proc. IEEE Int'l Conf. Software Maintenance,* Sept. 2004.

[13] A. Nijenhuis and H.S. Wilf, *Combinatorial Algorithms,* second ed. Academic Press, 1978.

[14] B.S. Mitchell, "A Heuristic Search Approach to Solving the Software Clustering Problem," PhD dissertation, Drexel Univ., 2002.

[15] "The Sun Developer Network," Javasoft, http://www.Javasoft.com, **YEAR?**.

[16] S. Mancoridis, T. Souder, Y. Chen, E.R. Gansner,, and J.L. Korn, "REportal: A Web-Based Portal Site for Reverse Engineering," *Proc. Working Conf. Reverse Eng.,* Oct. 2001.

[17] B.S. Mitchell and S. Mancoridis, "Using Interconnection Style Rules to Infer Software Architecture Relations," *Proc. Genetic and Evolutionary Computation Conf.,* 2004.

[18] M. Traverso and S. Mancoridis, "On the Automatic Recovery of Style-Specific Structural Dependencies in Software Systems," *J. Automated Software Eng.,* vol. 9, no. 3, 2002.

[19] D. Doval, S. Mancoridis, and B. Mitchell, "Automatic Clustering of Software Systems Using a Genetic Algorithm," *Proc. Software Technology and Eng. Practice,* Aug. 1999.

[20] R. Schwanke, "An Intelligent Tool for Re-Engineering Software Modularity," *Proc. 13th Int'l Conf. Software Eng.,* May 1991.

[21] S. Kirkpatrick, C.D. Gelatt Jr., and M. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, pp. 671-680, May 1983.

[22] B.S. Mitchell and S. Mancoridis, "Using Heuristic Search Techniques to Extract Design Abstractions from Source Code," *Proc. Genetic and Evolutionary Computation Conf.,* July 2002.

[23] E. Gansner, E. Koutsofios, S. North, and K. Vo, "A Technique for Drawing Directed Graphs," *IEEE Trans. Software Eng.,* vol. 19, no. 3, pp. 214-230, Mar. 1993.

[24] M. Shaw and D. Garlan, *Software Architecture: Perspectives on An Emerging Discipline.* Prentice-Hall, 1996.

[25] J.D. Foley, A. Van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics,* second ed. Addison-Wesley, 1990.

[26] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," *Proc. Sixth Int'l Workshop Program Comprehension,* June 1998.

[27] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *Proc. Int'l Conf. Software Maintenance,* pp. 50-59, Aug. 1999.

[28] I.T. Bowman, R.C. Holt, and N.V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture," *Proc. Int'l Conf. Software Eng.,* May 1999.

[29] A. Buchsbaum, Y.-F. Chen, H. Huang, M. Jankowsky, E. Koutsofos, S. Mancoridis, J. Mocenigo, and A. Rogers, "Enterprise Navigator: A System for Visualizing and Analyzing Software Infrastructures," *IEEE Software,* vol. 18, no. 5, 2001.

[30] P. Andritsos and V. Tzerpos, "Software Clustering Based on Information Loss Minimization," *Proc. IEEE Working Conf. Reverse Eng.,* Nov. 2003.

[31] Z. Wen and V. Tzerpos, "An Effectiveness Measure for Software Clustering Algorithms," *Proc. IEEE Int'l Conf. Software Maintenance,* Sept. 2004.
[32] Z. Wen and V. Tzerpos, "Evaluating Similarity Measures for Software Decompositions," *Proc. IEEE Int'l Workshop Program Comprehension,* June 2004.
[33] B.S. Mitchell and S. Mancoridis, "Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements," *Proc. Int'l Conf. Software Maintenance,* Nov. 2001.
[34] "Javasoft Swing Libraries: Java Foundation Classes," Swing, http://www.javasoft.com/products/jfc, **YEAR**.
[35] I. Sommerville, *Software Eng.,* seventh ed. Addison-Wesley, 2004.
[36] A. Shokoufandeh, S. Mancoridis, and M. Maycock, "Applying Spectral Methods to Software Clustering," *Proc. IEEE Working Conf. on Reverse Eng.,* pp. 3-10, Oct. 2002.
[37] "Computer and Information Science Papers Citeseer Publications Research Index," CiteSeer, http://citeseer.ist.psu.edu, **YEAR**.
[38] "Google Scholar Search Engine," http://scholar.google.com, **YEAR**.
[39] A. Buchsbaum, Y.-F. Chen, H. Huang, M. Jankowsky, E. Koutsofios, S. Mancoridis, J. Mocenigo, and A. Rogers, "Enterprise Navigator: A System for Visualizing and Analyzing Software Infrastructures," *IEEE Software,* vol. 18, no. 5, pp. 62-70, 2001.
[40] L.A. Belady and C.J. Evangelisti, "System Partitioning and Its Measure," *J. Systems and Software,* vol. 2, pp. 23-29, 1981.
[41] D. Hutchens and R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Eng.,* vol. 11, pp. 749-757, Aug. 1985.
[42] H. Müller, M. Orgun, S. Tilley, and J. Uhl, "A Reverse Eng. Approach to Subsystem Structure Identification," *J. Software Maintenance: Research and Practice,* vol. 5, pp. 181-204, 1993.
[43] N. Anquetil and T. Lethbridge, "Extracting Concepts from File Names: A New File Clustering Criterion," *Proc. 20th Int'l Conf. Software Eng.,* May 1998.
[44] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis," *Proc. IEEE Int'l Conf. Software Maintenance,* Nov. 2001.
[45] V. Tzerpos and R.C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering," *Proc. Working Conf. Reverse Eng.,* pp. 258-267, Nov. 2000.
[46] V. Tzerpos and R. Holt, "The Orphan Adoption Problem in Architecture Maintenance," *Proc. Working Conf. Reverse Eng.,* Oct. 1997.
[47] R. Koschke, "Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method," PhD dissertation, Univ. of Stuttgart, Stuttgart, Germany, 2000.
[48] A. Mockus and D.M. Weiss, "Globalization by Chunking: A Quantitative Approach," *IEEE Software,* vol. 18, no. 2, pp. 30-37, 2001.
[49] G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng.,* 1995.
[50] K. Mahdavi, M. Harman, and R. Hierons, "A Multiple Hill Climbing Approach to Software Module Clustering," *Proc. IEEE Int'l Conf. Software Maintenance,* Sept. 2003.
[51] B.S. Mitchell and S. Mancoridis, "CRAFT: A Framework for Evaluating Software Clustering Results in the Absence of Benchmark Decompositions," *Proc. Working Conf. Reverse Eng.,* Oct. 2001.
[52] B.S. Mitchell and S. Mancoridis, "Modeling the Search Landscape of Metaheuristic Software Clustering Algorithms," *Proc. Genetic and Evolutionary Computation Conf.,* 2003.
[53] "The Drexel University Software Eng. Research Group (SERG)," http://serg.cs.drexel.edu, **YEAR**.
[54] R. Koschke and T. Eisenbarth, "A Framework for Experimental Evaluation of Clustering Techniques," *Proc. Int'l Workshop Program Comprehension,* June 2000.
[55] V. Tzerpos and R.C. Holt, "MoJo: a Distance Metric for Software Clustering," *Proc. Working Conf. Reverse Eng.,* Oct. 1999.
[56] M. Harman, S. Swift, and K. Mahdavi, "An Empirical Study of the Robustness of Two Module Clustering Fitness Functions," *Proc. Genetic and Evolutionary Computation Conf.,* 2005.

**Brian S. Mitchell** received the ME degree (1995) in computer engineering from Widener University and the MS degree (1997) and the PhD degree (2002) in computer science from Drexel University. He has more than 15 years of technical and leadership experience in the software industry. He is currently a research associate and a member of the Software Engineering Research Group (SERG) at Drexel University. He is also an enterprise architect at CIGNA Corporation in Philadelphia, Pennsylvania, where he works as a lead software engineer on enterprise strategic initiatives. His interests include software architecture, reverse engineering, program understanding, software security, and computer architecture. He is a member of the IEEE and the IEEE Computer Society and has served as a program committee member for several IEEE, ACM, AAAI, and industry conferences. He is a member of the organizing committee for the 2006 IEEE ICSM conference.

**Spiros Mancoridis** received the PhD degree in computer science from the University of Toronto under the supervision of Professor Richard C. Holt. He joined Drexel University in 1996 and is now an associate professor in the Department of Computer Science and the founder and director of the Software Engineering Research Group (SERG) at Drexel University. At SERG, he works with a team of students and researchers on problems related to the reverse engineering of large software systems, program understanding, software testing, and software security. In 1998, he received a Career Award for Young Investigators from the US National Science Foundation. Since then, he has received additional funding from the NSF, DARPA, the US Army, AT&T, and Sun Microsystems. He is the general chair for the 2006 IEEE ICSM Conference in Philadelphia, Pennsylvania, and a senior member of the IEEE Computer Society and the IEEE.