Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures

S. Mancoridis, B. S. Mitchell
Dept. of Mathematics & Computer Science
Drexel University
3141 Chestnut Street
Philadelphia, PA, USA
+1 215 895 6824
e-mail: {smancori, bmitchel}@mcs.drexel.edu

Y. Chen, E. R. Gansner
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ, USA
+1 973 360 8653
e-mail: {chen,erg}@research.att.com

Abstract

Software systems are typically modified in order to extend or change their functionality, improve their performance, port them to different platforms, and so on. For developers, it is crucial to understand the structure of a system before attempting to modify it. The structure of a system, however, may not be apparent to new developers, because the design documentation is non-existent or, worse, inconsistent with the implementation. This problem could be alleviated if developers were somehow able to produce high-level system decomposition descriptions from the low-level structures present in the source code.

We have developed a clustering tool called Bunch that creates a system decomposition automatically by treating clustering as an optimization problem. This paper describes the extensions made to Bunch in response to feedback we received from users. The most important extension, in terms of the quality of results and execution efficiency, is a feature that enables the integration of designer knowledge about the system structure into an otherwise fully automatic clustering process. We use a case study to show how our new features simplified the task of extracting the subsystem structure of a medium size program, while exposing an interesting design flaw in the process.

Keywords: Automatic Clustering, Reverse Engineering, Software Re-engineering, Optimization Algorithms.

1. INTRODUCTION

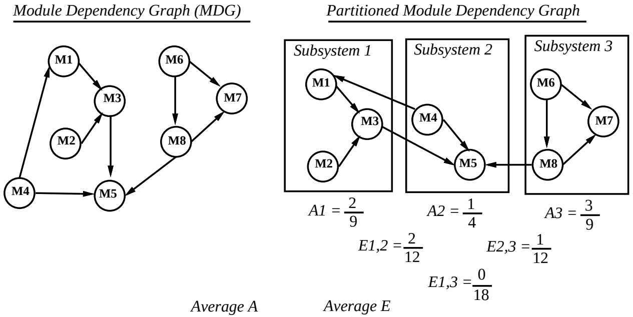
The size and complexity of software systems continues to grow, as developers create increasingly sophisticated applications. The implementation of these applications typically involves a large number of modules (or classes) that are interconnected in intricate ways.

Creating a good mental model of the structure of a complex system, and keeping that mental model consistent with changes that occur as the system evolves, is one of many serious problems that confront today's software developers. This problem is exacerbated by other problems such as inconsistent or non-existent design documentation and a high rate of turnover among information technology professionals.

In our attempt to alleviate some of the problems mentioned above, we have developed an automatic technique to decompose the structure of software systems into meaningful subsystems. Subsystems provide developers with high-level structural information that helps them navigate through the numerous software components, their interfaces, and their interconnections.

In two recent papers [10, 5] we described an automatic technique that treats clustering as an optimization problem that can be solved using hill-climbing [11] and genetic algorithms [8]. This technique is implemented in a tool called *Bunch*. Although the outcome of our first evaluation of Bunch was very encouraging, we felt that a further round of evaluation, using more systems of different sizes, was necessary.

The second round of evaluation confirmed that Bunch was capable of producing good results efficiently,



$$MQ = \frac{\overbrace{\frac{2}{9}+\frac{1}{4}+\frac{3}{9}}^{Average\ A}}{3} - \frac{\overbrace{\frac{2}{12}+\frac{0}{18}+\frac{1}{12}}^{Average\ E}}{3} = \frac{5}{27} = 0.185...$$

Figure 1. Modularization Quality Example

but also exposed some areas for improvement that were not apparent when we wrote our first papers. This follow-up paper presents the features we added to Bunch to address those weaknesses and shows how these features were applied to cluster the modules of the *dot* [7] graph drawing tool.

The structure of the rest of this paper is as follows: Section 2 presents an overview of the Bunch tool. Section 3 describes some of the limitations of Bunch that were identified during the first round of user evaluations. Section 4 presents new features that were added to Bunch based on the feedback we received from the evaluations. Section 5 describes a case study that was conducted in order to evaluate these new features. Section 6 presents an overview of related research in the area of software clustering. Finally, Section 7 summarizes the contributions of our work and outlines our future plans to improve Bunch.

2. THE BUNCH CLUSTERING TOOL

In another paper [10] we describe how Bunch can cluster source-level modules and dependencies into subsystems. Our tool assumes that the modules and dependencies of a system are mapped to a *Module Dependency Graph* (*MDG*). Formally, *MDG*=(M, R) is a graph where M is the set of named modules of a software system, and $R \subseteq M \times M$ is the set of ordered pairs $\langle u, v \rangle$ that represent the source-level dependencies (*e.g.,* procedural invocation, variable access) between modules u and v of the same system. The MDG is automatically constructed using readily available source code analysis tools such as CIA [3] for C, and Acacia [4] for C and C++. An example of an MDG is shown in Figure 7. For now, it suffices to mention that this MDG shows the C files (nodes) and dependencies (directed edges) of the *dot* graph drawing tool.

The clustering problem, as solved by Bunch, can be stated as "finding a good partition of an MDG graph". We use the term *partition* in the traditional mathematical sense, that is, the decomposition of a set of elements (*e.g.,* all nodes of a graph) into mutually disjoint 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. An example of a partitioned MDG for the *dot* tool is shown in Figure 8.

Finding a good graph partition involves systematically navigating through a very large search space of all possible partitions for that graph. Bunch treats graph partitioning (clustering) as an optimization problem. The goal of the optimization is to maximize the value of an objective function, called *Modularization Quality* (*MQ*), which is defined and explained in Table 1.

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 intra-connectivity (*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, and to penalize excessive coupling between clusters. All values of MQ are between -1 (no internal cohesion) and 1 (no external coupling). Figure 1 illustrates the MQ calculation (defined in Table 1) for a small MDG consisting of 8 modules that are partitioned into 3 subsystems.

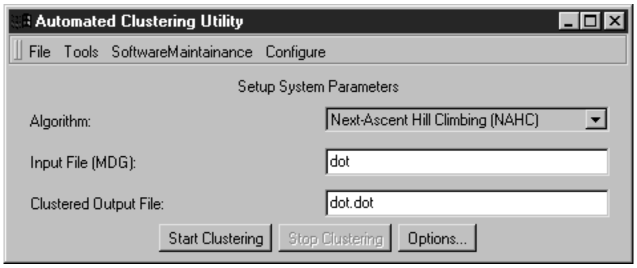
A naive algorithm for finding the "best" (optimal) partition of an MDG is to enumerate all of its partitions and select the partition with the largest MQ value. This algorithm is not practical for MDGs with a large (over 15) number of modules, because the number of partitions of a graph grows exponentially[1] with respect to its number of nodes [14]. Thus, Bunch uses more efficient search algorithms to discover acceptable sub-optimal results. These algorithms are based on hill-climbing and genetic algorithms.

Figure 2 shows the main window of the Bunch tool. This window prompts the user to specify a clustering algorithm (*e.g.,* hill-climbing), the name of the MDG input file (*e.g.,* `dot`), and the name of the clustered MDG output file (*e.g.,* `dot.dot`). The input file contains an encoding of the MDG as a list of ordered pairs. These pairs represent the dependencies between the source-level modules of a software system. The output file, which is generated by Bunch, contains a textual

[1] A graph with 15 nodes has 1,382,958,545 partitions.

Modularization Quality(MQ)	*Intra-Connectivity(A)*	*Inter-Connectivity(E)*
$\mathrm{MQ} = \begin{cases} \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & k > 1 \\ A_1 & k = 1 \end{cases}$	$A_i = \frac{\mu_i}{N_i^2}$	$E_{i,j} = \begin{cases} 0 & i = j \\ \frac{\varepsilon_{i,j}}{2N_iN_j} & i \neq j \end{cases}$

Table 1. The MQ of an MDG that is partitioned into k clusters is the difference between the average inter- and intra-connectivity of the k clusters. The intra-connectivity (A_i) of cluster i consisting of N_i nodes and μ_i intra-edges is the fraction of μ_i over the maximum number of intra-edges of i ($i.e.,$ N_i^2). The inter-connectivity ($E_{i,j}$) between two distinct clusters i and j consisting of N_i and N_j nodes, respectively, and with $\varepsilon_{i,j}$ inter-edges is the fraction of $\varepsilon_{i,j}$ over the maximum number of inter-edges between i and j ($i.e.,$ $2N_iN_j$).



Setup System Parameters

Algorithm: Next-Ascent Hill Climbing (NAHC)

Input File (MDG): dot

Clustered Output File: dot.dot

Figure 2. **The Main Window of Bunch**

description of the partitioned MDG in the *dot* format, which is readable by the *Dotty* [15] visualization tool.

So far, we have applied Bunch to a variety of software systems including: a compiler, a small operating system, a file system of an industrial strength operating system [10], a source code analysis system, and a graph drawing tool (see Section 5). Each result produced by Bunch was evaluated by a developer who had expert knowledge of the implementation of the system being analyzed. In general, we received positive feedback from developers, who noted that Bunch was able to identify many subsystems successfully. In fact, Bunch does a good job of producing a subsystem decomposition in the absence of any knowledge about the software design other than what exists in the source code. However, designers often have some knowledge about the design. As we see in the next section, several improvements to Bunch can be made by taking advantage of such knowledge.

3. LIMITATIONS OF EARLY VERSION OF BUNCH

Based on the feedback we received from developers, we identified three major limitations of Bunch:

1. Almost every system has a few modules that do not seem to belong to any particular subsystem, but rather, to several subsystems. These modules have been called *omnipresent* [12] because they either use or are used by a large number of modules in the system. Omnipresent modules that use other modules can be thought of as *clients* or *drivers*, whereas omnipresent modules that are used by other modules can be thought of as *suppliers* or *libraries*.

   During the clustering process, each module of an MDG is assigned to a subsystem. However, a suitable subsystem for an omnipresent module may not exist because a large number of subsystems may depend on that module. For example, in a C program, it may not make sense to assign stdio.h to any particular subsystem, if a large number of subsystems perform I/O operations.

   Users suggested that we provide facilities to identify and, subsequently, isolate omnipresent modules since these modules tend to obfuscate the system structure. A solution would be to isolate all driver modules in one subsystem and all library modules in another subsystem.

2. Experienced developers tend to have good intuition about which modules belong to which subsystems. Unfortunately, Bunch might produce results that conflict with this intuition. Although there is always a possibility that developers are mistaken, we believe that Bunch is more likely to be responsible for such conflicts for two reasons. 1) Bunch produces sub-optimal results. 2) MQ only takes into account the topology of the MDG graph and, hence, cannot hope to capture all of the subtleties of semantics-based clustering.

   Ideally, developers should be able to use their knowledge to bias the partitioning process. For example, developers should be able to specify constraints such as: “always group these two modules in the same subsystem”, “these five modules constitute an entire subsystem”, and so on. Such constraints can reduce the large search space of graph partitions and, thus, help Bunch produce good results more quickly.

3. During maintenance, the structure of a software system inevitably changes. For example, a new module is introduced, an old module is replaced, inter-module dependencies are added or deleted, and so on.

   A developer that used Bunch to re-cluster a modified MDG observed that a minor structural change to the software structure resulted in significant changes to the MDG partition. This is not very surprising, as Bunch results are sensitive to factors such as how long users are willing to wait, and which areas of the search space the algorithm happened to explore.

   Ideally, for maintenance purposes, Bunch should try to preserve the existing subsystem structure when minor changes to the system are made. A radical re-partitioning from scratch, rather than an incremental one, can be justified only after significant changes to the system structure are made.

The next section describes the enhancements we made to Bunch in order to solve the aforementioned shortcomings.

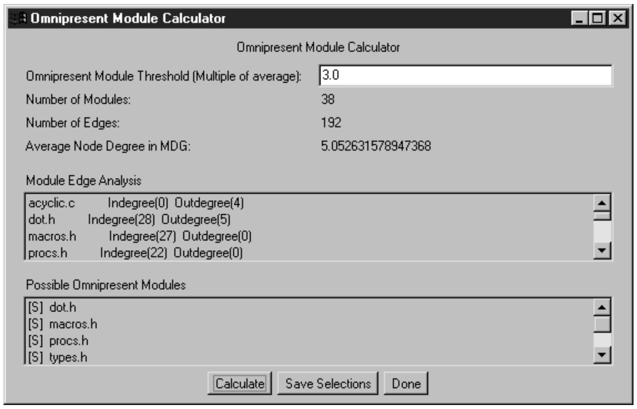
4. BUNCH EXTENSIONS

We now describe the three new features of Bunch.

4.1. Omnipresent Module Detection & Assignment

The new version of Bunch allows users to specify two lists of omnipresent modules, one for *clients* and another for *suppliers*. These lists can be specified manually or determined automatically using Bunch’s *omnipresent module calculator*. Regardless of how the omnipresent modules are determined, Bunch assigns the omnipresent clients and suppliers to two separate subsystems.

Figure 3 shows the user interface of the omnipresent module calculator. Users start by specifying an omnipresent module threshold as a multiple of the average edge in- and out-degree of the MDG nodes. For example, if the user-specified multiple is 3 (see Figure 3), that means that a module is omnipresent if it has at least three times more incident edges than a typical module. When the *Calculate* button is pressed, the calculator computes the average node degree for the graph and then searches the MDG for modules that have a degree that exceeds the threshold. Finally, the candidate omnipresent modules are displayed in a list (see the bottom of Figure 3), from which users can select their omnipresent modules. The selected modules are then automatically placed in the two lists of the omnipresent module window, shown in Figure 4. These lists may be edited by users who want to add or remove omnipresent modules. This window also enables users to specify how the incident edges of the omnipresent modules will be drawn by the graph drawing tool when creating the output. These edges may be drawn using a different color or may be removed from the graph presentation altogether.



Omnipresent Module Threshold (Multiple of average): 3.0
Number of Modules: 38
Number of Edges: 192
Average Node Degree in MDG: 5.052631578947368

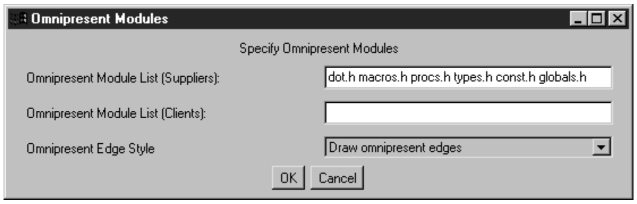
Module Edge Analysis

```
acyclic.c      Indegree(0)  Outdegree(4)
dot.h       Indegree(28)  Outdegree(5)
macros.h       Indegree(27)  Outdegree(0)
procs.h       Indegree(22)  Outdegree(0)
```

Possible Omnipresent Modules

```
[S]  dot.h
[S]  macros.h
[S]  procs.h
[S]  types.h
```

Figure 3. Omnipresent Module Calculator



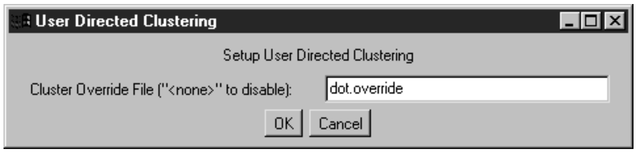
Omnipresent Module List (Suppliers): dot.h macros.h procs.h types.h const.h globals.h
Omnipresent Module List (Clients):
Omnipresent Edge Style: Draw omnipresent edges

Figure 4. Specifying Omnipresent Modules

4.2. User-Directed Clustering

A user who is trying to extract the structure of a software system often has some knowledge about the actual system design. The *user-directed clustering* feature of Bunch enables users to cluster some modules manually, using their knowledge of the system design while taking advantage of the automatic clustering capabilities of Bunch to organize the remaining modules.

Figure 5 shows the user-directed clustering dialog box. The user specifies a file name that describes the user-specified clusters in the software system. The format of the files is simply the cluster name followed by



Cluster Override File ("<none>" to disable): dot.override

Figure 5. User-directed Clustering Window

the names on the modules in the cluster. Bunch preserves all user-specified clusters while searching for the optimal partitioning of an MDG. By default, Bunch never removes modules from, or add modules to, the user-specified clusters. However, "locking" the user-specified clusters is not always desirable. Therefore, Bunch includes an option that allows the automatic clustering process to add modules to user-specified clusters, if doing so yields a better result.

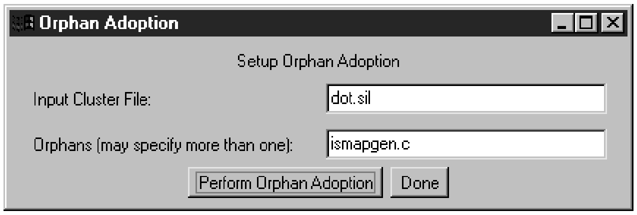
Both user-directed clustering and the manual placement of omnipresent modules into subsystems have the advantageous side-effect of reducing the search space of MDG partitions. By enabling the manual placement of modules into subsystems, these techniques decrease the number of nodes in the MDG for the purposes of the optimization and, as a result, speed up the clustering process.

4.3. Incremental Software Structure Maintenance

Once a system organization is obtained, it is desirable to preserve as much of it as possible during the evolution of the system. The integration of the *orphan adoption* technique [17] into Bunch enables designers to preserve the subsystem structure when orphan modules are introduced. An orphan module is either a new module that is being integrated into the system, or a module that has undergone structural changes (*e.g.,* new dependencies are created between the modified module and other modules of the system).

Figure 6 shows the orphan adoption dialog box. The user is prompted for the name of a file (*e.g.,* `dot.sil`) that specifies a known structure (MDG partitioning) of a software system. The known partition may be constructed manually by the user or obtained automatically using a previous result produced by Bunch. The user is also prompted for a list of orphan modules (*e.g.,* `ismapgen.c`). The new relationships involving the orphan modules are obtained by the file specified in the *Input File (MDG)* field of the main Bunch window (Figure 2).

Bunch implements orphan adoption by first measuring the MQ when the orphan module is placed alone into a new subsystem. Bunch then moves the orphan module into existing subsystems, one at a time, and records the MQ for each of the relocations. The subsystem that produces the highest MQ is selected as the parent for the module. This process, which is linear with respect to the number of clusters in the partition, is repeated for each orphan module.



Input Cluster File: dot.sil

Orphans (may specify more than one): ismapgen.c

Figure 6. Orphan Adoption Window

After these new features were added to Bunch, we began a new round of evaluation.

5. BUNCH REEVALUATION

This section details a case study of using Bunch to analyze the structure of a specific program that one of the authors helped design and implement. In particular, we applied Bunch to the graph drawing tool *dot*, for which we have access to a sequence of many versions evolving over six years. (Note that *dot* was used to draw Figures 7 through 11.) For the purposes of this paper, we limit ourselves to two recent, consecutive versions of *dot*. We first describe how a user-guided session with Bunch was used to recover the program's structure, and then show how Bunch handled the incremental transition from one version to the next.

In what follows, it is useful to have a high-level view of the *dot* program and its structure. The program reads in a description of a directed graph and, based on user options and the structure and attributes of the graph, it draws the graph using an automatic layout algorithm. The automatic graph layout technique is basically a pipeline of graph transformations, through which the graph is filtered, where 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[6] curves. Once the layout is done, *dot* writes it 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.

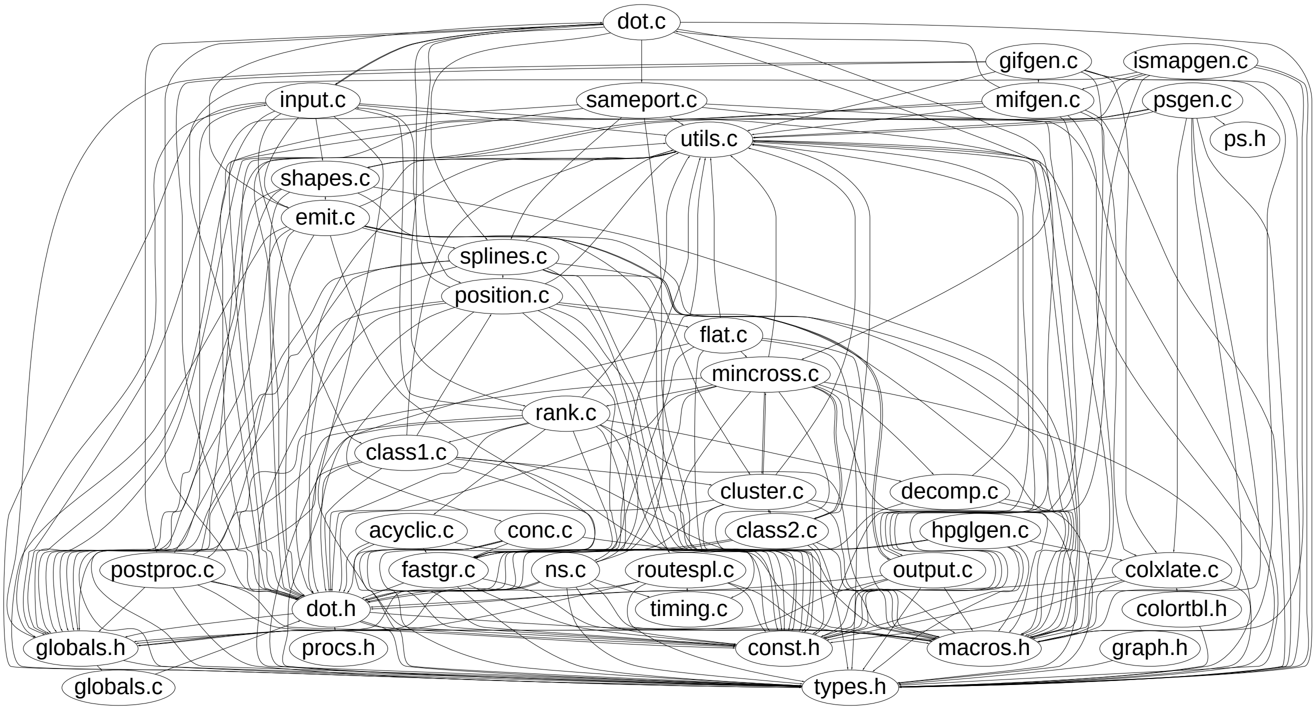


Figure 7. The Module Dependency Graph (MDG) of *dot*.

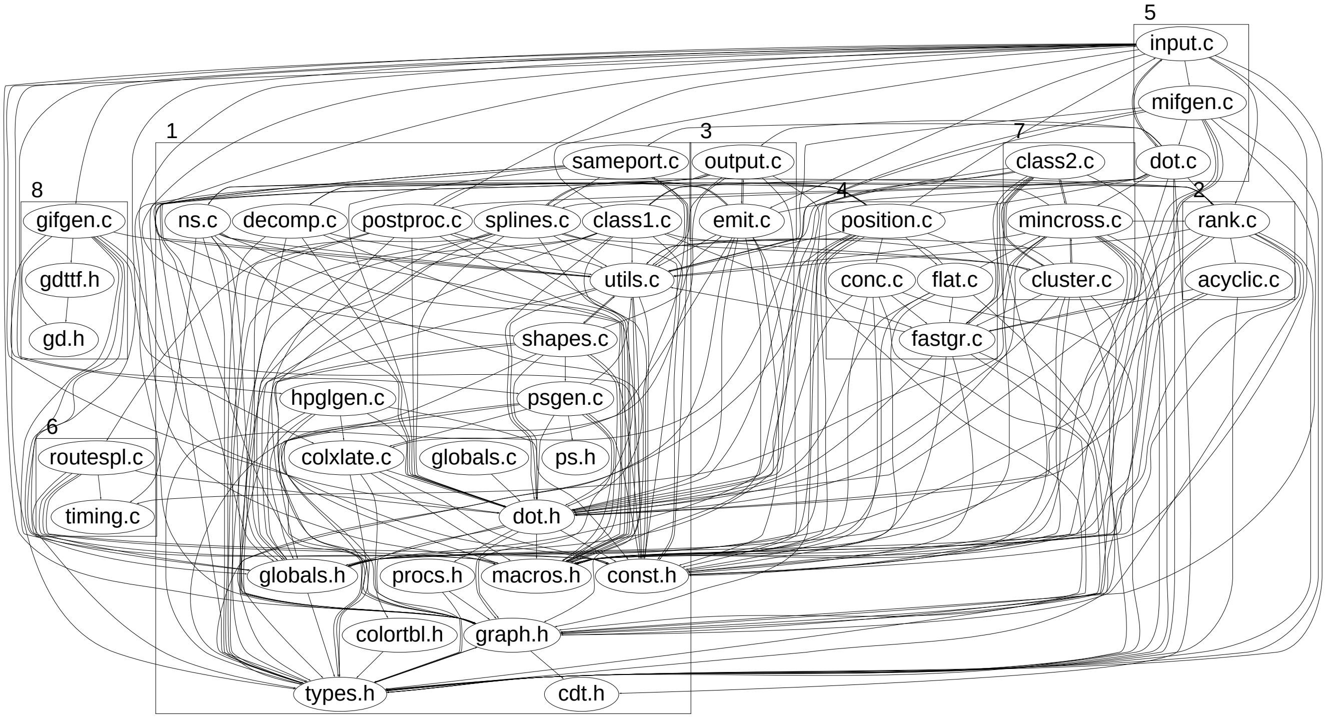


Figure 8. The automatically produced MDG partition of *dot*.

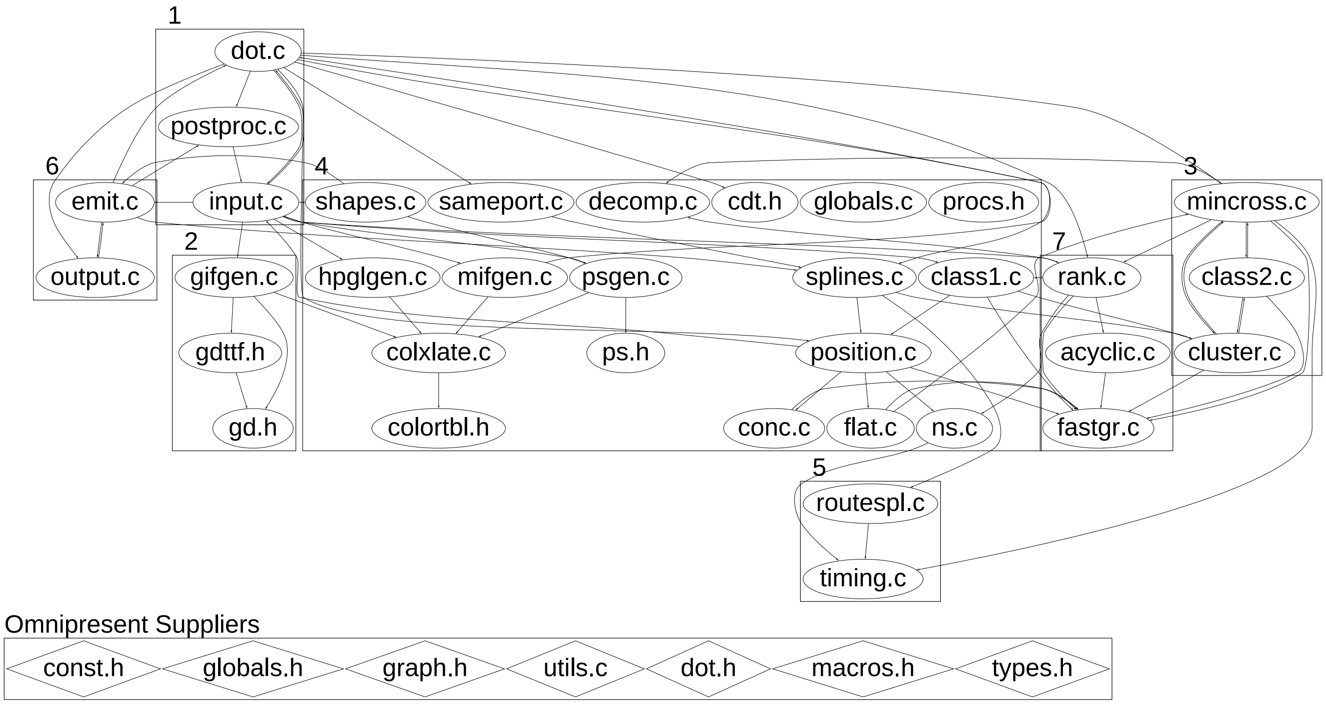


Figure 9. The *dot* partition after omnipresent modules have been identified and isolated.

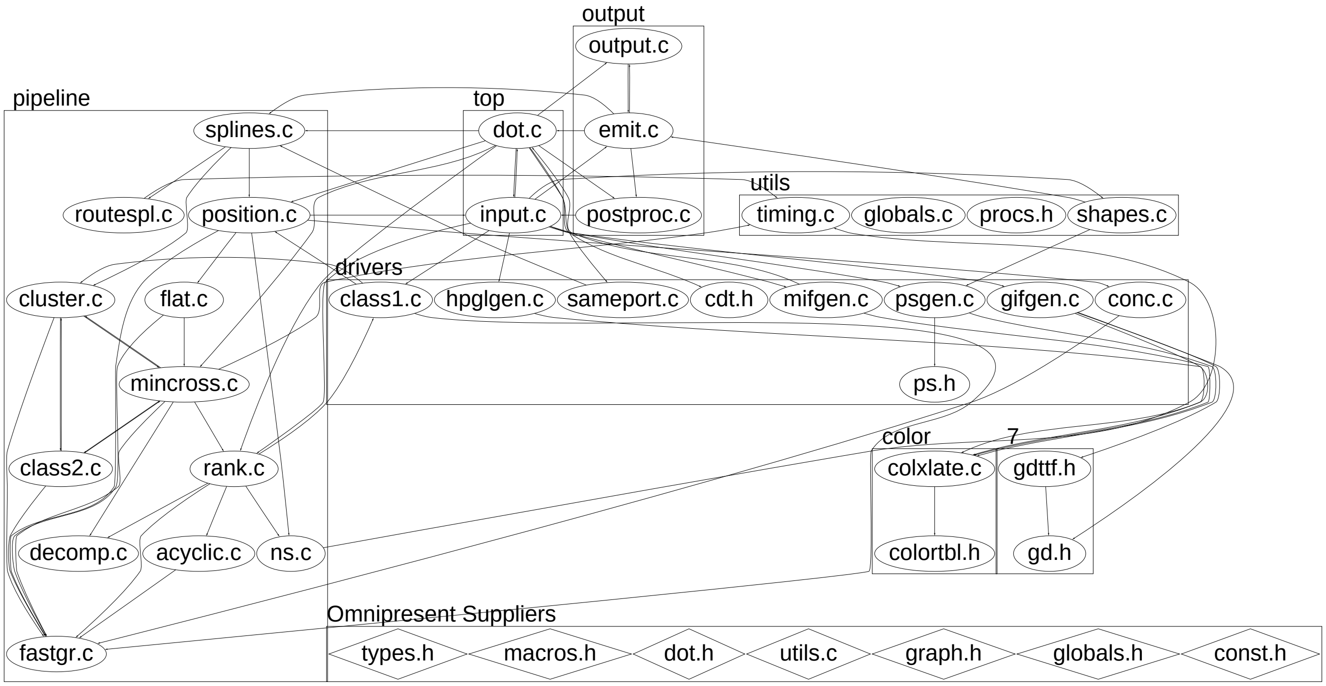


Figure 10. The *dot* partition after the user-defined clusters have been specified.

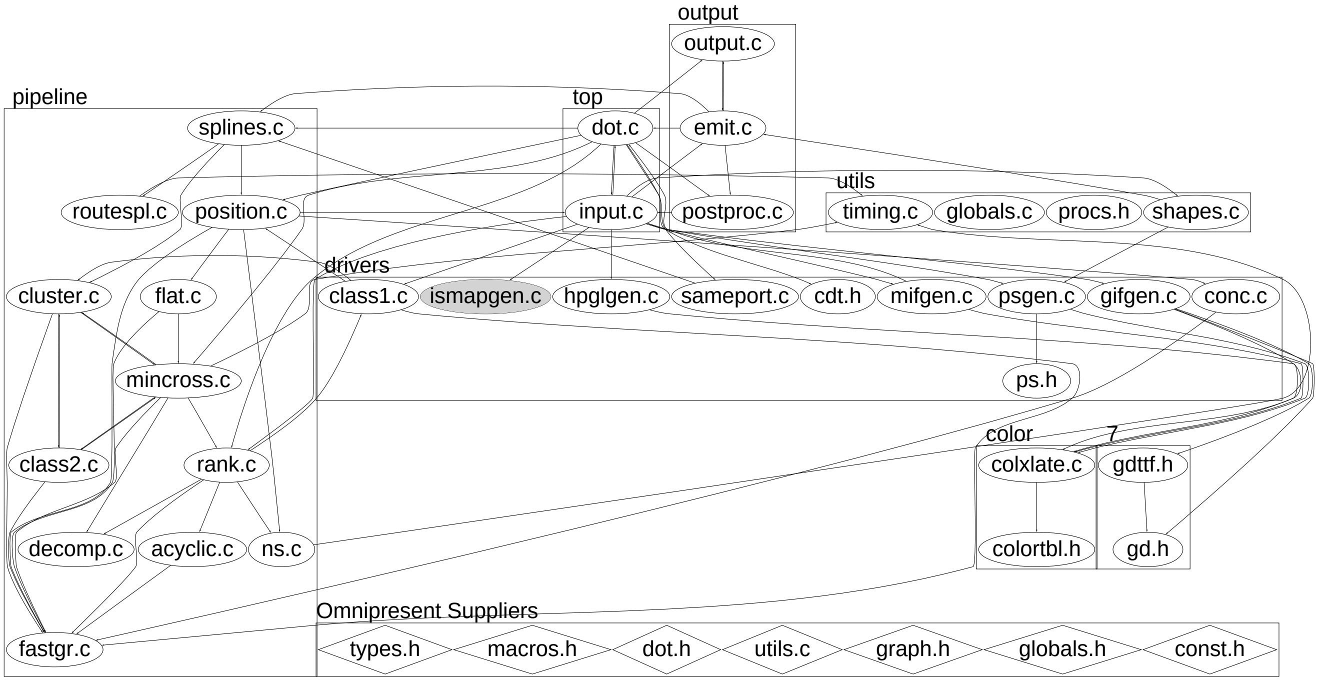


Figure 11. **The *dot* partition after the adoption of the orphan module** ismapgen.c**.**

To generate the MDG, we first used the Acacia [4] system to create a source code database for *dot*. Then, using Acacia and standard Unix tools, we generated[2] the MDG for input to Bunch. Figure 7 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 Figure 8.[3] The partitioning shown is reasonably close to the software structure described above. Cluster 1 contains most of the layout algorithm. Cluster 2 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 edge-crossing minimization. Finally, cluster 8 concerns GIF-related output.

However, there are some anomalies in the partition shown in Figure 8. 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, in fact, exposes 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 uses 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. By checking the omnipresent module calculator, we see that these modules are identified as potential omnipresent modules. Based on program semantics, we recognize these modules as *include* files that define data types that are common to many of the other program 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. The resulting clustered graph is shown in Figure 9. We notice immediately that the graph is much cleaner, and we recognize a basic high-level structure that closely matches the described program structure.

The next step is to start from this automatically generated structure and use the Bunch user-directed clustering mechanism to add user information, in order to fine-tune the cluster structure by "locking in" certain cluster associations. In this case, we keep clusters 1 and 6 largely unchanged, naming them **top** and **output**. We recognize that cluster 4 naturally separates into three sub-clusters: one representing the main functional pipeline of the layout algorithm, combined

[2] Using an SGI 200 MHz Indy, generating the database took 117 secs.; extracting the MDG used another 5 secs. For comparison, it took 80 secs. to compile and link *dot*.

[3] Using a Pentium II 266 MHz PC, generating the MDG partition took approximately 15 minutes.

with parts of cluster 3, 5 and 7; one concerning color information; and one representing the various output device-specific drivers, the last also incorporating the driver `gifgen.c`, which was previously in cluster 2. In addition, we clean up cluster 1 by moving the module `postproc.c` into the **output** cluster, and identify 3 modules as providing general-purpose utility routines. The user-specified clusters are as follows:

top	`dot.c input.c`
output	`emit.c postproc.c output.c`
pipeline	`decomp.c position.c rank.c ns.c splines.c fastgr.c cluster.c mincross.c flat.c acyclic.c routespl.c`
drivers	`ps.h psgen.c hpglgen.c gifgen.c mifgen.c`
color	`colxlate.c colortbl.h`
utils	`utils.c shapes.c timing.c`

The remaining modules may be assigned to any cluster. Bunch will try to assign them to clusters that maximize the value of MQ.

Based on these constraints, Bunch performs another clustering and, in under one minute, presents us with the structure shown in Figure 10. The modules that were not constrained by the user-directed clustering mechanism were distributed sensibly among the clusters. Specifically, Bunch suggested the need for an additional cluster (*i.e.,* 7) to contain the two auxiliary, GIF-related modules. By combining the automatic use of the MQ measurement and manual user guidance, we have arrived at a module architecture for the *dot* program that closely reflects the designer's view of the underlying structure.

Up until this point, we have focused on the analysis of a single version of *dot*. But, like most programs, *dot* exists in many versions, representing changes over time as bugs are fixed and new features are added. Barring major structural changes between versions, Bunch users should be able to create a new MDG partition by making incremental changes to the previous one. Conversely, if the resulting partition does not fit the program's organization, this could be considered a flag that some significant restructuring has occurred.

In the next version of *dot*, a new output format is supported and a new driver module `ismapgen.c` is introduced. We construct a new MDG that encodes the added module and its dependencies but, rather than starting our cluster analysis from scratch, we instruct Bunch to reuse our previous analysis and treat the new module as an orphan to be adopted by one of the subsystems. It then uses the MQ measurement to determine the best cluster to adopt the orphan, including the possibility of a new singleton cluster containing just the orphan. Within a two seconds, Bunch produces the new partition shown in Figure 11. As we can see, it is minimally different from the structure we saw in Figure 10. More importantly, we note that the new module has been placed, appropriately, with the other output drivers.

6. RELATED WORK

Early work by Belady and Evangelisti [2] identified automatic clustering as a means to produce high-level views of the structure of software systems. Much of the earlier work on software clustering, like that of Hutchens and Basili [9], focuses 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 [16] introduced concepts such as low-coupling and high-cohesion into the clustering problem. The Rigi system [12], 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 [1], who used common patterns in file names as a clustering criterion. Our first version of Bunch [10] treated clustering as an optimization problem and was the first system to employ genetic algorithms to overcome the problem of getting "stuck" on local optima solutions. In this paper, we described how Bunch was improved by integrating known clustering techniques such as omnipresent module detection and orphan adoption, along with our new technique for integrating user-specified clusters into Bunch's automatic clustering process.

In addition to bottom-up clustering approaches like ours, which produce high-level structural views starting from the source code, some promising top-down approaches have demonstrated their effectiveness in solving software maintenance problems. We already mentioned the work of Tzerpos and Holt on orphan adoption [17]. Another top-down approach by Murphy *et al.* uses Software Reflexion Models [13] to capture and exploit differences that exist between the actual source code organization and the designer's mental model of the high-level system organization.

For a thorough overview of software clustering techniques, we suggest a recent survey by Wiggerts [18].

7. CONCLUSIONS & FUTURE WORK

This paper shows how an automatic clustering tool can generate better results faster when users are able to integrate their knowledge – if and when it is available – into the clustering process. Specifically, the manual assignment of some modules to subsystems helps reduce the number of modules that need to be assigned to subsystems automatically, thus dramatically reducing the search space of MDG partitions. This paper also shows how the subsystem structure of a system can be maintained incrementally after the original structure has been produced.

In the future, we plan to conduct and document more case studies to demonstrate the effectiveness and any limitations we find in the version of Bunch. So far, our case studies have involved C, C++, and Turing programs. We plan to study Java programs after the source code analysis system for Java, which is currently being developed at the AT&T Labs, is complete.

One of our goals for the next version of Bunch is to further improve its performance. Specifically, we are adapting our existing optimization algorithms so that they can be executed in parallel over a network of computers.

Another goal is to experiment with MDGs that have weighted edges. This would require changing the objective function MQ, which currently treats all module dependencies equally. For example, pairs of modules that call each other once are considered equivalent to pairs of modules that call each other many times.

Finally, we hope to receive more feedback by making Bunch freely available from http://www.mcs.drexel.edu/˜ serg. Copies of Acacia, *dot*, and Dotty are available from http://www.research.att.com/sw/tools.

8. ACKNOWLEDGMENTS

This research is sponsored by a CAREER Award from the National Science Foundation (NSF), under grant CCR-9733569. Additional support was provided by a grant from the research laboratories of AT&T.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, the U.S. government, or AT&T.

References

[1] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proc. 20th Intl. Conf. Software Engineering*, May 1998.

[2] L. A. Belady and C. J. Evangelisti. System Partitioning and its Measure. *Journal of Systems and Software*, 2:23–29, 1981.

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

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

[5] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm (submitted for publication). 1998.

[6] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics.* Addison-Wesley, 2nd edition, 1990.

[7] E. Gansner, E. Koutsofios, S. North, and K. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, Mar. 1993.

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

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

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

[11] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[12] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[13] 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.

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

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

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

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

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