

Demonstration Proposal:
Clustering Module Dependency Graphs of Software Systems
Using the Bunch Tool

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

Abstract

In this demonstration we will show how our tool (Bunch), along with other tools for source code analysis and graph visualization, can be used to recover the high-level structure of a software system directly from its source code. We accomplish this task by first using a source code analysis system (e.g., CIA, Acacia) to produce a module dependency graph that represents the system modules and module-level inter-relationships. We then use this graph as input to Bunch, which partitions the graph. The resultant clustered graph is displayed using a graph visualization tool (e.g., dotty, Tom Sawyer).

1. Introduction

The manual decomposition of a software system into meaningful subsystems is a very time consuming process. This is primarily due to the complexity associated with understanding the large number of inter-relationships that exist between the source-level components. We rely on a set of tools to simplify, and to a large extent automate, this process. The first step involves the use of source code analysis tools to determine the *Module Dependency Graph (MDG)* of a software system. The second step involves the use of Bunch to partition the *MDG* into a set of non-overlapping clusters (e.g., subsystems). The third step involves the use of a graph visualization tool to show the partitioned *MDG*.

Our experience with using Bunch has shown that it is a good tool for recovering the high-level organization of a software system directly from source code. The original version of Bunch completely automated the software clustering process. Based on user feedback, and our own experience with using the tool, we subsequently extended Bunch with additional features that take advantage of user knowledge (if any exists) about the actual system structure. These new features complement Bunch's automatic clustering engine by allowing

user-supplied information to guide the otherwise automatic clustering process.

2. The Bunch Clustering Tool

Bunch is a clustering tool that partitions source-code modules into subsystems. This task requires a well-defined representation of the systems modules and dependencies, along with a set of algorithms that transform the original system representation into a hierarchy of subsystems.

Bunch accepts as input a description of a software system's structure in the form of a *Module Dependency Graph (MDG)*. The vertices of the *MDG* represent the software components (e.g., C files, C++ classes) and the edges represent inter-module dependencies (e.g., procedure calls, inherits). The *MDG* is a convenient input to Bunch because it can be generated automatically using readily available source code analysis tools such as CIA (for C) [1] and Acacia (for C/C++) [2]. Once the *MDG* is created, the user can select one of Bunch's clustering algorithms to partition the *MDG* into a hierarchy of subsystems.

Bunch currently supports four algorithms that are based on treating the clustering process as an optimization problem. The "optimal" algorithm

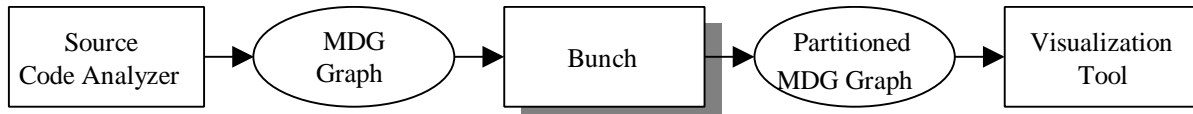


Figure 1. Software Architecture of Clustering System

produces the best results, but is only practical for small *MDGs* (*i.e.*, 15 vertices). Two additional hill climbing algorithms and a genetic algorithm produce sub-optimal, but acceptable, results on large systems. All of our algorithms are based on maximizing an objective function which is based on measuring the “quality” of a particular partition of the *MDG*. Our objective function, called *Modularization Quality (MQ)*, rewards the grouping of highly related modules into common subsystems, but penalizes partitions that have many inter-cluster dependencies. The algorithms supported by Bunch, and the *MQ* objective function, have been described extensively in other papers [3,4,5]. In Section 3.1 we present a brief description of our algorithms.

Once the clustering process is complete, the internal representation of the partitioned *MDG* can be visualized. The current version of Bunch can generate output for both the Tom Sawyer[7] and *dotty*[1] graph visualization tools. Figure 1 depicts the high-level architecture of our clustering system.

3. Using Bunch

The user interface of Bunch is based on a tabbed metaphor. Our goal was to design a well-organized interface that allows users to navigate through all of the common clustering options that are supported by Bunch. Rarely used options are relegated to other tabs or configuration dialog boxes that are accessible from Bunch’s main window. Figure 2 illustrates the main window of Bunch.

3.1 The Clustering Process

The first step in the automatic clustering process is to capture the source-level relationships in a *MDG* file. This step can either be performed manually, or by using a source code parsing utility. We use the CIA and Acacia tools for this step although the user is free to substitute any tool that

can scan source code and produce a module dependency graph file. The syntax of the *MDG* file is very simple. Each line in the file takes the form of: `<source_module> <target_module>`. An entry is placed in the *MDG* file if there is at least one dependency relationship (*e.g.*, function call, variable reference, macro reference, type reference) between the source and target modules. The *MDG* file name is specified in the Bunch tool by pressing the *Select...* button to the right of the *Input Graph File* field. Bunch will automatically define the output filename as the same as the name of the dependency file selected, but will add a file extension that is based on the output file format. This filename extension will avoid overwriting the original *MDG* file. If desired, the user can change the output filename, or the path of the output file, by directly editing the *Output Cluster File* text field.

Once the *MDG* file is specified, the user is given the opportunity to select a clustering algorithm using the *Clustering Method* list. The current version of Bunch supports the following clustering techniques:

1. **SAHC:** The *Steepest Ascent Hill Climbing* (SAHC) algorithm is based on traditional hill climbing techniques. The goal of this algorithm is to progressively create a new partition from the current partition of the *MDG* where the *MQ* of the newer partition is larger than the *MQ* of the original partition. Each iteration of the algorithm attempts to improve *MQ* by finding a *maximal neighboring partition (MNP)* of the current partition. We define *NP* to be a neighbor of partition *P* if and only if *NP* is exactly the same as *P* except that a single element in *P* is in a different cluster in *NP*. The *MNP* is determined by examining all *NP*’s of *P* and selecting the one that has the largest *MQ*. The SAHC algorithm converges when it is unable to find a *MNP* of the current partition with a larger *MQ*.

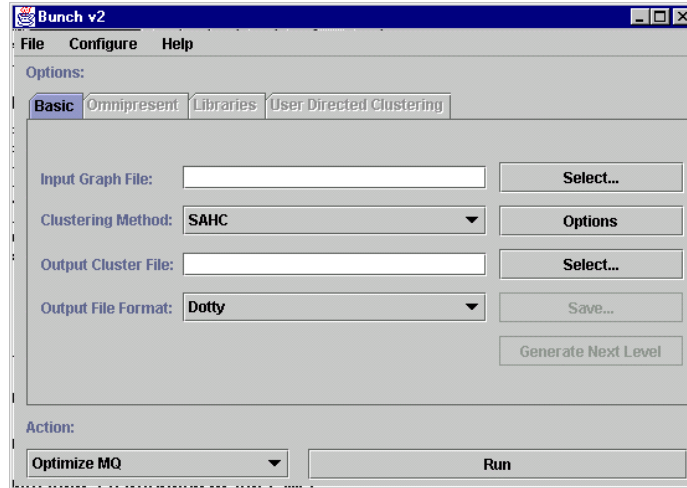


Figure 2. Bunch Main Window

2. **NAHC:** The *Next Ascent Hill Climbing* (NAHC) algorithm is another hill climbing algorithm that is similar, but often faster, than its SAHC counterpart. The NAHC algorithm differs from SAHC in how it systematically improves partitions of the *MDG*. In NAHC, our hill climbing improvement technique is based on finding a *better neighboring partition (BNP)* of the current partition. A *BNP* of the current partition *P* is found by randomly enumerating through the *NP*'s of *P* until a *NP* is found with a larger *MQ*. The NAHC algorithm converges when no *BNP* of *P* can be found with a larger *MQ*.
3. **GA:** A Genetic Algorithm that uses classical GA operators such as selection, crossover, and mutation techniques to find a good partition of the *MDG*.
4. **Optimal:** This algorithm is only usable for small graphs because it enumerates every possible partition of the *MDG* and chooses the best one.

Now that the *MDG* file and the clustered output file name have been selected, the user must select an output file format. The *Output File Format* list contains all the available output formats that are supported by Bunch. The available formats are:

- **dotty:** An output format viewable with the AT&T Labs-Research's dotty tool.

- **Tom Sawyer:** An output format viewable with the Tom Sawyer visualization tool.
- **Text:** A simple format that defines one cluster in each line of the output file and simply enumerates the names of the modules that are contained within that cluster. This file format can be used as input for the *User Directed Clustering* option. This option is briefly described in Section 4. A detailed description of this feature can be found in another paper[4].

After selecting all of the desired options, the clustering process is initiated by simply pressing the *Run* Button. As Bunch processes the *MDG*, the dialog in Figure 3 will appear. This dialog provides the user with feedback on the progress of the clustering process. Once the progress dialog disappears, the output file is created. This output file can then be visualized using either dotty or Tom Sawyer.

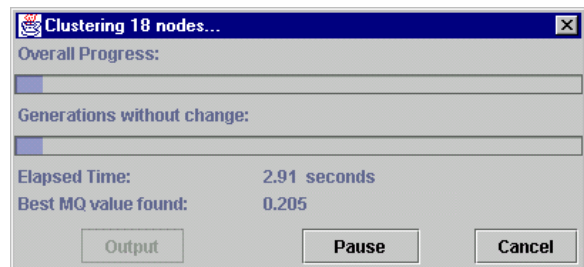


Figure 3. The Clustering Progress Dialog

4. Other Bunch Features

Bunch contains several other features that allow the user to direct Bunch's clustering engine by leveraging known information about the actual system structure. Specifically these features include:

1. *User-directed clustering* where the user provides a text file that contains known information about the actual system structure. For example, the user may want to specify that modules *A* and *B* both belong to the same subsystem (cluster). Bunch honors such user input during the automatic clustering process.
2. Specification of *library* and *omnipresent* modules. Because these modules are highly connected to other modules in the system, the automatic clustering engine will make compensations for these modules, which often negatively effects the quality of the results. By manually specifying these modules, Bunch will isolate these modules into their own subsystems or remove them from the clustering process altogether.
3. Support for the incremental maintenance of a system's structure using the *Orphan Adoption* technique[8]. This technique recommends which cluster is most suitable to "adopt" a new system module.
4. Bunch features a modularization quality calculator for evaluating the *MQ* value of a given partition of the *MDG*, and an omnipresent module calculator which identifies candidate omnipresent modules based on user supplied thresholds.

These features, which are all available through Bunch's user interface, are described in detail in a recent paper[4] and in our online documentation.

5. Obtaining Bunch

Bunch is implemented in Java and requires a JDK 1.1.7 compliant Java Virtual Machine (JVM). We have made Bunch freely available and encourage

people to try our tool and provide feedback to us for additional improvements. Bunch, and its online documentation, can be obtained from the Drexel University Software Engineering Research Group (SERG) web page which is located on the Internet (<http://www.mcs.drexel.edu/~serg>).

Users may also download the latest copies of the CIA, Acacia, and dotty from the AT&T Labs-Research web page which is located on the Internet (<http://www.research.att.com/sw/tools>).

6. References

- [1] Y. Chen. Reverse Engineering. In B. Krishnamurthy, editor, *Practical Reusable Unix Software*, chapter 6, pages 177-208. John Wiley & Sons, New York, 1995.
- [2] Y. Chen, E. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proc. 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1997.
- [3] S. Mancoridis, B.S. 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.
- [4] S. Mancoridis, B.S. Mitchell, Y. Chen, and E. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures (submitted for publication). 1998.
- [5] D. Doval, S. Mancoridis, B.S. Mitchell. Automatic Clustering of Software Systems using a Genetic Algorithm (submitted for publication). 1998.
- [6] 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.
- [7] Tom Sawyer. Graph Drawing and Visualization Tool. <http://www.tomsawyer.com>
- [8] V. Tzerpos, R.C.Holt. The Orphan Adoption Problem in Architecture Maintenance. In *Proc. Working Conference on Reverse Engineering*, 1997.