Bunch: A Clustering Tool for the Incremental Maintenance of the Structure of Software Systems

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

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

Abstract

The combination of source code analysis and automatic clustering techniques have been shown to produce good results for recovering the high-level organization of a software system directly from its source code. These techniques, however, have several limitations as they do not provide mechanisms for integrating the knowledge of a designer into the automatic clustering process. Instead, fully automatic clustering techniques must solely rely on source-code level inter-relationships to derive higher-level structural views of the systems organization.

This paper describes an automatic clustering system that we developed that is capable of integrating user-specified knowledge about the actual system structure into the otherwise fully automatic clustering process. These capabilities leverage the advantages of fully automatic clustering techniques while providing facilities to integrate knowledge that the designer may possess about the actual system structure.

Keywords: Clustering, Software Maintenance, Incremental Software Structure Maintenance, Program Understanding, Reverse Engineering, Software Structure, Optimization, Genetic Algorithms.

1 Introduction

Manually discovering the subsystem structure of a software system directly from the source code is often not possible due to the complexity associated with understanding the large number of inter-relationships that exist between the native source code components. Therefore, automatic software modularization techniques are helpful as they provide a valuable service to software professionals who are trying to understand the intricate organization of complex systems without having to go through an exhaustive (and time consuming) analysis of the source code. This task is accomplished by automatically clustering the software components in a way that derives the high-level subsystem structure of a software system directly from the component-level relationships that are present in the source code.

Do we want to keep or scrap the above paragraph?
Remainder of section to be written by Spiros.

2 Automatic Software Modularization

In a previous paper[3] we described an automatic software modularization technique based on formally quantifying the quality of a partitioning of a module dependency graph (MDG). The $MDG = (M, R)$ is a graph where M is the set of named modules in the software system, and $R \subseteq M \times M$ is a set of ordered pairs of the form $\langle u, v \rangle$ which represents the source-level relationships that exist between module u and module v. The MDG is a convenient input to our technique because it can be automatically constructed by using readily available source code analysis tools such as CIA[1] for C and Acacia[2] for C++.

Once the MDG of a software system is determined, we search for partitions so that the inter-connectivity (*i.e.,* connections between the components of two distinct clusters) is minimized and the intra-connectivity (*i.e.,* connections between the components of the same cluster) is maximized. This task is accomplished by treating the clustering process as an optimization problem where the goal is to maximize the value of an objective function that is based on a formal characterization of the trade-off between inter- and intra-connectivity. We refer to our objective function as the Modularization Quality (MQ) of a partition for a particular MDG. MQ is formally described in the next section of this paper.

One way to find the optimal partition would be to enumerate through all of the partitions of the MDG and select the one with the largest MQ value. However, this is often not possible because the number of partitions of the MDG grows exponentially with respect to the number of modules in the software system. Because discovering the optimal partition of the MDG is often only feasible for small software systems (*e.g.,* fewer then 15 modules), we direct our attention instead, to using search algorithms that are capable of efficiently discovering acceptable sub-optimal results. The sub-optimal search strategies that we have investigated and implemented are based on hill-climbing and genetic algorithm techniques.

2.1 Quantification of Automatic Clustering

Below we illustrate our modularization quality (MQ) measurement. This measurement, which is used as the objective function of our clustering algorithm, represents the "quality" of a system modularization. The MQ measurement is designed to reward the creation of highly cohesive clusters (high intra-connectivity) while penalizing excessive module coupling (high inter-connectivity). This tradeoff is established by subtracting the average intra-connectivity (A) from the average inter-connectivity (E).

$$A_i = \frac{\mu_i}{N_i^2} \qquad E_{i,j} = \begin{cases} 0 & i = j \\ \frac{\varepsilon_{i,j}}{2N_iN_j} & i \neq j \end{cases} \qquad MQ = \begin{cases} \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & k > 1 \\ A_1 & k = 1 \end{cases}$$

Intra-Connectivity(A) *Inter-Connectivity(E)* *Modularization Quality(MQ)*

The MQ measurement for an MDG partitioned into k clusters is calculated by subtracting the average inter-connectivity from the average intra-connectivity. The *intra-connectivity* measurement A_i of cluster i consisting of N_i components and μ_i intra-dependencies is defined as the fraction of the maximum possible number of intra-dependencies of cluster i, which is N_i^2. The *inter-connectivity* measurement $E_{i,j}$ between two distinct clusters i and j which consists of N_i and N_j components, respectively, with $\varepsilon_{i,j}$ inter-dependencies is defined as the fraction of the maximum possible number of inter-dependencies between clusters i and j, which is $2N_iN_j$.

Figure 1 illustrates an example MQ calculation for an MDG consisting of 8 components that are partitioned into 3 subsystems. The value of MQ is approximately 0.186, which is the difference between the average intra-connectivity (A) of 0.269, and an average inter-connectivity (E) of 0.083. The MQ measurement is bounded between -1 (no cohesion within the sub-

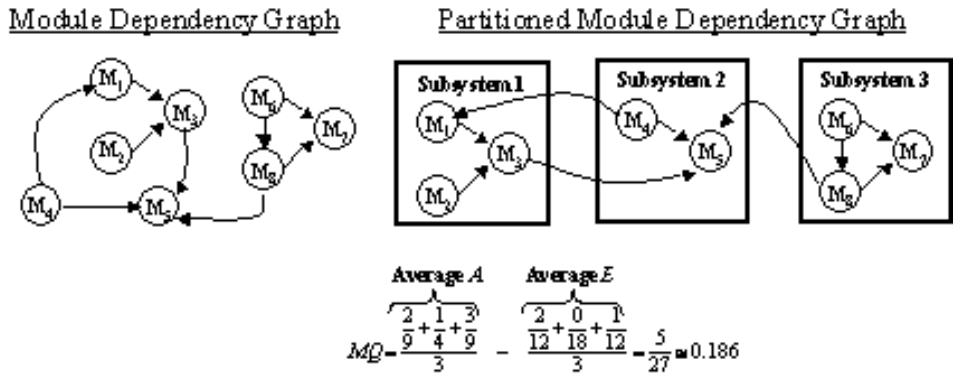


Figure 1: **Modularization Quality Example**

systems) and 1 (no coupling between the subsystems). The larger the value of MQ, the better the partition. Thus, the MQ measurement adheres to our fundamental assumption that well-designed software systems are organized into cohesive clusters that are loosely interconnected.

2.2 Limitations of Automatic Clustering Techniques

Fully automatic clustering techniques provide a valuable service to software practitioners who are trying to understand the organization of a software system in the absence of any knowledge about the actual system structure. However, because our automatic clustering strategy is based solely on the topology of the MDG several shortcomings associated with fully automatic clustering may arise that will negatively impact the results that are produced by our system. One can make better progress if they are able to integrate knowledge about the actual system design into the otherwise fully automatic clustering process. Specifically:

- Omnipresent entities are modules that either use a large number of the other modules in the system (*i.e.,* drivers), or modules that are used by a large number of other modules in the system (*i.e.,* libraries). Because these modules are highly interconnected to the other modules in the system, the automatic clustering algorithm will make compensations for these modules, which often negatively effects the quality of the result. We agree with Müller[4] assumption that omnipresent modules obscure system structure and should be removed from the clustering

analysis process.

- The original version of our automatic clustering tool did not enable a user to specify that certain modules must always be contained in the same subsystem. This feature is desirable because our automatic clustering technique makes its partitioning decisions based on maximizing the MQ measurement. Because this measurement is based on the overall average inter- and intra-connectivity, tradeoffs that are inconsistent with the users understanding of the actual system organization might be made in order to maximize the MQ.

- Each change to the MDG (*i.e.,* introduction of a new module or an update to an existing module that introduces additional inter-module dependencies) may result in a radical change in the automatically produced clustering. It would be desirable to offer a facility for preserving existing architectural and structural information by localizing the impact of a change in the MDG on the resultant partitioning. Integrating an Orphan Adoption[6] capability into our automatic clustering system can readily facilitate this capability.

Because fully automatic clustering techniques have these shortcomings, our goal is to leverage the benefits associated with fully automatic clustering techniques while providing facilities for specifying omnipresent modules, user-specified cluster overrides, and incremental structural maintenance (orphan adoption). In the next section of this paper, we illustrate how our clustering tool, named Bunch, can be used for recovering and maintaining the hierarchical structure of a software system.

3 The Bunch Clustering Tool

Figure 2shows the main window of the Bunch clustering tool. This window prompts the user to specify a clustering algorithm, the input MDG file and the name of the clustered output file. The three distinct clustering algorithms supported by Bunch make use of the MQ measurement and have been described elsewhere[3]. The input is a text file that contains a list of ordered pairs that specifies the relations between the source code entities in the software system. The clustered output file field specifies the name of the file that will be generated by Bunch. This file contains a text-based description of the partitioned MDG that can be read by the AT&T dotty tool[5] to visualize the results.

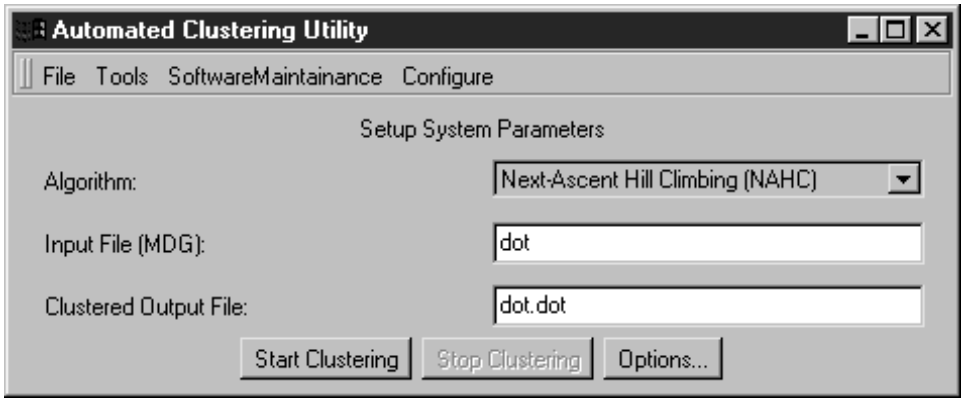


Figure 2: **The Main Window of Bunch**

The remainder of this section describes the features implemented in Bunch to support software maintenance.

3.1 Omnipresent Module Detection and Removal

Recall that omnipresent modules obscure the actual system structure because they either use, or are used by, many of the other modules in the system. Therefore, we wish to remove these modules and their associated edges from participating in the MQ calculation. Bunch enables the user to specify two different types of omnipresent modules. *Omnipresent Clients* are modules that behave like a driver because they use many of the other modules in the system. *Omnipresent Suppliers* are modules that behave like a library because they provide services to many of the other modules in the system. The specification of omnipresent clients and suppliers is a manual activity. However, Bunch includes a facility to automatically detect omnipresent modules.

Figure 3 illustrates the Bunch omnipresent module calculator. This feature enables the user to specify an omnipresent module threshold as a multiple of the average node degree in the MDG. Upon pressing the *Calculate* button, the omnipresent module calculator processes the MDG, and determines the average node degree for the graph. Once this has been accomplished, the MDG is searched for modules that have an in- or out-degree greater then the user-specified threshold. The results, which indicate candidate omnipresent modules, are displayed in the lower listbox. The user has the option to select the modules that will be treated as omnipresent.

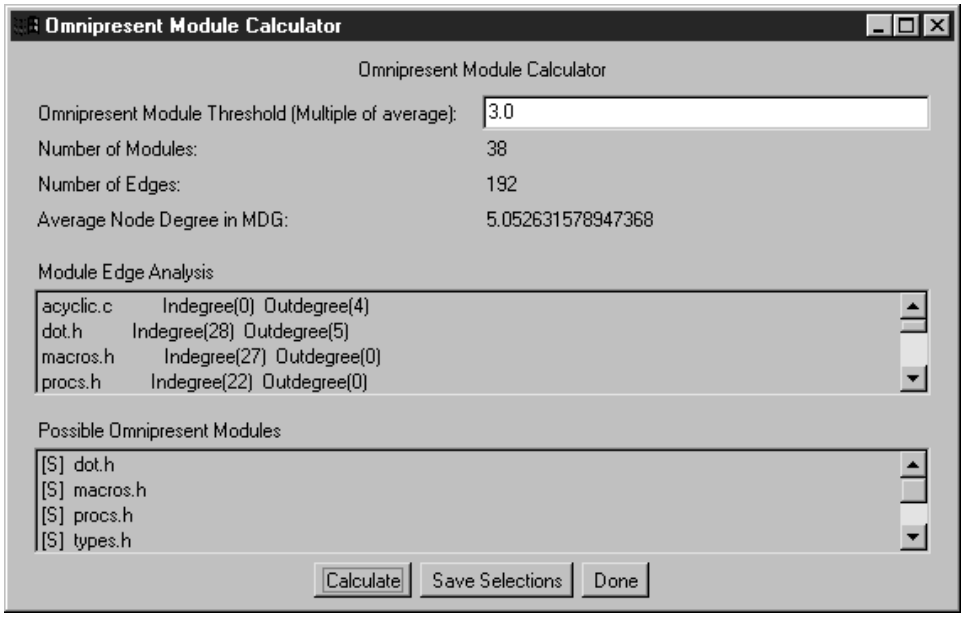


Figure 3: **Omnipresent Module Calculator**

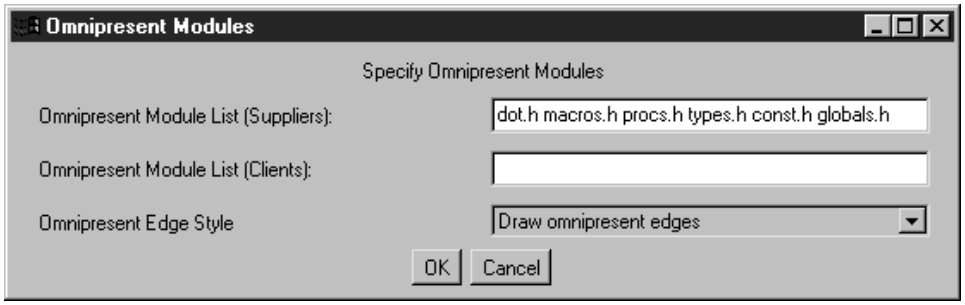


Figure 4: **Specifying Omnipresent Modules**

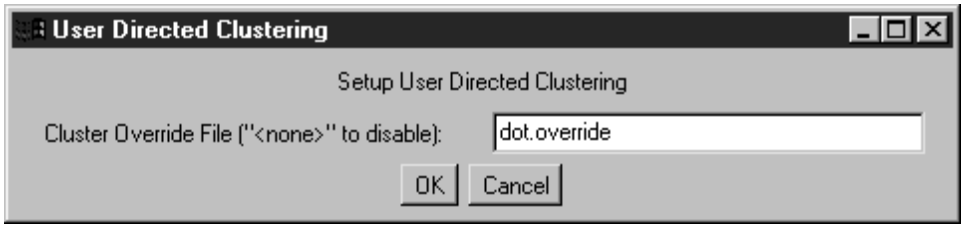


Figure 5: **User Directed Cluster Override Window**

Figure 4 illustrates the Bunch omnipresent module window. This window is used to specify the modules in the MDG that Bunch will treat as omnipresent. The user also has the option to specify how the edges into, and out of, the omnipresent modules will be represented in the generated output file. Bunch takes the modules specified in the omnipresent client and omnipresent supplier entry fields and creates special clusters to contain each of the omnipresent module types. The omnipresent edge style feature of Bunch allows the user to choose how the edges that connect the omnipresent modules to the rest of the modules in the system will be visualized. By default, Bunch will draw these edges in a different color, however, other options are supported for "lightly" drawing the edges, or removing the edges altogether from the output visualization.

3.2 User-Directed Cluster Overrides

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 override feature of Bunch enables users to manually cluster (known) modules based on their understanding of the system design, while taking advantage of the automatic clustering capabilities of Bunch to organize the remaining (unknown) modules.

Figure 5 illustrates the user directed clustering dialog box. The user specifies a file name that describes the known clusters in the software system. Bunch will respect the user-specified clusters while searching for an optimal partitioning of the MDG. Bunch, by default, will never remove modules from, or introduce new modules into the user-specified clusters. However, "locking" the user-specified clusters may not be always desirable; therefore, our system includes an option that enables the automatic clustering process to add modules to the user-specified clusters if the addition results in higher MQ.

3.3 Orphan Adoption

Determining the subsystem structure of a software system (by manual, semi-automatic, or fully automatic techniques) is an arduous task. Once the system organization is obtained, it is desirable to preserve the investment made in determining the subsystem structure as the system evolves. However, our fully automatic clustering process places modules into subsystems based on the topology of the MDG; therefore, a small change in the MDG can result in a large change in the partitioned result.

Orphan adoption techniques[6] address one aspect of incremental structural maintenance. By integrating the orphan adoption techniques into Bunch, designers are able to preserve a subsystem structure when:

- New modules are introduced into the system
- Existing modules modified resulting in the introduction of new dependencies between the updated module, and the other modules in the system.

Figure 6 shows the Orphan Adoption dialog box. The user is prompted for a file name that specifies the known subsystem decomposition of the software system. The input cluster file specifies a partitioning of the input MDG file. The user is also prompted to provide a list of orphan modules. An orphan module is either a new module that is being introduced into the system, or a module that has undergone some enhancements and has been removed from the known subsystem organization. Bunch facilitates the Orphan Adoption process by first measuring the MQ based on placing the orphan module 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 orphaned module. This process is repeated for each user-specified orphan module.

3.4 Obtaining Bunch

Interested readers may download a copy of Bunch, which was developed in Java, from the Drexel University Software Engineering Research Group (SERG) home page. The URL for SERG on the World Wide Web is http://www.mcs.drexel.edu/~serg.

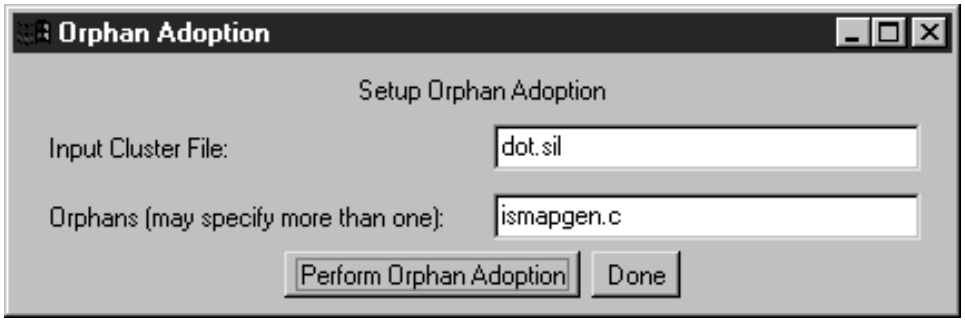


Figure 6: **Orphan Adoption Window**

4 Software Maintenance with Bunch - A Case Study

This section details the case study of using Bunch to recover the structure of the AT&T dot tool...

For your convenience, I have included the postscript files for the remodularization results.

5 Related Work

To be written by Spiros.

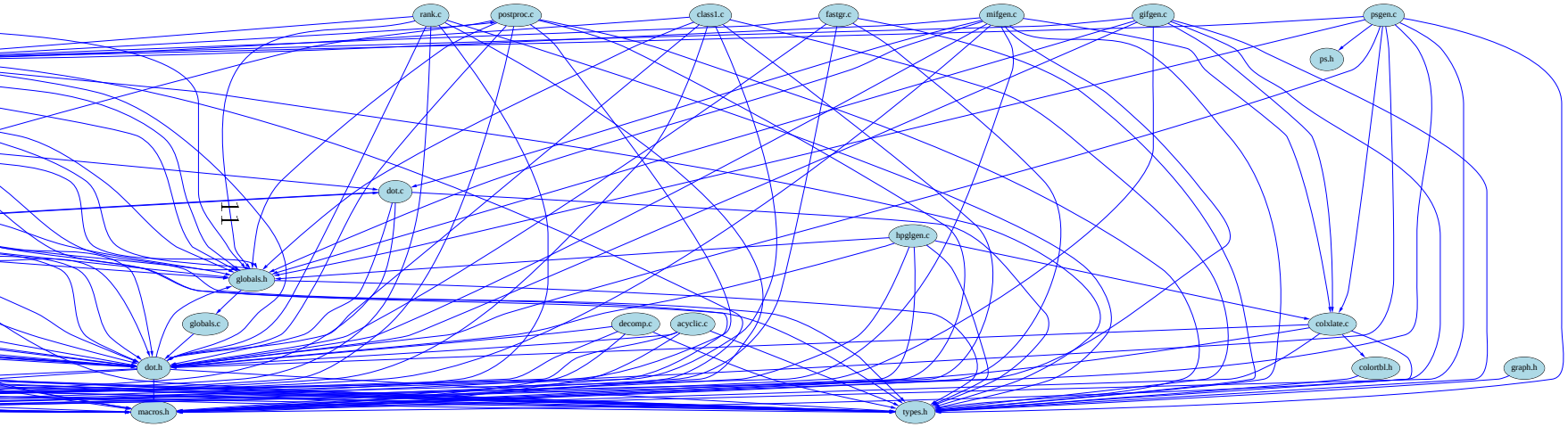
6 Conclusions and Future Work

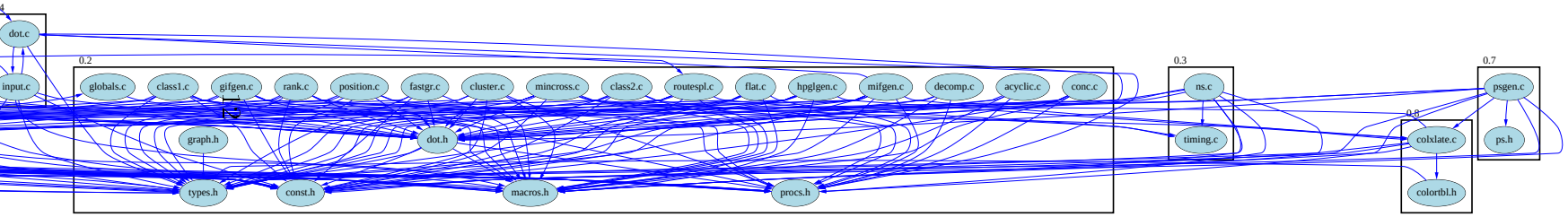
To be written by Spiros and Brian.

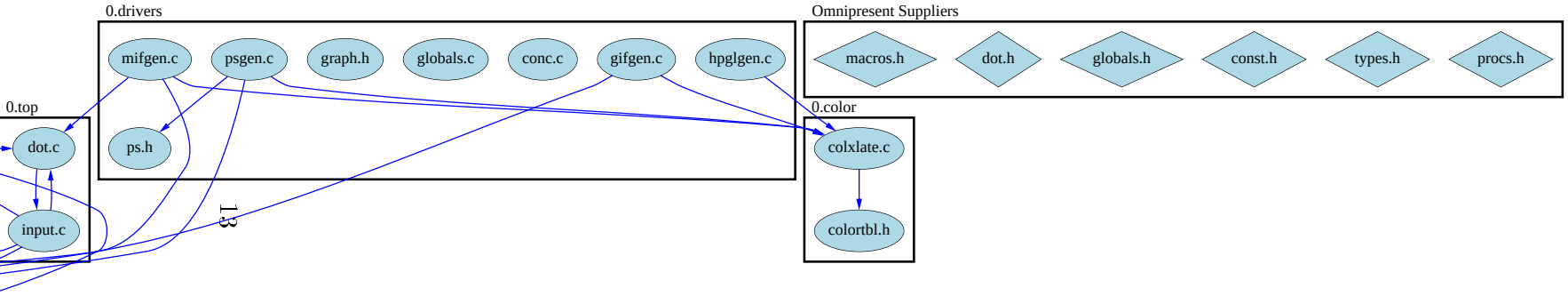
References

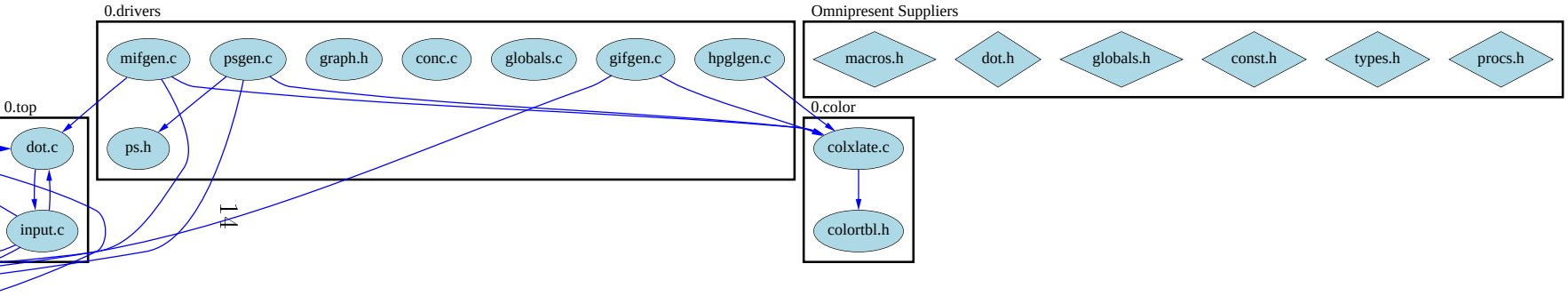
[1] Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

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


rank.c
postproc.c
class1.c
fastgr.c
mifgen.c
gifgen.c
psgen.c
ps.h
dot.c
hpglgen.c
globals.h
globals.c
decomp.c
acyclic.c
colxlate.c
dot.h
colortbl.h
graph.h
macros.h
types.h


dot.c
input.c
0.2
globals.c
class1.c
gifgen.c
rank.c
position.c
fastgr.c
cluster.c
mincross.c
class2.c
routespl.c
flat.c
hpglgen.c
mifgen.c
decomp.c
acyclic.c
conc.c
graph.h
dot.h
types.h
const.h
macros.h
procs.h
0.3
ns.c
timing.c
0.7
psgen.c
ps.h
0.8
colxlate.c
colortbl.h


0.drivers
Omnipresent Suppliers
mifgen.c
psgen.c
graph.h
globals.c
conc.c
gifgen.c
hpglgen.c
macros.h
dot.h
globals.h
const.h
types.h
procs.h
0.top
0.color
dot.c
ps.h
colxlate.c
input.c
colortbl.h
13


0.drivers
mifgen.c
psgen.c
graph.h
conc.c
globals.c
gifgen.c
hpglgen.c
ps.h
Omnipresent Suppliers
macros.h
dot.h
globals.h
const.h
types.h
procs.h
0.top
dot.c
input.c
0.color
colxlate.c
colortbl.h

[3] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension - IWPC98*, 1998.

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

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

[6] V. Tzerpos and R.C. Holt. The orphan adoption problem in architecture maintenance. In *Working Conf. on Reverse Engineering (WCRE97)*, 1997.