In Proceedings of the 8th Working Conference on Reverse Engineering, Oct. 2001, pp. 115-124, IEEE.

An Investigation into the Connectivity Properties of Source-Header Dependency Graphs*

Gerald C. Gannod† and Barbara D. Gannod
Department of Computer Science & Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: {gannod,bgannod}@asu.edu

Abstract

A modularization is a partitioning of a software system into components based on a variety of criteria, each depending on the clustering approach and desired level of abstraction. Source-header dependency graphs are bipartite graphs that are formed by flattening include file dependencies and enumerating source file to header file dependencies. In this paper, we describe an approach for identifying candidate modularizations of software systems by analyzing connectivity properties of source-header dependency graphs. In addition, we apply the approach to a large software system to demonstrate its applicability.

1 Introduction

The task of understanding, modifying, and maintaining large systems can be intimidating and frustrating, especially in environments where staff turnover rates are high. The cognitive models used to tackle these tasks consist of top-down, bottom-up and hybrid techniques [1]. Bottom-up approaches focus on the object of study with the perspective that direct source code analysis leads to formation of behavioral and structural abstractions [2]. Top-down approaches focus on successive refinements of candidate designs with verification and validation of recovered designs against the implemented software artifacts [3]. Hybrid approaches use a combination of both top-down and bottom up techniques in order to support a vertical analysis of a system.

Software artifacts are often organized at several different levels of discernible abstraction including files, procedures, and blocks. Each of these levels of abstraction present challenges to the maintainer. At the level of a file, the goal of the maintainer is to generate a high-level, informal model. At the level of a function and procedure, a maintainer is interested in constructing detailed structural models as well as determining the intended and actual behavior of the source code.

A *modularization* is a partitioning of a software system into components based on a variety of criteria, each depending on the approach and level of abstraction. Recent approaches have investigated the effectiveness of traditional clustering techniques [4, 5] and genetic algorithms [6] for modularization at the file level of abstraction.

A *source-header dependency graph* is a bipartite graph that is formed by flattening include file dependencies and enumerating source file to header file dependencies. In this paper, we describe an approach for identifying candidate modularizations of software systems by analyzing connectivity properties of source-header dependency graphs. These modularizations provide information that can lead to coarse-grained reuse of source code and are a perfect complement to our earlier work.

The remainder of this paper is organized as follows. Section 2 discusses background information on the areas of reverse engineering and graph theory. An approach for identifying modularizations, including a definition of the source-header dependency graph, is introduced in Section 3. Section 4 describes the application of the suggested approach on a portion of the source code for the Mozilla web browser [7]. Finally, related approaches are discussed in Section 5, and Section 6 draws conclusions and outlines future investigations.

2 Background

In this section, we describe the general area of reverse engineering as well as the formal framework that is used throughout this paper.

*This research supported in part by NASA Langley Research Grant NAG 1-2241.
†Contact author.

2.1 Reverse Engineering and Modularization

Reverse engineering is defined as the analysis of software components and their interrelationships in order to obtain a description of the software at a high level of abstraction [8]. This term is contrasted with reengineering, which is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [8]. Software reengineering is considered a potential solution for handling legacy code as opposed to developing software from the original requirements. Since the functionality of the existing software has been achieved over a period of time, it provides a means for preserving functionality and provides continuity to current users of the software.

In the context of software maintenance, we define a *structural* abstraction to be a description of a software system that is based on the syntactic and structural properties of a programming language or other system artifacts. For example, encapsulation of a sequence of programming statements into a module is a structural abstraction. In contrast, a functional abstraction is a description of a software system that is based on the semantics of a program. That is, a functional abstraction describes program behavior. For instance, if a sequence of statements is grouped into a module, then the high-level description of the function of that module is a functional abstraction. Recent work in reverse engineering has focused on both the derivation of structural and functional abstractions from program code. In this paper we focus entirely on structural abstractions in order to identify possible modularizations of software.

2.2 Graph Theory

A graph $G(V, E)$ is a finite, non-empty set of *vertices* V and a set (possibly empty) of *edges* E. As a convention, for a graph G, we use $V(G)$ to refer to the set of vertices of the graph, and $E(G)$ to refer to the set of edges. Below we define a number of terms that describe various properties, features, and graph related entities. Other concepts can be found in any standard graph theory text [9]. A *singleton* is a graph that contains only one vertex. A graph H is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. An *edge-induced* subgraph of G, denoted $\langle X \rangle$, is the minimal subgraph of G with edge set X. A *vertex-induced* subgraph of G, denoted $\langle S \rangle$, is the maximal subgraph of G with vertex set S. In this paper, we use the term *induced subgraph* to refer to both edge-induced and vertex-induced subgraphs; it will be clear from the context which definition is appropriate. A graph G is *bipartite* if $V(G)$ can be partitioned into two nonempty subsets V_1 and V_2 (called the *bipartitions* of G) so that every edge in $E(G)$ joins a vertex in V_1 with a vertex in V_2.

A graph G is *connected* if there is a path between any two arbitrary vertices in $V(G)$. A *component* G' of a graph G is a maximal subgraph that is itself connected. If there are two components G_1 and G_2 of a graph G, then no path exists between any arbitrary vertex in $V(G_1)$ and $V(G_2)$.

With respect to a vertex v, the *degree* indicates the number of edges that contain v as an end-point while the *neighborhood* of a vertex v is the set of vertices that share an edge with v.

3 Approach

In this section we define and discuss the analysis of source-header dependency graphs for the purpose of partitioning software systems into candidate modules.

3.1 Source-Header Dependency Graphs

In this paper, we consider the use of the *includes* relationship in order to define dependencies between source and header files.

3.1.1 Definition

In C and C++ source systems, the includes relationship is manifested in the use of the `#include` directive. The includes relationship encompasses a number of possible dependencies between source and header files:

Procedure Definition: The source file defines a procedure or function that is prototyped and exported by the corresponding header file.

Procedure Use: The source file uses a procedure or function that is prototyped in the corresponding header file.

Data Structure Use: The source file uses a data structure definition that is defined and exported by the corresponding header file.

Global Variable Use: The source file uses a global variable that is exposed by the corresponding header file.

Since any given instance of the includes relationship can represent many of the above dependencies, we assume that all relationships are present with the caveat that identifying the true dependencies is beneficial rather than detrimental to our approach. That is, discovery of the actual dependency that underlies the includes relationship facilitates construction of more detailed modularizations.

From the perspective of static relationships, the includes relationship is a hierarchical one. A graph showing source and header dependencies can be shown as a multi-level Hasse diagram as depicted in Figure 1. In the diagram, source file A includes header files B and C, header file B includes header files D and E, and header file C includes header files D and F.

Another potential perspective, as shown in Figure 2, is that of the compiler, where the includes relationship is

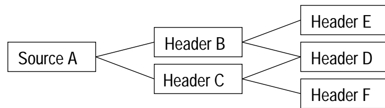

Figure 1. Multi-level Dependency Diagram

flattened into a single-level so that instead of showing the static relationship between source and header, all of the headers are depicted as having direct relationships to the source file.

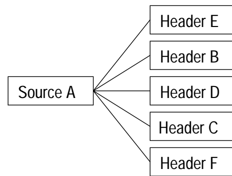

Figure 2. Single-level Dependency Diagram

In our approach, we consider source-header relationships only. Header-header relationships are flattened (as noted above) and thus are assumed to be transitive. In the case of source-source relationships, where functions and procedures in one source file *call* functions and procedures in another file, the relation can be captured as shown in Figure 3 where the dependency of source files A and B on header file A can represent a procedure definition dependency and a procedure use dependency.

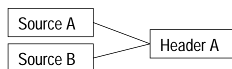

Figure 3. Source-Source Dependency

The focus on source-header relationships enables the construction of bipartite graphs that are structured with source files in one partition and header files in the other partition. As shown in Figure 4, a source-header dependency graph is formed by enumerating the dependencies of each file within a system. One advantage of using a bipartite graph to represent the relationships between files is that the number of potential relationships (e.g., edges) in the resulting graphs is reduced since source-source relationships are eliminated.

Figure 5 shows a portion of the Mozilla (Netscape) security subsystem using the aforementioned guidelines for constructing the dependencies between source and header files to construct a bipartite graph. This graph will be discussed and used more extensively in Section 4. Many of the source-header dependency graphs encountered in practice have similar properties to the one shown here with respect to cardinality of the vertex and edge sets as well as in the connectivity of the graphs.

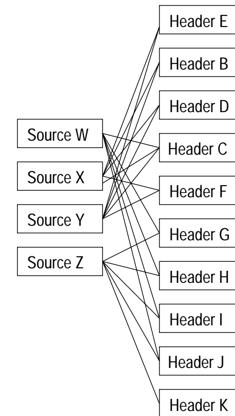

Figure 4. Bi-Partite Graph Depicting Source-Header Dependencies

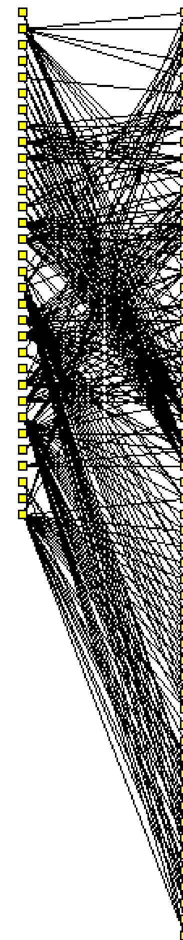

Figure 5. A Portion of the Mozilla Security Subsystem

3.1.2 Interpretation of Dependencies

There are several relationships that arise from the derivation of a source-header dependency graph in the manner described above. The relationships, with respect to source and header, may be $1:1$, $1:n$, $n:1$, and $n:m$.

As stated earlier, the $1:1$ relationship between two vertices of a source-header dependency graph represents an includes relation. In addition to the procedure definition, procedure use, data structure use, and global variable use dependencies, any given relation can represent a false dependency where a header file is included but its contents are never used.

The $n:1$ relationship between n source files and one header file represents several possible situations. With respect to procedures and functions, the $n:1$ relationship can represent the case where a set of procedures contained within a single source file are being *called* by procedures in other files. In this case, the header file contains the prototypes of the called procedures. The procedure definitions that implement the called procedures may be among the n source files, or may be contained elsewhere as in the case of libraries. Another interpretation of the $n:1$ relationship is the situation where the header file defines a data dependency. In this case the header file can either define a user-defined type that is exported to source files, or can declare globally accessible variables that are utilized by many routines. Finally, the $n:1$ relationship can describe the case where both data and call dependencies are present.

The $1:n$ relationship between one source file and n header files represents the case where a single source file utilizes information from several header files. For instance, the header files may provide "hooks" into libraries or other linkable resources (both procedures and data) that are all combined together by routines contained in the single source file.

The $m:n$ relationship between m source files and n header files represents the most general and frequent case. In the situation where the header files can be associated by a procedure definition dependency to n of the m source files, the $m:n$ relation indicates a tight coupling of procedures among the collection of m source files. In the case where the header files strictly represent a "procedure use" dependency, the $m:n$ relation indicates a tight coupling of procedures with some external set of *omnipresent routines* [6].

In most systems, files will individually be involved in $1:1$, $1:n$, and $n:1$ relationships. In addition, files collectively will be involved in $m:n$ relationships. With respect to the analysis of the bipartite source-header dependency graphs, the relationships described here provide the foundation for identifying the meaning of several graph theoretic properties. That is, these relationships are the basis for describing the impact of applying different graph algorithms on the source-header dependency graphs.

3.2 Module Identification Using Graph Connectivity

One of the goals of this research is to establish a foundation for applying graph algorithms and graph analysis on source-header dependency graphs in order to identify ways to partition source files into suitable modules. Elimination of source-source and header-header dependencies in graphs allows the analysis to focus on information that describes how interfaces to procedures and definitions of user defined data structures are exported to various parts of an application.

In this section, we describe an approach for identifying software modules via the analysis of source-header dependency graphs. Specifically, we examine the use of various graph algorithms for identifying bridges, edge-cuts, cut vertices, and cut-sets and their impact on identifying candidate modularizations at the source file level. In particular, we take advantage of the properties of high cohesion and low coupling as criteria for identifying modules.

3.2.1 Bridge Detection

A *bridge* in a graph $G(V, E)$ is an edge e whose removal results in a disconnected graph. That is, $\langle E(G) - \{e\} \rangle$ has two components, $G1$ and $G2$, such that no path exists between an arbitrary vertex in *G1* and an arbitrary vertex in *G2*.

Since every edge in a source-header dependency graph represents a dependency between a source file and a header file, the existence of a bridge in such graphs indicates a weak coupling between the components of the subgraph induced by the removal of the bridge. With respect to a modularization of source and header files, a bridge may indicate the existence of the procedure definition, procedure use, or data use dependencies and thus identifies potential exportation of behavior and data from one module to another. It is important to note that the existence of bridges in source-header dependency graphs, unlike the tree-like graphs typically used to represent software, is relatively rare [1].

Figure 6 shows a modularization of the graph from Figure 4 using a bridge. The diagram implies a dependency either via the use of data or from a potential call relationship between two source files as manifested via a prototype exported by a header file. In addition to identifying a partition between two sets of related files, a bridge also identifies a single point of entry between the modules and thus isolates exported interfaces.

[1] In a tree, every edge is a bridge.

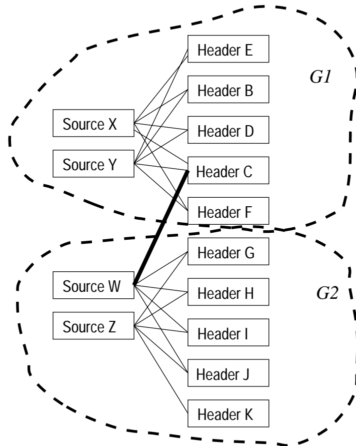

Figure 6. Graph Partitioned by a Bridge

With respect to detection of bridges within an arbitrary graph G, the basic algorithm works by exploiting the fact that an edge e is a bridge if and only if e does not lie on a cycle of G. Our implementation of bridge detection uses a simple depth first search to determine the status of each edge along the path of a cycle, and thus the complexity is $O(|E(G)| \cdot |V(G)|^2)$.

3.2.2 Edge-cut Detection

An *edge-cut* in a graph $G(V, E)$ is a set of edges *E'* whose removal results in a disconnected graph. Specifically, $\langle E(G) - E' \rangle$ has at least two components, $G1$ and $G2$, such that no path exists between an arbitrary vertex in *G1* and an arbitrary vertex in *G2*. In relation to edge-cuts, a bridge is an edge-cut of cardinality one. As such, the problem of detecting an edge-cut is a generalized version of bridge detection.

The interpretation of an edge-cut in the source-header dependency graph is similar to the case of the bridge. That is, the identification of an edge-cut, E', of size k (also known as a k-cut) induces a subgraph whose components are candidate modules. If we isolate any two components, $G1$ and $G2$, of $\langle E(G) - E' \rangle$, there are two classes of edges (in E') between $V(G1)$ and $V(G2)$ that must be considered. The first class of edge-cut is the case where all the edges identify a one-way relationship between two components $G1$ and $G2$. As shown in Figure 7, the edges in the edge-cut originate from sources in $G2$ and terminate at headers in $G1$. This one-way dependency can represent the case where one module represented by component $G2$ imports procedures contained within $G1$.

The second class of edge-cut, shown in Figure 8, represents the case where the dependencies are bidirectional so that source files in two components, $G1$ and $G2$, are dependent on headers in the sibling component. This two-way dependency represents the fact that two modules may share functionality across a partition.

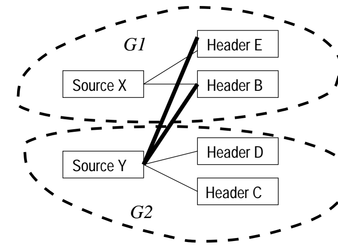

Figure 7. One-way edge-cut

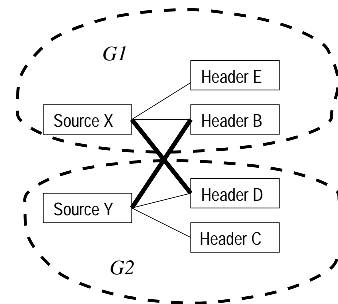

Figure 8. Two-way edge-cut

Jermaine discussed the general problem of identifying program modularizations using edge-cuts in program call graphs [10]. While the general problem is intractable, for certain restricted graphs, relatively efficient approximations are available. A major concern is the existence of many possible edge-cuts for any arbitrary graph. As such, any criteria that can be used to reduce the number of candidates is of interest.

3.2.3 Cut Vertex Detection

A *cut vertex* in a graph $G(V, E)$ is a vertex *v* whose removal results in a disconnected graph. That is, $\langle V(G) - \{v\} \rangle$ has at least two components, $G1$ and $G2$, such that no path exists between an arbitrary vertex in *G1* and an arbitrary vertex in *G2*.

There are three different configurations that are possible for cut vertices in a graph. The first case, as shown in Figure 9 is related to the bridges mentioned in the previous section. Specifically, each of the end-vertices of a bridge is a cut vertex since removal of either vertex will result in removal of the bridge. With respect to component modularizations, these cut vertices have the same interpretation as the bridge and act as an interaction point (e.g., port) between two modules.

The second kind of cut vertex in a source-header dependency graph arises when removal of a source vertex disconnects the graph. As shown in Figure 10, when a source

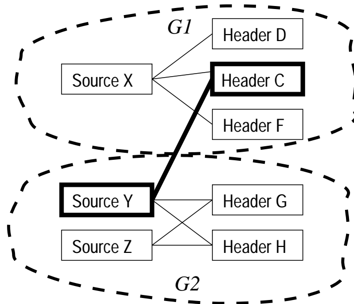

Figure 9. End-Vertices of a Bridge

vertex is also a cut vertex, it is dependent upon header files in at least two different components. As a result, the procedures contained in the file represented by the cut vertex can be interpreted as providing services from one component to another, as providing services to both components, or as a user of services from both components.

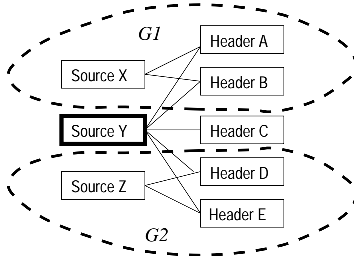

Figure 10. Source as Vertex Cut

Figure 11 depicts a header file as a cut vertex of a source-header dependency graph. One of the interpretations for a header file existing as a cut vertex is that the header may be an omnipresent file that provides system-wide or application wide exposure to a common procedure or service. Another interpretation of the header file is that it describes services that are provided and used both internally and externally by the components that are induced by its removal.

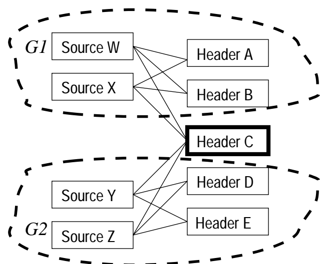

Figure 11. Header as Vertex Cut

The brute force implementation of cut vertex detection is to remove a vertex and check to see whether the remaining graph is connected. The complexity of such an algorithm is $O(|V(G)|^3 \times \Delta(v))$, where $\Delta(v)$ is the maximum degree of a vertex in $V(G)$. Since G is bipartite, $\Delta(v)$ is bounded by the cardinality of the larger of the two bipartitions of G.

3.2.4 Cut-set Detection

A *vertex-cut*, or simply *cut-set*, in a graph $G(V, E)$ is a set of vertices *V'* whose removal from the graph *G* results in a disconnected subgraph. Specifically, $\langle V(G) - V' \rangle$ has at least two components, $G1$ and $G2$, such that no path exists between an arbitrary vertex in *G1* and an arbitrary vertex in *G2*. Figure 12 shows a graph with a highlighted cut-set such that removal of the vertices induces the components shown in Figure 13.

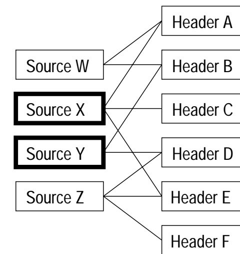

Figure 12. Cut Set of Graph

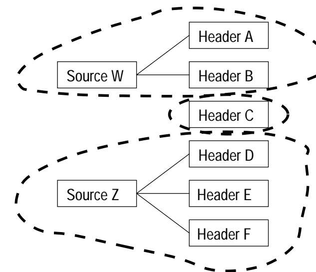

Figure 13. Components Induced by Cut Set

As in the case of the cut vertex, there are three different types of cut-sets that can be identified. The first, is a *source only* cut-set where the vertices are made up entirely of source files. An analogous cut-set consists of headers only. A third class of cut-sets is a heterogeneous cut consisting of both source and header files.

The interpretation of a headers only cut-set can be considered to be a generalization of the header cut vertex in the previous section. That is, it is possible to interpret a header only cut-set as a set of omnipresent files that are shared among many components in a system. This is especially true if the relationships between source files and the

headers in the cut-set are found to be of the use dependency variety.

The interpretation of a source only cut-set is a generalization of the source cut vertex, where the vertices in the cut-set can potentially be providing services from one component to another, providing services to both components, or using services from both components.

The interpretation of a heterogeneous cut-set gives rise to a situation where the cut-set itself potentially forms a modularization of the source system, as shown in Figure 14. Further study of this phenomenon is needed, but initially it appears that the pairwise analysis of components that result from heterogeneous cut-sets is similar to the analyses of the various connectivity properties described in this paper.

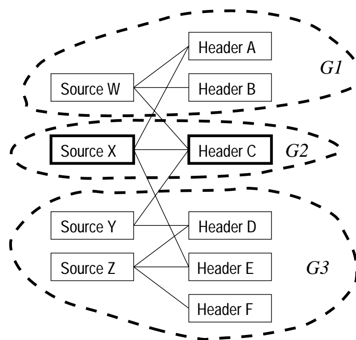

Figure 14. Cut Set as Component

The brute force approach for finding all cut-sets in a graph is to remove all possible subsets of vertices (all subsets in the power set of V(G)) from the graph, and then check if the remaining graph is connected. The worst case complexity of the algorithm is $O(2^{|V(G)|} \cdot max\{|V(G)|, |E(G)|\})$. Note that the brute force algorithm described would find cut vertices as well as cut-sets. More elegant algorithms exist [11] although the worst case complexity is similar. We are currently developing criteria for pruning the search space by focusing on cut-sets with more semantically meaningful interpretations as well as exploring ways to take advantage of the bipartite properties of source-header dependency graphs.

3.3 Discussion

The discussion in Section 3.2 uses the premise that we are interested in identifying components of a graph that are induced by the removal of features such as bridges, cut vertices, edge-cuts and cut-sets. With respect to the induced graph components, we are interested in non-singletons. For instance, in the source-header dependency graphs, for any vertex v with degree n, a candidate edge-cut of size n is one that includes all of the edges from v to the neighborhood of v. As a result, the subgraph induced by the removal of these edges has two components, $\langle v \rangle$ (a singleton) and $\langle V(G) - \{v\} \rangle$, each of which is of little interest regarding a modularization of the original graph.

Many reverse engineering techniques that are based on the identification of structural abstractions via clustering use a *tree* or *pseudo-tree* structure. That is, in the case of a tree, the graphs are acyclic. As a result, removal of any edge or vertex disconnects the graph. By using a bipartite graph, our approach makes the bridge, cut vertex, and other similar features meaningful.

3.4 Tools

To support the approach described in this paper, we have developed a number of tools that facilitate the construction and analysis of bipartite source-header dependency graphs. Specifically, we have created three tools:

srcdep: A unix filter designed to take the output of gcc to produce a file defining a source dependency graph as a set of vertices and edges

Bridge: A java application designed to identify the bridges of a graph

Cut: A java application designed to identify the cut vertices of a graph

The filter `srcdep` takes gcc output (with the -E -MG -MM options) and along with unix commands `sed` and `tr`, is used to produce a graph definition file in the format of *vcg* [12]. The resulting graph is a bipartite graph partitioned by source and header.

The `Bridge` program takes as input the file produced by `srcdep` and identifies the bridges of the input graph. The `Bridge` program will also graphically display the induced components of the graph (e.g., the subgraphs induced by removing the bridges). When used successively upon a graph, the `Bridge` program can identify candidate edge cuts.

The `Cut` program, like the `Bridge` program, takes as input the file produced by `srcdep` and identifies the cut vertices of the input graph. In addition, the `Cut` program will display the induced components of the graph that are formed by removing the cut vertices. Both the `Bridge` and `Cut` programs were constructed using the IBM Graph Foundation Classes (GFC) [13].

4 Example

In this section we discuss the application of the approach described in this paper to a portion of the Mozilla software system [7]. Specifically, we analyzed a portion of the Network Security Subsystem (NSS) including a number of applications used to support server applications. The process that was used to perform this analysis includes three basic steps: 1) extraction of dependency information,

2) construction of the source-header dependency graph, and 3) analysis of the source-header dependency graph.

4.1 Extraction and Graph Construction

The process of extracting information about file dependencies in a software system can be performed by constructing a macro pre-processor, using an existing pre-processor, or by examining compiler output. The advantage of using compiler output is that the information is directly attainable and does not require completely compilable source code. In addition, since the verbose option in many compilers supports generation of our desired target information, the approach can be easily replicated for a wide variety of languages and platforms.

For the example, we performed a file-by-file compilation and captured the dependency information in intermediate files that were later merged to form the input to the `srcdep` program. Using standard unix commands such as `sed` (a stream editor) and `tr` (a simple line translator), the entire process can be (and has been) automated.

Currently, the output of the `srcdep` program is a formatted file that is compatible with the *vcg* tool [12]. Our plans include modifying the output to support the GXL language [14]. Figure 5 shows the graph that resulted from the extraction and construction step. The left-hand partition of the graph depicts the source files that were included in the analysis while the right-hand partition depicts the header files.

4.2 Analysis

Given the graph shown in Figure 5, the next stage of the approach is the analysis step. In this example, the `Bridge` program was used initially to identify bridges in the graph. The results of the analysis identified a number of bridges, one of which was interesting. The remaining bridges resulted in creation of singletons which are summarily dismissed. Figure 15 shows the graph that is induced by removal of the bridges.

One of the interesting aspects of the graph is the difference in the topologies of the induced (non-singleton) components. The upper component in the graph has properties similar to the original graph in that the cardinality of the source partition is smaller than the cardinality of the header partition. In contrast, the lower component has the inverse property of having a larger source partition and smaller header partition.

The bridge between these components (not shown in the graph) connects a source file from the upper component with a header file in the lower component. As a result, the interpretation of the graph is that the lower component provides some service to the upper component. At the moment, our approach does not reveal the true nature of a dependency but such information would provide a better interpretation of the component relationships.

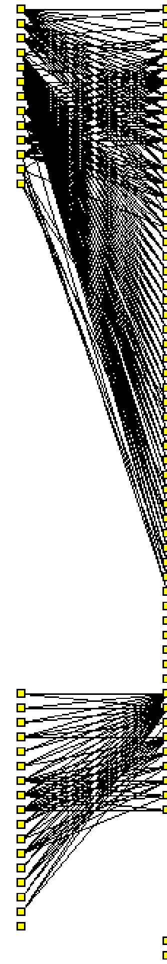

Figure 15. Graph Induced by Removal of Bridges

Bridge detection in a graph provides a relatively quick first pass through a graph to identify loosely coupled modules within a system. However, it is entirely possible that few bridges exist in a graph, if any. In the case of cut vertices, the situation is analogous in that detection of a cut vertex is relatively simple, although a graph may not have a cut vertex.

The next step in an analysis is to recursively apply any of the given connectivity detection methods to the induced components. We are currently in the process of developing a set of criteria for determining ideal parameters for cut set and edge cut algorithms.

Figure 16 shows the induced subgraphs of the NSS subsystem after applying basic criteria for identifying an edge cut upon the upper component of the graph from Figure 15. The graph shows the existence of a component that is induced by removing an edge cut of size 5. This particular

edge cut was a one-way cut from the upper component to the middle component.

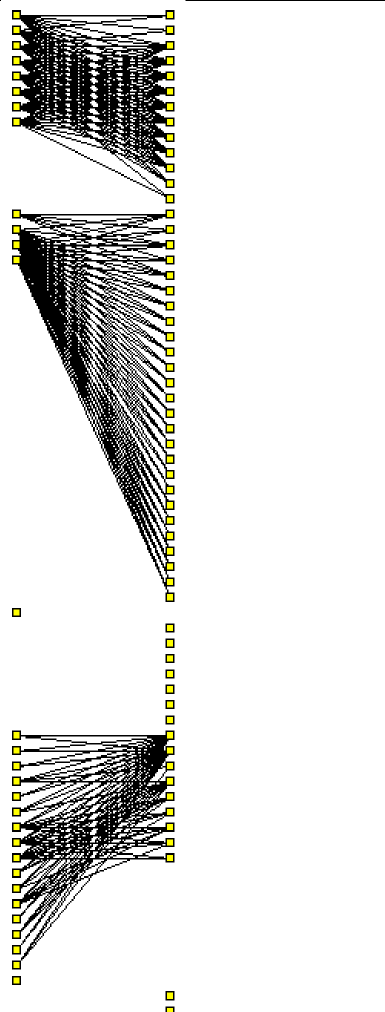

Figure 16. Graph Induced by Applying Edge Cut Criteria

As stated earlier, the approach described in the paper is purely a structural technique. Accordingly, the resulting approach does not provide a clear assignment of behavior to each of the partitions. Furthermore, while the method and techniques that we use to identify the components provides a partitioning that is easily identifiable from the graph, there needs to be some method for determining the associated behavior of the resulting modules. However, by creating the partitions, the system is decomposed into more manageable pieces that can be used as the entry point for applying other analysis techniques that work at both the call graph [15] and function level [16].

5 Related Work

Several approaches have been suggested for modularization of software into components. At the lowest level of granularity, where small code fragments are clustered using concept analysis, approached include those suggested by Lindig and Snelting [17] and Siff and Reps [18]. At the call graph or pseudo-call graph level, techniques include those presented by Tzerpos and Holt [19], Davey and Burd [4], and Choi and Scacchi [20], among others. While each of these approaches perform clustering in some manner, each operates at levels of granularity that are much finer than the focus presented in this paper.

Jermaine [10] describes a graph theoretical approach for identifying k-cuts within a call graph as a means for determining software modularizations. Specifically, Jermaine discusses the application of several algorithms for building an approximation for computing k-cuts. In addition, the author defines a *sociability* metric for identifying appropriate values for k. Our approach is similar to the one suggested by Jermaine in the sense that it relies on observable properties of graphs that represent software structure. However, our approach differs in the level of granularity (e.g., call graphs vs. file dependencies) as well as in the types of the graphs being studied (e.g., pseudo-trees vs. bipartite graphs).

Anquetil and Lethbridge [5] describe several similarity metrics for the purpose of identifying clusters at the file level. Using a number of features including formal descriptors (e.g., features directly impacted by program behavior) and characterizations of links between files, the approach applies clustering algorithms to create system partitions. Our approach differs in the sense that we focus entirely upon graph theoretic concepts to identify clusters and do not currently consider descriptors that may potentially impact the quality of a modularization.

Mancoridis et al. [6, 21] have developed an approach for clustering software systems at the file level to identify candidate components. Of the approaches surveyed, this technique most closely matches ours in the level of granularity due to its focus on file level structures. However, the approach differs from ours in a couple of ways. First, the input to their approach is a tree-like structure and thus is limited in its ability to use graph-based features such as bridges and cuts. Secondly, their approach utilizes a genetic algorithm that is governed by similarity measures for finding candidate clusters. Our approach, in contrast focuses solely upon graph algorithms to identify clusters.

6 Conclusions and Future Investigations

In this paper, we describe an approach for identifying candidate modularizations at the file level. In particular, this paper introduces the use of a source-header dependency as the primary medium for applying graph algorithms that reveal connectivity properties in bipartite graphs. The approach utilizes the notions of high cohesion

and low coupling as the main criteria for justifying the resulting modularizations.

In previous work, we developed an approach for supporting coarse-grained reuse of applications via the construction of application wrappers that are integrated at run-time using a Jini-based framework [22]. The work described in this paper complements our application level reuse technique by providing a mechanism for identifying coarse-grained partitions within large systems as well as the entry points (e.g., headers) that are (and have been) used to publish procedure interfaces and data definitions. As a result, we are able to identify possible modularizations that expose dependencies that may exist beyond traditional partitions. As such, these modularizations are candidates for coarse-grained reuse under certain conditions (notwithstanding understood behavior).

Future investigations in this area include development of approaches for introducing lower-level information in order to remove false dependencies as well as filtering certain classes of dependencies. Specifically, we are interested in studying characteristics of connectivity and how they are affected by information revealed by removing file-based abstractions and introducing behavioral specifications such as those used in our previous work [16]. In addition, we are interested in expanding the investigations described here by studying in detail how the approach compares to other similar techniques by conducting formal experiments to determine the quality of the clusterings.

References

[1] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *Transactions on Software Engineering*, SE-10(5):595–609, September 1984.

[2] Ira D. Baxter and Michael Mehlich. Reverse Engineering is Reverse Forward Engineering. In *Proceedings of the Fourth IEEE Working Conference on Reverse Engineering*. IEEE, October 1997.

[3] Gail C. Murphy and David Notkin. Reengineering with Reflexion Models: A Case Study. *Computer*, 30(8):29–36, August 1997.

[4] J. Davey and E. Burd. Evaluating the suitability of data clustering for software remodularisation. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 268–276. IEEE, November 2000.

[5] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 235–255. IEEE, October 1999.

[6] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch:a clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the 1999 International Conference on Software Maintenance*. IEEE, August 1999.

[7] Mozilla.org. [Online] Available http://mozilla.org.

[8] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[9] Gary Chartrand and Ortrud R. Oellermann. *Applied and Algorithmic Graph Theory*. McGraw-Hill, 1993.

[10] Christopher Jermaine. Computing Program Modularizations Using the *k*-Cut Method. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 224–234. IEEE, October 1999.

[11] V.V. Rao, P. Sankaran, K. S. Rao, and V. G. Murti. Enumeration of all cutsets of a graph. *Proceedings of the IEEE*, 56:1247–1248, July 1968.

[12] G. Sander. Graph layout through the vcg tool. In *Proceedings of Graph Drawing, DIMACS International Workshop GD'94, Lecture Notes in Computer Science*, volume 894, pages 194–205. Springer-Verlag, 1995.

[13] Graph foundation classes for java. [Online] Available http://www.alphaworks.ibm.com/tech/gfc.

[14] Richard C. Holt, Andreas Winter, and Andy Schurr. Gxl: Towards a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, November 2000.

[15] Scott R. Tilley, Kenney Wong, Margaret-Anne Storey, and Hausi A. Müller. Programmable Reverse Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[16] Gerald C. Gannod and Betty H. C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996.

[17] Christian Lindig and Gregor Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 349–359. ACM, 1997.

[18] Michael Siff and Thomas Reps. Identifying Modules via Concept Analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, November/December 1999.

[19] V. Tzerpos and R. C. Holt. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 258–267. IEEE, November 2000.

[20] Song C. Choi and Walt Scacchi. Extracting and Restructuring the Design of Large Systems. *IEEE Software*, 7(1):66–71, January 1990.

[21] S. Mancoridis and R. Holt. Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering. In *Proceedings of the 1999 International Conference on Software Maintenance*, pages 23–32. IEEE, 1996.

[22] Gerald C. Gannod, Sudhakiran V. Mudiam, and Timothy E. Lindquist. An Architectural Based Approach for Synthesizing Wrappers for Legacy Software. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 128–137. IEEE, November 2000.