Supporting Program Comprehension Using Semantic and Structural Information

Jonathan I. Maletic, Andrian Marcus
Division of Computer Science
Department of Mathematical Sciences
The University of Memphis
Campus Box 523240
Memphis, TN 38152, USA
+01 901 678 3140
jmaletic@memphis.edu, amarcus@memphis.edu

Abstract

The paper focuses on investigating the combined use of semantic and structural information of programs to support the comprehension tasks involved in the maintenance and reengineering of software systems. Here, semantic refers to the domain specific issues (both problem and development domains) of a software system. The other dimension, structural, refers to issues such as the actual syntactic structure of the program along with the control and data flow that it represents. An advanced information retrieval method, latent semantic indexing, is used to define a semantic similarity measure between software components. Components within a software system are then clustered together using this similarity measure. Simple structural information (i.e., file organization) of the software system is then used to assess the semantic cohesion of the clusters and files, with respect to each other. The measures are formally defined for general application. A set of experiments is presented which demonstrates how these measures can assist in the understanding of a nontrivial software system, namely a version of NCSA Mosaic.

1. Introduction

Program comprehension is a complex task. The software engineer must examine both the structural aspect of the source code (e.g., programming language syntax) and the nature of the problem domain (e.g., comments, documentation, and variable names) to extract the information needed to fully understand any part of a software system [7, 13, 31, 42, 45]. A number of tools and methods [1, 7, 8, 20, 27, 30, 41] have been investigated to address both of these aspects. In general,

structural information is easy to extract, but the real problem is on how to utilize that information properly. Semantic information, on the other hand, is much more difficult to extract. Knowledge-based systems, of one form or another have often been used to address this problem. Typically, a knowledge base of programming plans or schemes is constructed and then used to automatically identify concepts in a program. But, there is an inherent difficultly in the use of knowledge bases, namely someone must construct them. Assembling such domain specific knowledge is very time consuming and expensive.

In the research presented here, we take the approach of using cheaper but less accurate methods to extract semantic information. Specifically, we are investigating how well information retrieval methods can be used to extract relevant semantic information from software. The PROCSSI[1] system uses an advanced information retrieval technique, Latent Semantic Indexing (LSI), to identify semantic similarities between pieces of source. This semantic similarity measure is used to cluster software components. The paper presents a model that also incorporates structural information to assist in the comprehension task. A set of experiments is presented which demonstrates how these measures can be utilized in the understanding of a nontrivial software system, namely a version of Mosaic [36]. Finally, conclusions are drawn based on these experiments and future research directions are discussed.

2. Information retrieval and software

There are a variety of information retrieval methods including traditional [14, 43] approaches such as signature

---

[1] PROCSSI is short for PROgram Comprehension Combining Semantic and Structural Information. We pronounced it "proxy".

files, inversion, and clustering. Other methods that try to capture more information about documents to achieve better performance include those using parsing, syntactic information, natural language processing techniques, methods using neural networks, and advanced statistical methods. Much of this work deals with natural language text and a large number of techniques exist for indexing, classifying, and retrieving text documents. These methods produce for each document a profile. A profile is an abbreviated description of the original document that is easier to manipulate.

The research that has been conducted on the specific use of applying information retrieval methods to source code and associated documentation typically relates to indexing reusable components [15, 16, 18, 28, 29, 32, 35]. Notable is the work of Maarek [28, 29] on the use of an IR approach for automatically constructing software libraries. The success of this work along with the inefficiencies and high costs of constructing the knowledge base associated with natural language parsing approaches to this problem [12, 13] are main motivations behind our research. In short, it is very expensive (and often impractical) to construct the knowledge base(s) necessary for parsing approaches to extract even reasonable semantic information from source code and associated documentation. Using IR methods (based on statistical and heuristic methods) may not produce as good of results, but they are inexpensive to apply and coupled with the structural information of the program, should produce good quality and low cost results.

2.1. Latent semantic indexing

Latent Semantic Indexing (LSI) [5, 25] is a corpus-based statistical method for inducing and representing aspects of the meanings of words and passages (of natural language) reflective in their usage. The method generates a real valued vector description for documents of text. This representation can be used to compare and index documents using a variety of similarity measures. By applying LSI to source code and its associated internal documentation (i.e., comments), candidate components can be compared with respect to these similarity measures. Results have shown [5, 25] that LSI captures significant portions of the meaning not only of individual words but also of whole passages such as sentences, paragraphs, and short essays. The central concept of LSI is that the information about word contexts in which a particular word appears or does not appear provides a set of mutual constraints that determines the similarity of meaning of sets of words to each other.

LSI relies on a Single Value Decomposition (SVD) [40, 46] of a matrix (word × context) derived from a corpus of natural text that pertains to knowledge in the particular domain of interest. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of a feature space without serious loss of specificity. Typically, the word by context matrix is very large and (quite often) sparse. SVD reduces the number of dimensions without great loss of descriptiveness. Single value decomposition is the underlying operation in a number of applications including statistical principal component analysis [22], text retrieval [6, 11], pattern recognition and dimensionality reduction [10], and natural language understanding [25]. For complete details of Latent Semantic Indexing see [9].

The resulting profile is that each word is represented as a vector in a d-dimensional space. Performance depends strongly on the choice of the number of dimensions. The optimal number is typically around between 250 and 350 and may vary from corpus to corpus, domain to domain. The similarity of any two words, any two text passages, or any word and any text passage, are computed by measures on their vectors. Often the cosine of the contained angle between the vectors in d-space is used as the degree of qualitative similarity of meaning. The length of vectors is also useful as a measure.

One of the criticisms of this method, when applied to natural language texts is that it does not make use of word order, syntactic relations, or morphology. But very good representations and results are derived without this information [6]. This characteristic is very well suited to the domain of source code and internal documentation. Because much of the informal abstraction of the problem concept may be embodied in names of key operators and operands of the implementation, word ordering has little meaning. Source code is hardly English prose, but through the use of selective naming, much of the high level meaning of the problem at hand is conveyed to the reader (programmer/developer). Internal source code documentation is also commonly written in a subset of English [13] that may also lend itself to the IR methods utilized.

A fundamental deficiency of a number of IR methods is that they fail to deal properly with two major issues: synonymy and polysemy. Synonymy is used in a very general sense to describe the fact that there are many ways to refer to the same object. People in different contexts, with different knowledge, or linguistic habits will describe the same information using different terms. Polysemy refers to the general fact that most words have more than one distinct meaning. In different contexts or when used by different people the same term takes on varying referential significance [9]. Although software developers may tend to use standard terms for the concepts they are working on, a flexible technique capable to deal with variability is needed. It has been shown that LSI tends to address these issues [25]. Also, like some other IR methods LSI does not utilize a grammar or a predefined vocabulary. Though, many IR

methods do use a list of non-essential words with low discriminatory power. This makes automation much simpler and supports programmer defined variable names that have implied meanings (e.g., avg) yet are not in the English language vocabulary. The meanings are derived from usage rather than a predefined dictionary. This is a stated advantage over using a traditional natural language approach, such as in [12, 13], where a (subset) grammar for the English language must be developed.

3. Clustering source code components

Clustering of source code based on semantic and structural information is very useful in the maintenance and evolution of legacy software systems. For instance, the clustering can be used to assist in the re-modularization [37, 38, 49] of systems and the identification of abstract data types [8, 17]. If the system were to be reengineered into an object-oriented language from a structured one, this type of clustering would prove to be very useful. The objective is to reduce the amount of source code an engineer needs to view at one time and give them clues about possible relationships with the system not apparent from the current organization of the files or documentation.

The work presented here focuses on using the profile generated by IR methods, in this case a vector representation from LSI, to compare components and classify them into clusters of semantically similar concepts. Given a software system, it can be broken down in to a set of individual source code documents. Profiles for each document are then generated by the IR method. To cluster the source code documents they are partitioned based on similarity value λ with respect to the other documents, in the semantic space. There is a variety of clustering algorithms and they can be divided broadly into four categories: graph theoretical algorithms, construction algorithms, optimization algorithms, and hierarchical algorithms. There are also several hybrid methods that use ideas from different categories for specific problems. Here, a simple graph theoretic approach is used, but a number of other types of clustering algorithms have been used to cluster software [3, 4, 21, 28].

A minimal spanning tree (MST) algorithm [23] is used to cluster the documents based on a given threshold for the similarity measure. A document is added to a cluster if it is at least λ similar to any one of the other documents in the cluster. This strategy attempts to group as many documents together within the given similarity range. The similarity measures are computed by the cosine of the two vector representations of the source code documents. The similarity value therefore has a domain of [-1, 1], with the value 1 being "exactly" similar.

A simple parsing of the source code is done to break the source into the proper granularity and remove any non-essential symbols. Comment delimiters and many syntactical tokens are removed as they add little or no semantic knowledge of the problem domain. Also, the LSI method inherently will see such ubiquitous tokens such as a semi-colon as a totally non-discriminating feature between to source code components. That is, every meaningful C++ component contains a semi-colon. Therefore, the variance of this feature is very low (most likely zero) thus; if two components have a semi-colon then nothing can be said about their similarity.

The granularity of the source code input to LSI is of interest at this point. In the applications of LSI on natural language corpuses, typically a paragraph or section is used as the granularity of a document. Sentences tend to be to small and chapters too large. In source code, the analogous concepts are function, structure, module, file, class, etc. Obviously, statement granularity is too small and a file containing multiple functions may be too large.

3.1. Previous Experiments using LSI

In previous experiments done by the authors, the function and class declaration levels have been used [32]. Two readily available software systems were used as data for the experiments: LEDA [26] (Library for Efficient Data structures and Algorithms) and MINIX [47] (Operating System). LEDA is a library of the data types and algorithms for combinatorial computing and provides a sizable collection of data types and algorithms in a form that allows them to be used by non-experts. LEDA is composed of over 140 C++ classes. MINIX is a simple version of the UNIX operating system and widely used in university level computer science OS courses. It is written in C and consists of approximately 28,000 lines of code. Given that LEDA is written in C++ using an object-oriented methodology the granularity chosen is that of the class LEDA has 144 source code documents. For MINIX the function level is used along with some whole files that are made up of data structure definitions. This resulted in 498 source code documents for MINIX.

The previous work supported the concept of using LSI as a similarity measure for clustering software at a given level of granularity, namely a class or function level [32]. The clusters automatically produced by this method tended to reflect the reality of the source code [32]. Pieces of source code that had large amounts of semantic similarity were in general grouped together and modules with no relation to others remained apart. The clusters in the LEDA library seem to reflect class categories, that is, groups of related classes that function on similar concepts or solve common types of problems. In the MINIX system, the clusters are quite different due to the different methodology and programming language utilized. In this case, the clusters represented sets of documents that

represent a class or abstract data type. The larger clusters are typically composed of one or two data structure definitions and a number of functions that utilize these data structures.

While these experiments support the use of LSI to source code, the fact is that both of these software systems are very well written, documented, and organized. Also, neither of these systems is very large. In general, one does not need complex tools to help in understanding these types of software systems. We will demonstrate the use of these methods on a more real world type problem in a following section.

We are developing the PROCSSI system as an experimental platform in order to test these methods usefulness to the general problem of program comprehension. This system utilizes a number of metrics and measures derived from the semantic information produced from LSI. These metrics are now described.
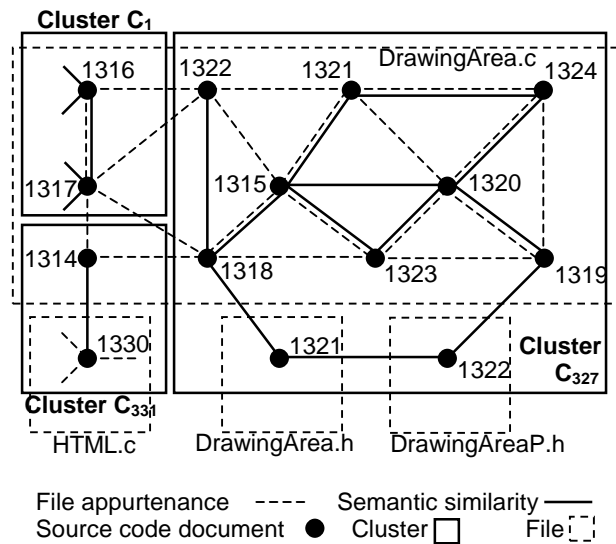

Figure 1: A part of the relationship graph representing Mosaic. The files DrawingArea.c, DrawingArea.h and Drawing AreaP.h are shown entirely. Not all edges shown.

4. Metrics for comprehension

The PROCSSI system uses a graph theoretic approach to define metrics that will be used in the comprehension task. Our choice of representation is a multi-graph, similar to how a data flow or control flow graph is represented (see figure 1). Each node represents a source code document. The document relates to the level of granularity used for clustering. We automatically label the nodes with a unique document number generated during parsing. But in the best case, the nodes could be labeled with the name of the document (e.g., function name, class name, etc.), which can be derived from the associated source . Below are some basic definitions of the model.

Definition. A *source code document* (or simply document) *d* is any contiguous set of lines of source code and/or text. Typically, a document is a function, block of declarations, definitions, or a class declaration including its associated internal documentation (comments).

Definition. A *software system* is a set of documents *S* = $\{d_1, d_2, \ldots, d_n\}$. The Total number of documents in the system is n = |*S*|.

Definition. A *cluster*, c_k, is a set of documents from *S* such that $c_k \subseteq S$. Size of a cluster, c_k, is the number of documents in a cluster, noted $|c_k|$.

Definition. A *file* f_i, is then composed of a number of documents and the union of all files is *S*. Size of a file, f_i, is the number of documents in the file, noted $|f_i|$. A file is a cluster defined by the developers.

Definition. A *relationship graph* is represented as a multi-graph *G* = (*S*, *E*), where the nodes *S* are the documents, *E* is a set of weighted edges, and a function *e*: $E \rightarrow \{(d_i, d_j) \mid d_i, d_j \in S; d_i \neq d_j\}$. The function *e* defines which nodes are connected by which edge. The edges *u* and *v* are called parallel or multiple edges if $e(u) = e(v)$.

Each parallel edge represents a relationship between the nodes (i.e., documents) it connects. There are several types of edges; each represents different relationships between the two source code documents. Here we consider two types of relationships, namely semantic similarity and structural relationships. They are defined as follows.

Definition. The function *sem* : $S \times S \rightarrow E$ defines edges that represents a semantic similarity between source code documents. The edges defined using the *sem* function are called *semantic edges*.

Definition. The function *struct* : $S \times S \rightarrow E$ defines edges that represent a structural relationship (e.g., data flow, control flow, coupling, etc.). The edges defined using the *struct* function are called *structural edges*.

A cluster c_k is then represented as a connected (by edges of one type) sub-graph of *G* (figure 1). Each node with zero degree represents a singleton cluster. There will be as many types of clusters as there are edge types. In particular, we define *semantic clusters*, having edges defined by the semantic similarity function and *structural clusters*, having edges defined with the structural connectivity function. Multiple semantic similarity and structural connectivity functions can be incorporated into this model. Currently, we are only using one type of each for the work presented here.

The weighted semantic edges $\lambda(d_i, d_j) \in E$ represent the similarity measure between to adjacent nodes and in this case $\lambda = lsi(d_i, d_j)$, where the function *lsi* : $S \times S \rightarrow R$[-

1,1] is a real value between –1 and 1 that represents the similarity measure between its document arguments. This represents the cosine between the vector descriptions of the two documents. Of particular interest here are the edges that represent a similarity between two documents with a value $\lambda > 0.7$ (i.e., this corresponds to an angle of $45°$ or less between the vectors). Edges that do not fit this constraint can be disregarded or removed from the graph. The semantic similarity function to be used is *lsisem*: $S \times S \rightarrow E$, where $lsisem(d_i, d_j) \in E$ if $lsi(d_i, d_j) > 0.7$.

The structural information currently being used is derived from the file structure of the software system. While this is a very simple form of structural information, it has been shown to convey a good deal of information [3]. In addition, most existing software clustering methods that use a graph representation model the files as simple nodes [2-4, 33, 34]. It is often considered that files are implicitly cohesive units of a software system. Unfortunately, that is not the case in many legacy systems.

The un-weighted structural edges $(d_i, d_j) \in E$ represents the appurtenance to the same file. The function *file*: $S \times S \rightarrow \{1, 0\}$ is 1 if the argument documents are from the same source code file and 0 otherwise. The structural connectivity function to be used is *filestruct*: $S \times S \rightarrow E$, where $filestruct(d_i, d_j) \in E$ if $filestruct(d_i, d_j) = 1$. Thus, in this case, a structural cluster will represent a file.

Given these definitions, we build a set of metrics for use in the comprehension task. For simplicity, we will refer to semantic clusters as simply clusters and to the structural clusters as files. The following is a set of measures and metrics that pertain to (semantic) clusters of source code documents:

Definition. The number of files that contain a document from a given cluster is $|FDC_k|$ where
$$FDC_k = \{f \subseteq S \mid c_k \cap f \neq \varnothing\}$$

Definition. The semantic cohesion of a cluster with respect to files is
$$SCCF_k = 1 - \frac{|FDC_k| - 1}{|c_k|}.$$

Definition. The number of documents in a cluster from a given file is $|DCF_{i,k}|$ (number of common nodes between the two clusters) where
$$DCF_{i,k} = \{d \mid d \in c_k \cap f_i, c_k \subseteq S, f_i \subseteq S\}.$$

Definition. The degree of relationship of a given file with a given cluster $R_{i,k}$ is $|DCF_{i,k}| / |f_i|$.

Below is a set of measures and metrics that deal directly with files (structural clusters) of the software system:

Definition. The number of clusters that contain a document from a given file is $|CDF_i|$ where
$$CDF_i = \{c_k \subseteq S \mid c_k \cap f_i \neq \varnothing\}$$

Definition. The semantic cohesion of a file with respect to clusters is
$$SCFC_i = 1 - \frac{|CDF_i| - 1}{|f_i|}.$$

Definition. Number of files related by a cluster to a given file, f_i, is $|RF_i|$ where
$$RF_i = \{f \subseteq S \mid c_k \cap f \cap f_i \neq \varnothing, f \neq f_i, c_k \subseteq S\}.$$

Definition. Number of files strongly related by a cluster to a given file, f_i, is SRF_i: $SRF_i = |RF_i| - \max |c_k| - 1$ and $c_k \in LC_k$ where LC_k is the set of clusters that contain documents from f_i and have a low semantic cohesion with respect to files.
$$LC_k = FDC \cap \{c_j \subseteq S \mid 1 - \frac{|FDC_j| - 1}{|c_j|} < \varepsilon\}$$

where ε is an empirically established threshold.

All these definitions can be generalized to relate to semantic clusters and structural clusters of any kind.

5. Experiments with Mosaic

To determine how well the selected IR method supports the program understanding process, the source code for version 2.7 of Mosaic [36] was

Table 1. Vitals for Mosaic.

LOC	95,000
Vocabulary	5,114
Number of parsed documents	2,347
Number of clusters produced	655

used as input into LSI and clustered using the previously described method. The resulting partitioning is then used to help support understanding of portions of the source code. This experiment was undertaken as a control, using only the IR method without any serious structural information. It was felt that if reasonable results were produced with the IR method alone, then combining that with more structural information would lead to a powerful tool.

Mosaic is written in C and was programmed and developed by multiple individuals. No single coding standard is observed over the entire system and often times different standards are used within a given file. Little or no external documentation on the design or architecture is available and the internal documentation is often scarce or missing. In short, Mosaic reflects the kinds of realities often found in commercial software due to the many external issues that affect a software development project.

5.1. Clustering Mosaic

Table 1 gives the size of the Mosaic system (269 files containing approximately 95 KLOC). A semantic space using a dimensionality of 350 for the 2,347 documents

Table 2. A distribution of the size of clusters. The number of clusters that contain a given number of documents.

Number of Documents	Number of Clusters
1	481
2	98
3 - 5	46
6 - 10	15
11 -30	8
38	1
99	1
1084	1

was generated. The documents were then clustered, as stated previously, based on the angle of 45 degrees or less (i.e., between 1.0 and 0.7) between any two vectors, which resulted in 655 groupings.

A distribution of the clusters based on the number of documents they contain is given in table 2. There are a large number of singleton clusters (481) and few really large clusters. These numbers reflect the same type of trends that were found in the earlier experiments. The large number of clusters of size one reflects the fact that many functions often stand by themselves semantically. The largest cluster is for the most part composed of a common header comment that is found in almost every file. It also includes a large number of very small documents that were parsed out to be only one or two lines of code.

A number of scenarios were envisioned that require such understanding of a large software system with little existing external documentation. The system may be under maintenance by a person with little knowledge of the system or a reengineering of the system may be planned. In such a case, the software is written in C, a reengineering of the system in another language, say C++, may be planned. In fact, such a reengineering of Mosaic actually took place and current versions are written in C++.

The clustering of the source code gives another dimension to view relationships among pieces of source code. Grouping functions and structures together within a file often represents some semantic relationship within the grouping. For instance, an abstract data type (ADT) is often encapsulated in the C language within an implementation file (.c) and an associated specification file (.h). Unfortunately, not all software systems are written with good habits of coupling and cohesion in mind. In legacy systems, it is quite common that little (or no) semantic encapsulation is used, concepts are spread over multiple files, and files contain multiple concepts.

5.2. Understanding Mosaic

The metrics described in section 4 were computed for the clustering of Mosaic that was generated. The resulting values are used to identify groups of documents in the software system that should be investigated as a whole. The following guidelines are utilized in assessment of clusters and files:

- Semantic cohesion of a file with respect to clusters ($SCFC_i$) should be high.
- Number of files strongly related by a cluster to a given file (SRF_i) should be low.
- Semantic cohesion of a cluster with respect to files ($SCCF_k$) should be high.
- Degree of relationship of a given file with a given cluster ($R_{i,k}$) should be high.

Each of the following examples presents a group of files and clusters that are related. They were selected either solely based on the values of their associated metrics, or in conjunction with some additional domain knowledge (e.g., file names, existence of some variables in the files, etc.). Files that satisfy the above-mentioned conditions were considered for further manual inspection. This step, selecting the files that are candidates for manual inspection, can be automated and would reduce the amount of manual work needed to understand the software system. For the files and clusters the measurements and metrics are computed and presented in groups of three tables: one for the metrics and measurements dealing with files (tables 3, 6 and 9), another for the metrics dealing with clusters (tables 4, 7, and 10), and the third for the degree of relationship between the files and clusters (tables 5 and 8).

Table 3. Metrics on the important files related to DrawingArea.c

File (f_i)	$\mid f_i \mid$	$SCFC_i$	SRF_i
DrawingArea.c	11	0.73	3
DrawingArea.h	1	1.00	2
DrawingAreaP.h	1	1.00	2
HTML.c	91	0.69	14

5.2.1. Example: DrawingArea

The first example shows a group of related files (table 3) that were selected based on the file names, a natural choice that an analyst would do when starting to understand a software system. The goal of the experiment was to see if using the measurements and values of the metrics, one could identify related files. DrawingArea.c was the first selected file, and the existing measurements indicated that it is strongly related with three other files (see SRF_i in table 3). The degree of relationship with cluster c_1 is very low (see table 5), so the related files through that cluster were not considered for further analysis. The measurements and the metrics (table 3, 4 and 5) indicated DrawingArea.h and DrawingAreaP.h as strong candidates for analysis. Given the names of the files, this is not a surprising finding. The values also indicated HTML.c as the best candidate among the rest of the related files. HTML.c does not satisfy entirely the

Table 4. Semantic cohesion of clusters with respect to files

Cluster	$SCCF_k$
c_{327}	0.64
c_{331}	0.50
c_1	0.81

above-mentioned constraints, however, the fact that it relates to DrawingArea.c through three clusters with two having high metric values (see table 4), and it has a relatively high cohesion (table 3), promoted it as a candidate for in-depth analysis.

The in-depth analysis revealed that indeed the three strongly related files implement a minimalist drawing area widget. Additionally, the files implement a well-defined abstract data type – drawing area. A form of information hiding was even used by using a separate file namely, DrawingAreaP.h, to implement some "private" functions. Two of the functions in DrawingArea.c connect the files with over 200 other files, through cluster c_1. Closer inspection revealed that the two functions are in fact constructors and have only two lines of code. This makes them similar with many other constructor-type functions, so the induced relationships were ignored. The values of the metrics in table 1 and table 3 signify this fact to some degree. The analysis confirmed that this was a coincidental relationship.

Although the metrics indicated a weak relationship with the HTML.c file, the relating functions were analyzed. The two functions (from DrawingArea.c and HTML.c) in cluster c_{331} are related because they are definitions to similar structures: one defines htmlClassRec, while the other defines drawingAreaClassRec. Both definitions use the same constant names and similar identifiers (e.g., TRUE, FALSE, NULL, Initialize, Inherit, Resize, etc.).

Table 5. Degree of relationship of a given file with a given cluster

File(f_i)	Cluster	$R_{i,k}$
DrawingArea.c	c_1	0.18
DrawingArea.c	c_{327}	0.73
DrawingArea.c	c_{331}	0.09
DrawingArea.h	c_{327}	1.00
DrawingAreaP.h	c_{327}	1.00
HTML.c	c_1	0.01
HTML.c	c_{327}	0.01
HTML.c	c_{331}	0.01

These semantic similarities indicated that both functions use the same global constructs (i.e., user defined types and identifiers) and the ADTs that they relate to could be in fact specializations of the same parent (or abstract) class. The other related function from the HTML.c file, is a geometry manager for a widget that is also used by the drawing area ADT.

These findings indicated that HTML.c should be also analyzed in conjunction with DrawingArea.c and its

Table 6. Metrics on the important files related to HTChunk.c

File(f_i)	$|f_i|$	$SCFC_i$	SRF_i
HTChunk.c	8	0.88	3
HTChunk.h	1	1.00	3
HTAAFile.c	5	0.60	16
HTNews.c	55	0.49	17

closely related files. Using the same metrics and considering HTML.c as the starting file, it is found that the HTMLWidget.c file is also related to the concepts derived previously. Finally, it was concluded that these five files contain definitions for a general (abstract) widget structure (ADT or class) and implementation of at least two specializations of it: drawingAreaClassRec and htmlClassRec.

This example proved that by analyzing the metrics, groups of files that contain cohesive implementation of ADTs or classes representing some concepts (drawing area) could be identified. The metrics helped identify files that contain implementation of similar structures (html record) and implementation of a general (abstract) concept (e.g., widget) that is a generalization of the previous ones.

Table 7. Semantic cohesion of clusters with respect to files

Cluster	$SCCF_k$
c_{472}	0.67
c_{466}	0.63

5.2.2. Example: chunk handling and flexible arrays

In this experiment, a file was selected at random from among those with very high semantic cohesion with respect to clusters and containing between 5 and 20 documents. The selected file was HTChunk.c, with 8 documents and a cohesion value of 0.88 in table 6. From this point on, a similar procedure with the one described in the first example is followed. The metrics indicated a highly cohesive set of files: HTChunk.h, HTChunk.c, and HTAAFile.c. Upon further analysis, it was determined that the two functions from HTAAFile.c that are related perform similar functions on different data structures (e.g., adding a character to a list of characters) and share variables and constants with the same name (e.g., ch, FILE, NULL). In addition, the size of these functions is relatively small (5-10 lines of code). Therefore, this file was not considered further in the analysis. However, the similarity shows that HTAAFile.c contains functions that implement some sort of list, even if not directly related to the chunks concept. Thus, a separate analysis of this is recommended. Similar facts were found for the HTNews.c file, although its metrics were even lower.

It was also found that the remaining two files

Table 8. Degree of relationship of a given file with a given cluster

File(f_i)	Cluster	$R_{i,k}$
HTChunk.c	c_{472}	1.00
HTChunk.h	c_{472}	1.00
HTAAFile.c	c_{472}	0.40
HTAAFile.c	c_{466}	0.60
HTNews.c	c_{472}	0.02

(HTChunk.c and HTChunk.h) implement an ADT that deals with flexible arrays or chunks. A chunk, in this system, is a structure that may be extended. These routines create and append data to chunks and automatically reallocate them as necessary. The generality of the structure determined the other (weaker) relationships with the other files. This suggested that those files implement similar structures (lists) but using other concepts (e.g. files and news articles rather than chunks).

The study of the related files and clusters indicated that cluster c_{466} (see tables 8 and 9) should be analyzed

Table 9. Metrics on the important files related to newsrc.c

File (f_i)	$|f_i|$	$SCFC_i$	SRF_i
newsrc.c	55	0.74	1
HTNews.c	31	0.49	17

separately. This supports the values in table 7 that also indicated the "interestingness" of the HTAAFile.c that strongly relates to the cluster c_{466} (table 8). This example showed that, solely using the metrics, groups of strongly related and cohesive files could be identified and that they implemented a general structure (chunks or flexible arrays). The metrics helped to identify files that contain similar structures (lists). The fact that Mosaic was written by several authors led to interesting facts such as the fact that often, different authors implemented their own list-processing module, instead of using a general one, across the system. Again, in this case the identified similarities help in finding these structures (e.g., lists). After that, it is easy to manually identify the concepts (objects) that are handled by the structures (e.g., stored in lists).

5.2.3. Example: cluster c_{466}, the password and access control

In this example, a cluster was chosen as a starting element in the analysis. The starting cluster is c_{466} and was indicated in the previous example as a candidate for separate analysis. The cluster spans over 14 highly related and cohesive files. The manual analysis revealed that 10 of these files implemented the basic functions and structures to handle passwords and access control. The other files were using these functions and structures. This time, the names of the files would not have indicated the relationships. Due to limited space, the actual metrics are not shown here.

5.2.4. Example: top clusters and newsgroup

This example deals with the analysis of a number of clusters with very high cohesion with respect to files. The second example (chunk handling) showed that the HTNews.c file should be a candidate for manual inspection, but the observed relationships were not relevant to that

Table 10. Semantic cohesion of clusters with respect to files

Cluster	$SCCF_k$
c_1	0.81
c_{200}	0.93
c_{205}	0.88
c_{206}	0.89
c_{201}	0.50

group of files. As mentioned, the relationships only indicated that there is a type of list structure implemented in this file. HTNews.c is related to the newsrc.c file that has high cohesion (table 9). More than that, newsrc.c is related to three of the top ten (table 10) most cohesive clusters (c_{200}, c_{205}, and c_{206}). Therefore, these were the exact type of clusters to be considered as starting elements in this experiment.

The analysis showed that newsrc.c implements a number of structures that deal with the concepts of "news group" and "news article". The relationship with the HTNews.c file, although not very strong, is significant because HTNews.c implements, among other things, a "Network News Transfer protocol module for the WWW library". In order to do that it uses the structures implemented in the newsrc.c file. More than that, the list structure, indicated by the relationships in the second example, is in fact a list of news articles.

6. Related work

Related research on similarity measures includes the work of Girard and Koschke [17, 19]. This work is also based on similarity metrics between software components and defines similarity metrics that combine structural and semantic information. The structural information is defined using a resource flow graph representation of the source code and semantic information uses the work of Schwanke [44]. Schwanke's work is based on Parnas's [39] information hiding principle and on Tversky's [48] research. The work proposed here differs in that the clusters are used as a parameter to the metrics. Other research that clusters software components includes work by Anquetil [3, 4], Wiggerts [49], Merlo [34], and Lakhotia [24]. Lakhotia lists a number of works on

software clustering. Most of them use formal features (i.e., structural information) and two of them ([28] and [37]) use semantic information (referred to as non-formal descriptive features). The research that has been conducted on the specific use of applying information retrieval methods to source code includes [15, 16, 18, 28-30, 32, 35].

7. Conclusions

The experiments with the PROCSSI system show that the semantic similarity of source code documents provides valuable information that can be used in the tasks of software maintenance and evolution. It also shows that concepts from the problem domain are often spread over multiple files, and files contain multiple concepts. The next steps in the development of the PROCSSI system will be to incorporate additional structural information into the model. The low cost of the methods used in this system and their flexibility allows experiments on software written in other languages. The only necessary change is in the initial parsing of the software system. All the tasks performed by PROCSSI can be fully automated and performed in relatively short time repeatedly, using different kinds of information.

The examples describe how the methods can be used to assist in the program comprehension process. The methods appear to be reasonable for automatically grouping semantically similar software components based on variable and type names along with comment text. The clusters produced by these methods represent an abstraction of the source code based on a semantic similarity, which should relate to higher-level concepts. The clusters produced are often similar to those that would be produced by a programmer with good knowledge of the particular software. Adding more structural information should allow the development of tools to assist the understanding of large-scale software systems. Development of cost effective methods that do not rely on the acquisition and representation of large amounts of knowledge is necessary to support program comprehension tools that are widely usable.

8. References

[1] Abd El, H. and Basili, V., "A Knowledge-Based Approach to the Analysis of Loops", *IEEE Transactions on Software Engineering*, vol. 22, no. 5, May 1996, pp. 339-360.

[2] Anquetil, N., "A Comparison of Graphs of Concept for Reverse Engineering", in Proceedings of IWPC'00, Limerick, Ireland, June 2000.

[3] Anquetil, N. and Lethbridge, T., "Extracting Concepts from File Names; a New File Clustering Criterion", in Proceedings of 20th International Conference on Software Engineering (ICSE'98), 1998.

[4] Anquetil, N. and Lethbridge, T., "Experiments with Clustering as a Software Remodularization Method", in Proceedings of 6th Working Conference on Reverse Engineering, 1999.

[5] Berry, M. W., "Large Scale Singular Value Computations", *International Journal of Supercomputer Applications*, vol. 6, 1992, pp. 13-49.

[6] Berry, M. W., Dumais, S. T., and O'Brien, G. W., "Using Linear Algebra for Intelligent Information Retrieval", *SIAM: Review*, vol. 37, no. 4, 1995, pp. 573-595.

[7] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "Program Understanding and the Concept Assignment Problem", *CACM*, vol. 37, no. 5, May 1994, pp. 72-82.

[8] Canfora, G. and al., e., "Experiments in Identifying Reusable Abstract Data Types in Program Code", in Proceedings of IEEE 2nd Workshop on Program Comprehension, 1993, pp. 36-45.

[9] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[10] Duda, R. O. and Hart, P. E., *Pattern Classification and Scene Analysis*, Wiley, 1973.

[11] Dumais, S. T., "Latent Semantic Indexing (LSI) and TREC-2", in Proceedings of The Second Text Retrieval Conference (TREC-2), March 1994, pp. 105-115.

[12] Etzkorn, L. H., Bowen, L. L., and Davis, C. G., "An Approach to Program Understanding by Natural Language Understanding", *Natural Language Engineering*, vol. 5, no. 1, 1999, pp. 1-18.

[13] Etzkorn, L. H. and Davis, C. G., "Automatically Identifying Reusable OO Legacy Code", *IEEE Computer*, vol. 30, no. 10, October 1997, pp. 66-72.

[14] Faloutsos, C. and Oard, D. W., "A Survey of Information Retrieval and Filtering Methods", University of Maryland, Technical Report CS-TR-3514, August 1995.

[15] Fischer, B., "Specification-Based Browsing of Software Component Libraries", in Proceedings of 13th ASE, 1998, pp. 74-83.

[16] Frakes, W., "Software Reuse Through Information Retrieval", in Proceedings of 20th Annual HICSS, Kona, HI, Jan. 1987, pp. 530-535.

[17] Girard, J. F. and Koschke, R., "A Comparison of Abstract Data Type and Objects Recovery Techniques", *Journal Science of Computer Programming, Elsevier* 1999.

[18] Girard, J. F., Koschke, R., and Schied, G., "Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding", in Proceedings of Working Conference on Reverse Engineering, 1997, pp. 66-75.

[19] Girard, J. F., Koschke, R., and Schied, G., "A Metric-Based Approach to Detect Abstract Data Types and State Encapsluation", *Journal Automated Software Engineering*, vol. 6, no. 4, October 1999.

[20] Harandi, M. and Ning, J., "Knowledge-Based Program Anaylsis", *IEEE Software*, vol. 7, no. 1, January 1990, pp. 74-81.

[21] Hutchens, D. and Basili, V., "System Structure Analysis: Clustering With Data Bindings", *IEEE Transactions on Software Engineering*, vol. 11, no. 8, 1985, pp. 749-757.

[22] Jolliffe, I. T., *Principal Component Analysis*, Springer Verlag, 1986.

[23] Kruskal, J. B., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. Amer. Math. Soc.*, vol. 7, no. 1, 1956, pp. 48-50.

[24] Lakhotia, A., "A Unified Framework for Expressing Software Subsystem Classification techniques", *Journal of Systems and Software*, vol. 36, March 1997, pp. 211-231.

[25] Landauer, T. K. and Dumais, S. T., "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge", *Psychological Review*, vol. 104, no. 2, 1997, pp. 211-240.

[26] LEDA, "The LEDA Manual Version R-3.7", LEDA Research, Webpage, Date Accessed: 4/29/1999, http://www.mpi-sb.mpg.de/LEDA/index.html, 1998.

[27] Livadas, P. E. and Alden, S. D., "A Toolset for Program Understanding", in Proceedings of IEEE 2nd Workshop on Program Comprehension, 1993, pp. 110-118.

[28] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering*, vol. 17, no. 8, 1991, pp. 800-813.

[29] Maarek, Y. S. and Smadja, F. A., "Full Text Indexing Based on Lexical Relations, an Application: Software Libraries", in Proceedings of SIGIR89, Cambridge, MA, June 1989, pp. 198-206.

[30] Maletic, J. I. and Marcus, A., "Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding", in Proceedings of 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Vancouver, British Columbia, November 13-15 2000, pp. 46-53.

[31] Maletic, J. I. and Reynolds, R. G., "A Tool to Support Knowledge Based Software Maintenance: The Software Service Bay", in Proceedings of The 6th IEEE International Conference on Tools with Artificial Intelligence, New Orleans LA, Nov. 6-9 1994, pp. 11-17.

[32] Maletic, J. I. and Valluri, N., "Automatic Software Clustering via Latent Semantic Analysis", in Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99), Cocoa Beach Florida, October 1999, pp. 251-254.

[33] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., and Gansner, E. R., "Using Automatic Clustering to Produce High-Level Organization of Source Code", in Proceedings of 6th International Workshop on Program Comprehension (IWPC'98), Italy, June 1998.

[34] Merlo, E., McAdam, I., and De Mori, R., "Source code informal information analysis using connectionist models", in Proceedings of Int'l Joint Conference on Artificial Intelligence (IJCAI'93), 1993, pp. 1339-1344.

[35] Michail, A. and Notkin, D., "Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships", in Proceedings of International Conference on Software Engineering, 1999.

[36] Mosaic, "Mosaic Source Code v2.7b5", NCSA, ftp site, Date Accessed: 4/12/2000, ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/source/, 1996.

[37] Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S., "A Reverse Engineering Approach to Subsystem Structure Identification", *Software Maintenance: Research and Practice*, vol. 5, no. 4, 1993, pp. 181-204.

[38] Ning, J. Q., Engberts, A., and Kozaczynski, W., "Recovering Reusable Components from Legacy Systems", in Proceedings of Working Conference on Reverse Engineering, 1993.

[39] Parnas, D. L., "Information Distribution Aspects of Design Methodology", in *Information Processing 71*, North-Holland, 1972.

[40] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, 1996.

[41] Rich, C. and Waters, R. C., "The Programmer's Apprentice: A Research Overview", *IEEE Computer*, vol. 21, no. 11, November 1988, pp. 12-25.

[42] Rist, R., "Plans in Program Design and Understanding", in Proceedings of Workshop Notes for AI & Automated Program Understanding, AAAI-92, San Jose CA 1992, pp. 98-102.

[43] Salton, G., *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*, Addison-Wesley, 1989.

[44] Schwanke, R. W., "An intelligent tool for re-engineering software modularity", in Proceedings of 13th International Conference on Software Engineering, 1991, pp. 83-92.

[45] Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge", *IEEE Transactions on Software Engineering*, vol. 10, no. 5, September 1984, pp. 595-609.

[46] Strang, G., *Linear Algebra and its Applications*, 2nd ed., Academic Press, 1980.

[47] Tanenbaum, A. and Woodhull, A., *Operating Systems Design and Implementation*, Prentice Hall, 1997.

[48] Tversky, A., "Features of similarity", *Psychological Review*, vol. 84, no. 4, July 1977.

[49] Wiggerts, T., "Using clustering algorithms in legacy systems remodularization", in Proceedings of Working Conference on Reverse Engineering, 1997, pp. 33-43.