Information-Theoretic Software Clustering

Periklis Andritsos, *Member*, *IEEE Computer Society*, and
Vassilios Tzerpos, *Member*, *IEEE Computer Society*

Abstract—The majority of the algorithms in the software clustering literature utilize structural information to decompose large software systems. Approaches using other attributes, such as file names or ownership information, have also demonstrated merit. At the same time, existing algorithms commonly deem all attributes of the software artifacts being clustered as equally important, a rather simplistic assumption. Moreover, no method that can assess the usefulness of a particular attribute for clustering purposes has been presented in the literature. In this paper, we present an approach that applies information theoretic techniques in the context of software clustering. Our approach allows for weighting schemes that reflect the importance of various attributes to be applied. We introduce LIMBO, a scalable hierarchical clustering algorithm based on the minimization of information loss when clustering a software system. We also present a method that can assess the usefulness of any nonstructural attribute in a software clustering context. We applied LIMBO to three large software systems in a number of experiments. The results indicate that this approach produces clusterings that come close to decompositions prepared by system experts. Experimental results were also used to validate our usefulness assessment method. Finally, we experimented with well-established weighting schemes from information retrieval, web search, and data clustering. We report results as to which weighting schemes show merit in the decomposition of software systems.

Index Terms—Reverse engineering, reengineering, architecture reconstruction, clustering, information theory.

1 Introduction

A common phenomenon in large software projects is that their size and complexity increases with time, while the system's structure deteriorates from continuous maintenance activities. As a result, the task of understanding such large software systems can be quite hard. The problem is often exacerbated in the case of legacy systems by the absence of the original developers and the lack of up-to-date documentation.

An approach that can be of significant help to the process of understanding, redocumenting, or reverse engineering a large software system is to create a meaningful decomposition of its structure into smaller, more manageable subsystems. Many researchers have attempted to create such decompositions automatically, giving rise to the research area of software clustering.

The majority of the software clustering approaches presented in the literature attempt to discover clusters by analyzing the dependencies between software artifacts, such as functions or source files [22], [18], [21], [29], [28], [13], [25], [17], [35]. Software engineering principles, such as information hiding or high-cohesion and low-coupling are commonly employed to help determine the boundaries between clusters.

Other approaches have also demonstrated merit. Using naming information, such as file names or words extracted from comments in the source code [5], [3], [23], has been shown to be a good way to cluster a given system. The ownership architecture of a software system, i.e., the mapping that shows which developer is responsible for what part of the system, can also provide valuable hints [9]. Some researchers have also attempted to combine structural information (based on dependencies) with nonstructural information (based on naming) in their techniques [4]. Others have proposed ways of bringing clustering into a more general data management framework [1]. Finally, concept analysis [20], [30] and pattern matching [8], [27] have also been successfully used to cluster large software systems.

Even though the aforementioned approaches have shown that they can be quite effective, there are still several issues that can be identified:

1. There is no guarantee that the developers of a legacy software system have followed software engineering principles such as high-cohesion and low-coupling. As a result, the validity of the clusters discovered following such principles, as well as the overall contribution of the obtained decomposition to the reverse engineering process, can be challenged.
2. Software clustering approaches based on high-cohesion and low-coupling often fail to discover utility subsystems, i.e., collections of utilities that do not necessarily depend on each other, but are used in many parts of the software system (they may or may not be omnipresent nodes [25]). Such subsystems do not exhibit high-cohesion and low-coupling, but they are frequently found in manually created decompositions of large software systems.
3. It is not clear what types of nonstructural attributes are appropriate for inclusion in a software clustering approach. Clustering based on the lines of code of

- *P. Andritsos is with the Department of Computer Science, University of Toronto, 40 St. George St., Toronto, Ontario, Canada, M5S 2E4. E-mail: periklis@cs.toronto.edu.*
- *V. Tzerpos is with the Department of Computer Science and Engineering, York University, 4700 Keele St., Toronto, Ontario, Canada M3J 1P3. E-mail: bil@cs.yorku.ca.*

Manuscript received 16 Apr. 2004; revised 15 Dec. 2004; accepted 28 Dec. 2004; published online 20 Feb. 2005.
Recommended for acceptance by E. Stroulia and A. van Deursen.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0072-0404.

0098-5589/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

each source file is probably inappropriate, but what about using timestamps? Ownership information has been manually shown to be valuable [10], but its effect in an automatic approach has not been evaluated. The study in [6] was inconclusive as to the usefulness of comments and identifier names.

4. Software clustering algorithms commonly treat all attributes of the software artifacts equally. However, a domain expert clustering the same system would invariably assign different importance to particular attributes based on her intuition. As a result, she might consider certain attributes as more important than others for the determination of the clusters.

In this paper, we present an approach that addresses these issues. Our approach is based on minimizing information loss during the software clustering process.

The objective of software clustering is to reduce the complexity of a large software system by replacing a set of artifacts with a cluster, a representative abstraction of all artifacts grouped within it. Thus, the obtained decomposition is easier to understand. However, this process also reduces the amount of information conveyed by the clustered representation of the software system. Our approach attempts to create decompositions that convey as much information as possible by choosing clusters that represent their contents as accurately as possible. In other words, one can predict with high probability the features of a given object just by knowing the cluster that it belongs to.

In terms of the first issue raised above, our algorithm makes no assumptions about software engineering principles followed by the developers of the software system. It also creates decompositions that convey as much information about the software system as possible, a property that should be helpful to the reverse engineer. Furthermore, as will be shown in Section 2, our approach can discover utility subsystems as well as ones based on high-cohesion and low-coupling. Addressing the third issue raised above, any nonstructural attribute may be included in our approach. As a result, our approach can be used in order to evaluate the usefulness of attributes, such as timestamps or ownership. Finally, our approach can accommodate any weighting scheme that assigns importance to the various attributes considered during clustering.

The contributions of this paper can be summarized as:

- The introduction of a novel software clustering approach based on the minimization of information loss during the clustering process. We formulate the software clustering problem within the Information Bottleneck framework and introduce a scalable algorithm called LIMBO that clusters large data sets effectively and efficiently.
- The development of a method that can evaluate the usefulness of nonstructural attributes for clustering purposes. The method was validated by evaluating four different nonstructural attributes whose usefulness, or lack thereof, was already established.
- The first study on the applicability of weighting schemes to the software clustering process. Since our approach can easily incorporate weighting schemes, we conducted experiments with several well-established weighting schemes from information retrieval, Web search, and data clustering. We report results as to which weighting schemes show merit in the decomposition of software systems.

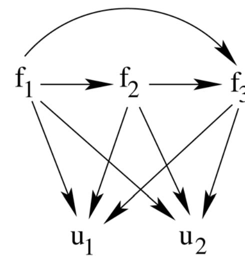

Fig. 1. Example dependency graph.

The structure of the rest of this paper is as follows: Section 2 presents some background from Information Theory, as well as the way our approach quantifies information loss for software systems. Section 3 presents LIMBO, a scalable hierarchical clustering algorithm that improves on the *Agglomerative Information Bottleneck* method [31]. In Section 4, we compare LIMBO to several other software clustering algorithms that have been presented in the literature. In Section 5, we present a method that assesses the usefulness of any nonstructural attribute to the software clustering process. Section 6 introduces five different weighting schemes and presents experimental results of applying them to three large software systems. Finally, Section 7 concludes the paper and presents directions for future work.

2 Background

This section introduces the main concepts from Information Theory that will be used throughout the paper. We present the *Agglomerative Information Bottleneck* algorithm and show examples of how it can be used with both structural and nonstructural attributes. We also motivate the use of weighting schemes by introducing an example in which the presence of weights over the different attributes produces different results with respect to the unweighted version of the data.

In this paper, we use the term *attribute* to refer to different types of information that may characterize a software artifact, such as its dependencies to other artifacts, the name of its developer, or its timestamp. The different values that each attribute takes are called *features*. For instance, the set of features of a given software artifact may be $\{f_1, f_2$, Alice, Jan-2005$\}$, if it depends on artifacts f_1 and f_2, is developed by Alice, and was last modified in January 2005.

2.1 Basics from Information Theory

In the following paragraphs, we give some basic definitions of Information Theory and their intuition. These definitions can also be found in any information theory textbook, e.g., [14].

Throughout this section we will assume the dependency graph of an imaginary software system given in Fig. 1. This graph contains three program files f_1, f_2, and f_3, and two

TABLE 1
Example Matrix M Representing the Dependencies in Fig. 1

	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$
f_1	0	1	1	1	1
f_2	1	0	1	1	1
f_3	1	1	0	1	1
u_1	1	1	1	0	0
u_2	1	1	1	0	0

utility files u_1 and u_2. This software system is clearly too trivial to require clustering. However, it will serve as an example of how our approach discovers various types of subsystems.

Our approach starts by translating the dependencies shown in Fig. 1 into the matrix M shown in Table 1. The rows of this matrix represent the artifacts to be clustered while the columns represent the values of the attributes that describe these artifacts. Since our example contains only structural information (nonstructural attributes will be added in Section 2.4), the features of a software artifact are other artifacts. To avoid confusion, we will represent the software artifacts to be clustered with italic letters, e.g., f_1, u_1, and the corresponding features with bold letters, e.g., $\mathbf{f}_1, \mathbf{u}_1$.

In matrix M, we indicate the presence of features with 1 and their absence with 0. Note that, for a given artifact a, feature $\mathbf{f}$ is present if a depends on f, or f depends on a. This was a choice made to simplify this example. One may decide that feature $\mathbf{f}$ is present only if a depends on f without affecting the rest of this discussion.

Let A denote a discrete random variable taking its values from a set of artifacts $\mathbb{A}$. In our example, $\mathbb{A}$ is the set $\{f_1, f_2, f_3, u_1, u_2\}$. If $p(a_i)$ is the probability mass function of the values a_i that A takes ($a_i \in \mathbb{A}$), the *entropy* $H(A)$ of variable A is defined by

$$H(A) = -\sum_{a_i \in \mathbb{A}} p(a_i) \log p(a_i).$$

Intuitively, entropy is a measure of disorder; the higher the entropy, the lower the certainty with which we can predict the value of A. We usually consider the logarithm with base two. Thus, entropy becomes the minimum number of bits required to describe variable A [14].

Now, let B be a second random variable taking values from the set $\mathbb{B}$ of all the features in the software system. In our example, $\mathbb{B}$ is the set $\{\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3, \mathbf{u}_1, \mathbf{u}_2\}$. Then, $p(b_j|a_i)$ is the conditional probability of a value b_j of B given a value a_i of A. The *conditional entropy* $H(B|A)$ is defined as

$$\begin{aligned} H(B|A) &= \sum_{a_i \in \mathbb{A}} p(a_i) H(B|A = a_i) \\ &= -\sum_{a_i \in \mathbb{A}} p(a_i) \sum_{b_j \in \mathbb{B}} p(b_j|a_i) \log p(b_j|a_i). \end{aligned}$$

$H(B|A)$ gives the uncertainty with which we can predict the value of B given that a value of A appears.

TABLE 2
Normalized Matrix of System Features

$A \backslash B$	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$
f_1	0	1/4	1/4	1/4	1/4
f_2	1/4	0	1/4	1/4	1/4
f_3	1/4	1/4	0	1/4	1/4
u_1	1/3	1/3	1/3	0	0
u_2	1/3	1/3	1/3	0	0

Each raw stores the distribution $p(B|A = a_i)$

An important question that arises is: "to what extent can the value of one variable be predicted from knowledge of the value of the other variable?" This question has a quantitative answer through the notion of *mutual information*, $I(A; B)$, which measures the amount of information that the variables hold about each other. The mutual information between two variables is the amount of uncertainty (entropy) in one variable that is removed by knowledge of the value of the other one. More precisely, we have

$$I(A; B) = H(A) - H(A|B) = H(B) - H(B|A) = I(B; A).$$

Mutual information is symmetric, nonnegative, and equals zero if and only if A and B are independent.

Let us consider a particular clustering C_k of the elements of $\mathbb{A}$. We introduce a third random variable C taking values from set $\mathbb{C} = \{c_1, c_2, \ldots, c_k\}$, where $c_1, c_2, \ldots, c_k$ are the k clusters of C_k. The mutual information $I(B; C)$ quantifies the information about the values of B (the features of the software system) provided by the identity of a cluster (a given value of C). The higher this quantity is, the more informative the cluster identity is about the features of its constituents. Therefore, our goal is to choose C_k in such a way that it maximizes the value of $I(B; C)$.

The maximum value for $I(B; C)$ occurs when $|\mathbb{C}| = |\mathbb{A}|$, i.e., each cluster contains only one object. The minimum value for $I(B; C)$ occurs when $|\mathbb{C}| = 1$, i.e., when all objects are clustered together. Interesting are the cases in-between, where we seek a *k-clustering* C_k, that contains a sufficiently small number of clusters (compared to the number of objects), while retaining a high value for $I(B; C)$.

Recasting the problem of software clustering within the information theory context, we normalize matrix M in Table 1, so that the entries of each row sum up to one. Formally, for each artifact a_i, we define

$$p(a_i) = 1/n, \tag{1}$$

$$p(b_j|a_i) = \frac{M[a_i, b_j]}{\sum_{b \in \mathbf{B}} M[a_i, b]}. \tag{2}$$

The normalized matrix M for our example is depicted in Table 2.

Each row of the normalized matrix holds the *feature vector* of an artifact, which is equivalent to the conditional probability $p(B|A = a_i)$.

TABLE 3
Pairwise δI Values for Vectors of Table 2

	f_1	f_2	f_3	u_1	u_2
f_1	-	0.10	0.10	0.17	0.17
f_2	0.10	-	0.10	0.17	0.17
f_3	0.10	0.10	-	0.17	0.17
u_1	0.17	0.17	0.17	-	**0.00**
u_2	0.17	0.17	0.17	**0.00**	-

2.2 Agglomerative Information Bottleneck

Slonim and Tishby [31] proposed a solution to the optimization problem of finding a clustering of small cardinality and large information content. Their approach is called the *Agglomerative Information Bottleneck (AIB)* algorithm. It has been successfully used in document clustering [32] and the classification of galaxy spectra [26]. Similar to all agglomerative (or bottom-up) techniques, the algorithm starts with the clustering C_n, in which each object $a_i \in \mathbb{A}$ is a cluster by itself. As stated before, $I(A;B) = I(C_n;B)$. At step $n - \ell + 1$ of the AIB algorithm, two clusters c_x, c_y in ℓ-clustering C_ℓ are merged into a single component c^* to produce a new $(\ell - 1)$-clustering $C_{\ell-1}$. As the algorithm forms clusterings of smaller size, the information that the clustering contains about the features in $\mathbb{B}$ decreases; that is, $I(B;C_{\ell-1}) \leq I(B;C_\ell)$. The clusters c_x and c_y to be merged are chosen to minimize the information loss in moving from clustering C_ℓ to clustering $C_{\ell-1}$. This information loss is given by $\delta I(c_x, c_y) = I(B;C_\ell) - I(B;C_{\ell-1})$. We can also view the information loss as the increase in the uncertainty of predicting the features in the clusters before and after the merge.

After merging clusters c_x and c_y, the new component $c^* = c_x \cup c_y$ has [31]

$$p(c^*) = p(c_x) + p(c_y), \tag{3}$$

$$p(b_j|c^*) = \frac{p(c_x)}{p(c^*)} p(b_j|c_x) + \frac{p(c_y)}{p(c^*)} p(b_j|c_y). \tag{4}$$

Tishby et al. [33] show that

$$\delta I(c_x, c_y) = [p(c_x) + p(c_y)] \cdot D_{JS}[p(b_j|c_x), p(b_j|c_y)], \tag{5}$$

where D_{JS} is the *Jensen-Shannon (JS)* divergence, defined as follows: Let $p_x = p(b_j|c_x)$ and $p_y = p(b_j|c_y)$ and let

$$\bar{p} = \frac{p(c_x)}{p(c^*)} p_x + \frac{p(c_y)}{p(c^*)} p_y$$

denote the weighted average distribution of distributions p_x and p_y. Then, the D_{JS} distance is:

$$D_{JS}[p_x, p_y] = \frac{p(c_x)}{p(c^*)} D_{KL}[p_x||\bar{p}] + \frac{p(c_y)}{p(c^*)} D_{KL}[p_y||\bar{p}].$$

D_{KL} is the *Relative Entropy*, or the *Kullback-Leibler (KL)* divergence, a standard information-theoretic measure of the difference between two probability distributions. Given two distributions p and q over a set A, the relative entropy is

$$D_{KL}[p\|q] = \sum_{a_i \in \mathbb{A}} p(a_i) \log \frac{p(a_i)}{q(a_i)}.$$

Intuitively, the relative entropy $D_{KL}[p\|q]$ is a measure of the redundancy in an encoding that assumes the distribution q, when the true distribution is p.

Then, D_{JS} distance is the average D_{KL} distance of p_x and p_y from $\bar{p}$. It is nonnegative and equals zero if and only if $p_x = p_y$. It is also bounded above by one, and it is symmetric. Note that the information loss for merging clusters c_x and c_y, $\delta I(c_x, c_y)$, depends only on the clusters c_x and c_y and not on other parts of the clusterings C_ℓ and $C_{\ell-1}$.

Intuitively, at each step, AIB merges two clusters that will incur the smallest value in δI. The probability of the newly formed cluster becomes equal to the sum of probabilities of the two clusters (3) and the conditional probability of the features given the identity of the new cluster is a weighted average of the conditional probabilities in the clusters before the merge (4).

TABLE 4
Normalized Matrix after Forming c_f and c_u

$A\backslash B$	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$
c_f	1/8	1/8	1/4	1/4	1/4
f_3	1/4	1/4	0	1/4	1/4
c_u	1/3	1/3	1/3	0	0

2.3 Structural Example

By applying the equations of the previous section to our example software system, we can compute all pairwise values of information loss (δI). These values are given in Table 3. The value in position (i, j) indicates the information loss we would incur if we chose to group the ith and the jth artifact together.

Clearly, if utility files u_1 and u_2 get merged in the same cluster, c_u, we lose no information about the system, since they have exactly the same structural features. On the other hand, we lose some information if f_1 and f_2 get merged in the same cluster c_f, which is the same loss of information if any pair among the program files forms a cluster. Table 4 depicts the new matrix after forming clusters c_f and c_u. Intuitively, c_u represents the dependencies of its constituents *exactly* as good as u_1 and u_2 before the merge, while c_f is *almost* as good. We compute the probabilities of the two new clusters using (3) from Section 2.2 as $p(c_f) = 2/5$ and $p(c_u) = 2/5$, while the new distributions $p(B|c_f)$ and $p(B|c_u)$ are calculated using (4) of the same section. The obtained values are shown in Table 4.

The new matrix of pairwise distances is given in Table 5, which suggests that c_f will next be merged with f_3 as their δI value is the minimum. This indicates that our approach is able to discover both utility subsystems (such as c_u) as well as cohesive ones (such as the cluster containing f_1, f_2, and f_3).

TABLE 5
Pairwise δI after Forming c_f and c_u

	c_f	f_3	c_u
c_f	-	**0.04**	0.26
f_3	**0.04**	-	0.24
c_u	0.26	0.24	-

It is important to note that our approach commonly discovers several utility subsystems in large software systems, since it is unlikely that all utilities serve all program files as in this example. In a large software system, there are probably sets of utilities that serve particular subsystems. Our approach is able to cluster together such sets of utilities, since their elements have similar feature sets.

2.4 Nonstructural Example

One of the strengths of our approach is its ability to consider various types of information about the software system. Our example so far contained only structural attributes. We will now expand it to include nonstructural attributes as well, such as the name of the developer or the location of an artifact.

All we need to do is extend the universe $\mathbb{B}$ to include the values of nonstructural attributes. This way our algorithm is able to cluster the software system in the presence of meta-information about software artifacts.

The files of Fig. 1 together with their *developer* and *location* are given in Table 6.

The normalized matrix when $\mathbb{B}$ is extended to $\{\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3, \mathbf{u}_1, \mathbf{u}_2, \mathbf{Alice}, \mathbf{Bob}, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ is given in Table 7.

After that, $I(A;B)$ is defined and clustering can be performed as in the case of structural data, without necessarily giving the same results. More on this issue will be presented in the experimental evaluation section of this paper.

2.5 Incorporating Weighting Schemes

In the example of the previous section, we normalized each row of matrix M in order to make it a probability distribution. This way, we consider the appearance of a feature in a vector as probable as any of the other ones in the same vector. If we represent importance with numerical weights, the aforementioned conceptualization of our software system assigns equal importance to all attributes.

TABLE 6
Values of Nonstructural Attributes for the Files in Fig. 1

	Developer	Location
f_1	**Alice**	$\mathbf{p_2}$
f_2	**Bob**	$\mathbf{p_1}$
f_3	**Bob**	$\mathbf{p_2}$
u_1	**Alice**	$\mathbf{p_3}$
u_2	**Bob**	$\mathbf{p_3}$

However, in a real-life reverse engineering project, there may be valid reasons to assign different importance to different attributes. For example, the importance of ownership information (what file was developed by which developer) might be quite different for a system developed using agile methodologies as opposed to more traditional approaches, such as the waterfall model.

One of our goals in this paper is to study how particular weighting schemes over the attributes of the artifacts being clustered influence the resulting clusters. Our approach allows for any weighting scheme over the attributes to be applied. We describe how this is done through an example.

Consider the feature vectors in Table 7. By applying (5), we can compute all pairwise values of information loss (δI). These values are given in Table 8. The value in position (i, j) indicates the information loss we would incur if we chose to group the ith and the jth artifact together.

From the information losses of Table 8, we conclude that the algorithm is going to merge pair (u_1, u_2), which has the lowest value of 0.08. Eventually, the same clusters as in the structural case will be formed.

Let us now assume that a particular weighting scheme has assigned weights to the attributes in this example (larger weights correspond to more important attributes). If this weighting scheme considers the developer of each file as its most important attribute, the vector of weights w may be

$$w = (0.05, 0.05, 0.05, 0.05, 0.05, 0.30, 0.30, 0.05, 0.05, 0.05).$$

In other words, the two developer features (Alice and Bob) are considered six times more important for our clustering purposes than the rest of the features.

TABLE 7
Normalized Matrix of System Dependencies with Structural and Nonstructural Features

	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$	**Alice**	**Bob**	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$
f_1	0	1/6	1/6	1/6	1/6	1/6	0	0	1/6	0
f_2	1/6	0	1/6	1/6	1/6	0	1/6	1/6	0	0
f_3	1/6	1/6	0	1/6	1/6	0	1/6	0	1/6	0
u_1	1/5	1/5	1/5	0	0	1/5	0	0	0	1/5
u_2	1/5	1/5	1/5	0	0	0	1/5	0	0	1/5

TABLE 8
Pairwise δI Values for Vectors of Table 7

	f_1	f_2	f_3	u_1	u_2
f_1	-	0.20	0.1333	0.1813	0.2542
f_2	0.20	-	0.1333	0.2542	0.1813
f_3	0.1333	0.1333	-	0.2542	0.1813
u_1	0.1813	0.2542	0.2542	-	**0.08**
u_2	0.2542	0.1813	0.1813	**0.08**	-

TABLE 9
Data Representation with Weights

	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{u_1}$	$\mathbf{u_2}$	**Alice**	**Bob**	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$
f_1	0	0.0909	0.0909	0.0909	0.0909	0.5455	0	0	0.0909	0
f_2	0.0909	0	0.0909	0.0909	0.0909	0	0.5455	0.0909	0	0
f_3	0.0909	0.0909	0	0.0909	0.0909	0	0.5455	0	0.0909	0
u_1	0.1	0.1	0.1	0	0	0.6	0	0	0	0.1
u_2	0.1	0.1	0.1	0	0	0	0.6	0	0	0.1

TABLE 10
Pairwise δI Values for Vectors of Table 9

	f_1	f_2	f_3	u_1	u_2
f_1	-	0.2909	0.2545	*0.0950*	0.3238
f_2	0.2909	-	**0.0727**	0.3238	*0.0950*
f_3	0.2545	**0.0727**	-	0.3238	*0.0950*
u_1	*0.0950*	0.3238	0.3238	-	0.24
u_2	0.3238	*0.0950*	*0.0950*	0.24	-

In order to have the importance implied by this weighting scheme reflected in matrix M, we replace each appearance of a feature in an artifact with its weight and normalize the rows of matrix M so that they sum up to one. Using the example vector w given above the new matrix M is given in Table 9.

The new pairwise distances are given in Table 10.

In the presence of importance for the attributes, the closest artifacts are now f_2 and f_3 since, in addition to having similar structural features, they were also both developed by Bob. As a result, the first clustering step will merge f_2 and f_3 into a cluster $c_{f_{23}}$.

More importantly, as evidenced by the second highest values in Table 10 (presented in italics), the subsequent steps will merge f_1 and u_1, as well as $c_{f_{23}}$ and u_2. Thus, the two clusters that will eventually be formed will be $\{f_1, u_1\}$ and $\{f_2, f_3, u_2\}$. This indicates clearly that assigning importance to particular attributes can significantly impact the clustering process.

After this illustrative example, we are now ready to formally define the data representation in the presence of weights for the attributes. For each vector a_i we define:

$$p(a_i) = 1/n, \tag{6}$$

$$p(b_j|a_i) = \frac{w(b_j) \cdot M[a_i, b_j]}{\sum_{b \in \mathbb{B}} w(b) \cdot M[a_i, b]}. \tag{7}$$

It is clear that (2), presented earlier, is just a special case of (7), where all features have equal weight.

Finally, it is interesting to note that our approach provides maximum flexibility to the weighting schemes, since it allows for different weights to be assigned to particular values of a given attribute. For instance, one might decide that the fact that Alice developed a particular artifact is the most important factor for our clustering purposes. In this case, the weight vector might be

$$w = (0.05, 0.05, 0.05, 0.05, 0.05, 0.55, 0.05, 0.05, 0.05, 0.05).$$

3 Clustering Using *LIMBO*

Given a large number of artifacts n, the Agglomerative Information Bottleneck algorithm suffers from high computational complexity, Namely, $\mathcal{O}(n^2 \log n)$, which is prohibitive for large data sets. In this section, we introduce the *scaLable InforMation BOttleneck (LIMBO)* algorithm that

improves on the AIB algorithm and is capable of handling larger inputs.[1]

Our aim is to use the AIB algorithm but on a smaller set of artifacts. In order to reduce the number of artifacts, we perform an initial phase, where artifacts are summarized in a newly created set $\mathbb{S}$ of summaries that we call *Summary Artifacts, (SA)*. A summary artifact is denoted by $SA(S)$, where S is the set of artifacts it summarizes. Its probability distribution is derived from the probability distribution of the elements of S as described in the following section. The only difference between a summary artifact and an "original" artifact is that the summary artifacts did not exist prior to the application of our approach.

Intuitively, we are trying to merge the original artifacts into summaries that have the same characteristics as the original artifacts, i.e., they store the distributions of features and can be used in the expressions of the AIB method.

3.1 The LIMBO Clustering Algorithm

We now present the LIMBO algorithm. In what follows, n is the number of original artifacts, and q is the number of features all artifacts are expressed over.

The LIMBO algorithm proceeds in four phases. In the first phase, we summarize the original artifacts into a set $\mathbb{S}$ of SAs. In the second phase, the *Agglomerative Information Bottleneck* algorithm is employed on $\mathbb{S}$ in order to produce a series of clusterings of SAs with decreasing cardinality. The third phase transforms these clusterings into decompositions of the original artifact set. Finally, phase 4 selects one of these decompositions as the final result.

Phase 1: Creation of the Summary Artifacts. In this phase, original artifacts are read one by one. The first artifact a_1 is converted into the summary artifact $SA(\{a_1\})$, whose probability vector is equal to that of a_1. For each subsequent artifact a_i, we compute its distance to each existing SA.

The distance between an original artifact a_i and a summary artifact $SA(S_j)$ is the information loss that we would incur if we added a_i into S_j. By applying (5) from Section 2.2, we can compute this information loss as

$$\delta I\big(a_i, SA(S_j)\big) = \left(\frac{1}{|S_j|+1} + \frac{|S_j|}{|S_j|+1}\right) D_{JS}[p(B|a_i), p(B|SA(S_j))].$$

Next, we identify the summary artifact $SA(S_{min})$ with the smallest distance to a_i. If this distance is smaller than a predefined threshold (described in Section 3.2), then $SA(S_{min})$ is replaced by a new summary artifact $SA(S_{min} \cup \{a_i\})$. The probability distribution for the new summary artifact is given by:

$$p(B|SA(S_{min} \cup \{a_i\})) = \frac{1}{|S_{min}|+1} p(B|a_i) + \frac{|S_{min}|}{|S_{min}|+1} p(B|SA(S_{min})).$$

If the distance $\delta I\Big(a_i, SA(S_{min})\Big)$ is larger than the predefined threshold, a_i is converted into a new summary artifact $SA(\{a_i\})$, whose probability vector is equal to that of a_i.

In order to reduce the amount of time it takes to find $SA(S_{min})$, we organize the SAs into a B-tree-like data structure with a branching factor of E (default value is 4). This results in the computational complexity of this phase being $\mathcal{O}(qEn \log_E n)$ [2]. The I/O cost is $\mathcal{O}(n)$ since only one scan of the data is required.

Phase 2: Application of the AIB algorithm. Phase 1 replaces the set of original artifacts with a much smaller set of SAs. In the second phase, our algorithm employs the AIB algorithm to cluster the set of SAs. The input to the AIB algorithm is the set of conditional probability distributions of all SAs created in the first phase. This phase creates many clusterings of the summary artifacts, one for every value between 2 and $|\mathbb{S}|$. Note that the AIB algorithm is applied only once, with intermediate clusterings recorded at each step.

The time for this phase depends upon the number of SAs, which can be much smaller than n (depending on the threshold used in Phase 1). The computational complexity of this phase is $\mathcal{O}(|\mathbb{S}|^2 log|\mathbb{S}|)$. There is no I/O cost involved in this phase since all computations are done in main memory.

Phase 3: Associating original artifacts with clusters. Phase 2 produces $|\mathbb{S}| - 1$ clusterings, each one containing a number of probability distributions representing its clusters. In the third phase, we perform a scan over the set of original artifacts and assign each one of them to the cluster to which it is closest, with respect to the D_{KL} distance, in all clusterings.

The I/O cost of this phase is the reading of the data set and the clusterings from the disk. The CPU complexity is $\mathcal{O}(|\mathbb{S}|^2 qn)$, since each artifact is compared against all clusters in all clusterings, a total of $\frac{|\mathbb{S}|(|\mathbb{S}|+1)}{2} - 1$ clusters.

Phase 4: Determining the number of clusters. In order to choose an appropriate number of clusters k, we examine the decompositions created in Phase 3 for ascending values of k starting at 2. Let C_k be a clustering of k clusters and C_{k+1} a clustering of $k+1$ clusters. If the clusters in C_k reflect inherent groupings in the data, then C_{k+1} must contain the same clusters, except that one of them has been split into two (since the number of clusters must increase). Using MoJo [36], we can detect this situation by computing the distance between C_{k+1} and C_k, i.e., the value of $MoJo(C_{k+1}, C_k)$. If this value is equal to one, the difference between the two clusterings must be a single Join operation between two clusters of C_{k+1} to produce the k clusters of C_k. As a result, k is chosen as the smallest value, for which $MoJo(C_{k+1}, C_k) = 1$. Section 4 presents experimental data that indicate that this is indeed an effective method to choose the number of clusters.

3.2 Threshold Value

LIMBO uses a threshold value $\tau(\phi)$, which is a function of a user-specified parameter ϕ, to control the decision to merge an artifact into an existing SA or place it in an SA by itself. This threshold controls the amount of information loss in our summary of the original data set. It also affects the number of SAs created, which, in turn, determines the computational cost of the AIB algorithm in Phase 2. A good

1. An implementation of LIMBO is available through http://www.cs.utoronto.ca/db/limbo.

choice for ϕ is necessary to produce a concise and useful summarization of the data set.

In LIMBO, we adopt a heuristic based on the mutual information between variables A and B in order to set the value of $\tau(\phi)$. Before running LIMBO, the value of $I(A;B)$ is calculated by doing a full scan of the data set. Since there are n vectors in A, "on average" every vector contributes $I(A;B)/n$ to the mutual information $I(A;B)$. We define the threshold $\tau(\phi)$ as follows:

$$\tau(\phi) = \phi \frac{I(A;B)}{n}, \qquad (8)$$

where $0 \leq \phi \leq n$ denotes the multiple of the "average" mutual information that we wish to preserve when merging an artifact into an existing SA. If the merge would incur information loss more than ϕ times the "average" mutual information, then the new artifact is placed in an SA by itself.

In the next section, we present experiments that compare LIMBO to existing software clustering algorithms.

4 Comparison of LIMBO to Other Clustering Algorithms

In order to evaluate the applicability of LIMBO to the software clustering problem, we applied it to three large software systems of known authoritative decomposition and compared its output to that of other well-established software clustering algorithms.

The software systems we used for our experiments came in a variety of sizes and development philosophies:

1. **TOBEY**. This is a proprietary industrial system that is under continuous development. It serves as the optimizing backend for a number of IBM compiler products. The version we worked with was comprised of 939 source files and approximately 250,000 lines of code. The authoritative decomposition of TOBEY was obtained over a series of interviews with its developers.
2. **Linux**. We experimented with version 2.0.27a of the kernel of this free operating system that is probably the most famous open-source system. This version had 955 source files and approximately 750,000 lines of code. The authoritative decomposition of this version of the Linux kernel was presented in [11].
3. **Mozilla**. The third software system we used for our experiments was Mozilla, an open-source Web browser. We experimented with version 1.3 that was released in March 2003. It contains approximately four million lines of C and C++ source code.

   We built Mozilla under Linux and extracted its static dependency graph using CPPX and a dynamic dependency graph using *jprof*. A decomposition of the Mozilla source files for version M9 was presented in [16]. For the evaluation portion of our work, we used an updated decomposition for version 1.3 [37].

The software clustering approaches to which we compared LIMBO were the following:

1. **ACDC**. This is a pattern-based software clustering algorithm that attempts to recover subsystems commonly found in manually created decompositions of large software systems [35].
2. **Bunch**. This is a suite of algorithms that attempt to find a decomposition that optimizes a quality measure based on high-cohesion and low-coupling. We experimented with two versions of a hill-climbing algorithm, which we will refer to as NAHC and SAHC (for nearest and steepest-ascend hill-climbing) [22]. Bunch allows combinations of these two algorithms as well. However, we did not see improved results when a combination was used, so we will only report results for the pure versions of the two algorithms.

   We experimented with version 3.3.6 of Bunch. This version contains a facility that identifies certain artifacts as "libraries" and places them in a separate subsystem. Since this is similar to identifying utility subsystems, we conducted experiments with this version as well. We will refer to the two algorithms as NAHC-lib and SAHC-lib when this feature is turned on.
3. **Cluster Analysis Algorithms**. We also compared LIMBO to several hierarchical agglomerative cluster analysis algorithms. We used the Jaccard coefficient that has been shown to work best in a software clustering context [4]. We experimented with four different algorithms: single linkage (SL), complete linkage (CL), weighted average linkage (WA), and unweighted average linkage (UA).

In order to compare the output of the algorithms to the authoritative decomposition, we used the MoJo distance measure[2] [34], [36]. MoJo measures the distance between two decompositions of the same software system by computing the number of Move and Join operations one needs to perform in order to transform one to the other. Intuitively, the smaller the distance of a proposed decomposition to the authoritative one, the more effective the algorithm that produced it.

It is important to note that the MoJo distance measure does not include a Split operation. If the process of transforming an automatically created decomposition A to the authoritative decomposition B requires the splitting of a cluster, this has to be simulated by a series of Move operations. This penalizes the algorithm that created A for producing a decomposition whose clusters are too coarse. On the other hand, decompositions containing too fine-grained clusters are penalized by having to perform a large number of Join operations. Finally, decompositions at the right level of granularity, but incorrect placement of objects into clusters are penalized by having to perform a large number of Move operations.

For the experiments presented in this section, all algorithms were provided with the same input, the dependencies between the software artifacts to be clustered. In this case, the software artifacts were source files, while the dependencies were procedure calls and variable

2. A Java implementation of MoJo is available for download at: http://www.cs.yorku.ca/~bil/downloads.

TABLE 11
The Number of Clusters k and the MoJo Distance to the Authoritative Decomposition for the Decompositions Proposed by 10 Different Algorithms for the Three Example Software Systems

	TOBEY		Linux		Mozilla	
	k	MoJo	k	MoJo	k	MoJo
Authoritative	69	-	7	-	10	-
LIMBO	80	316	56	237	75	438
ACDC	94	320	66	342	205	439
NAHC	33	382	35	249	17	461
NAHC-lib	15	415	33	281	15	490
SAHC	15	482	15	353	21	440
SAHC-lib	39	396	18	314	17	502
SL	67	688	9	402	150	614
CL	153	361	154	304	223	479
WA	139	351	70	309	158	442
UA	131	354	38	316	83	427

references. The value of ϕ chosen for TOBEY and Linux was 0.0, while the value for Mozilla, a larger data set, was 0.2. Since Bunch is using randomization, its algorithms were run five times with each input. We report the average values rounded to the closest integer. The traditional cluster analysis algorithms were run with a variety of cut-point heights. The smallest MoJo distance obtained is reported below. This biases the results significantly in favor of the cluster analysis algorithms since in a different setting the cut-point height would have to be estimated without knowledge of the authoritative decomposition. However, as will be shown shortly, LIMBO outperforms the cluster analysis algorithms despite this bias in all cases but one.

Table 11 presents the results of our experiments. With the exception of UA and Mozilla, LIMBO created a decomposition that is closer to the authoritative one for all example systems, although the nearest-ascend hill-climbing algorithm of Bunch comes very close in the case of Linux, as is ACDC in the case of TOBEY, and ACDC, SAHC, and WA in the case of Mozilla. The difference in MoJo distance in these cases is too small to be used for the purpose of ranking these algorithms. However, the results in Table 11 clearly indicate that LIMBO performs at least as well as other existing algorithms.

We believe that the fact that LIMBO performed so well can be attributed mostly to its ability to discover utility subsystems. An inspection of the authoritative decompositions for all systems revealed that they contain such collections of utilities. Since, in our experience, that is a common occurrence, we are optimistic that similar results can be obtained for other software systems as well.

The results of these experiments indicate that the idea of using information loss minimization as a basis for software clustering has definite merit. Even though further experimentation is required in order to assess the usefulness of LIMBO to the reverse engineering process, it is clear that it can create decompositions that are close to the ones prepared by humans.

It is interesting to note that turning the library facility on for Bunch produced worse results in four out of six cases. In fact, NAHC was always negatively affected by it. Even when the use of the library facility produced improved results, the obtained decompositions were not as close to the authoritative one as those of other algorithms. This behavior can probably be attributed to the fact that all libraries are placed in the same subsystem even though they might be serving different parts of the system.

Another interesting observation is the fact that the cardinality of the decompositions produced by the various algorithms varies considerably. LIMBO produced decompositions of much larger cardinality that the authoritative one for Linux and Mozilla. Despite this fact, it still had one of the smallest MoJo distances, which implies that the clusters were well-formed (fewer Move operations were required).

In the case of TOBEY, LIMBO appears to be at the right level of granularity. However, the Bunch algorithms appeared to produce course decompositions. In order to give them a fair comparison, we modified their options so that they produce decompositions at the "detailed level" (the default behavior is to produce decompositions at the "median level"). However, this invariably resulted in decompositions of cardinality 300 or more. The MoJo distances of these decompositions were quite larger than the ones reported in Table 11.

We also tested LIMBO's efficiency with all systems. LIMBO was able to cluster TOBEY and Linux in approximately 31 seconds, while Mozilla required only 18 seconds. The similarity in the efficiency of LIMBO for the TOBEY and Linux systems does not come as a surprise, since the number of source files to be clustered was similar (939 in TOBEY and 955 in Linux) and ϕ was set to 0.0 in both cases. On the other hand, the improvement in the execution time for Mozilla can be attributed to the higher ϕ value (0.2).

Finally, we were interested to see how effective was our method of selecting the number of clusters k in Phase 4 of

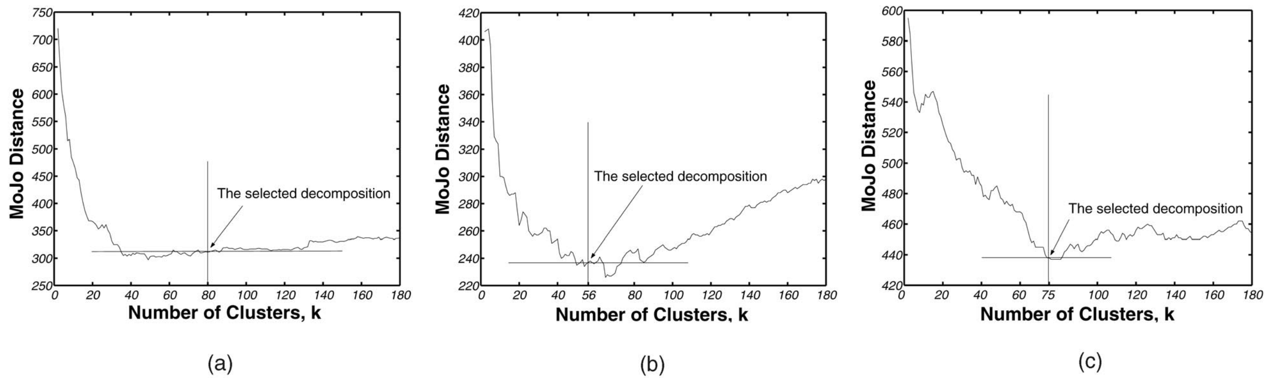

Fig. 2. The MoJo distance to the authoritative decomposition for all clusterings produced by LIMBO in Phase 3. (a) TOBEY. (b) Linux. (c) Mozilla.

LIMBO. For this reason, we computed the MoJo distance to the authoritative decomposition for all clusterings produced in Phase 3. The obtained results are shown in Fig. 2. As can be seen in these diagrams, the selected decomposition is always quite close to the optimum one. In fact, the error rate was always less than 6 percent (in the case of Mozilla it was only 0.23 percent). This makes us confident that, in a setting where the authoritative decomposition is unknown, LIMBO will select a meaningful decomposition.

5 A USEFULNESS ASSESSMENT METHOD

Several software clustering approaches have attempted to utilize nonstructural information as a clustering criterion [5], [6]. However, no method that can evaluate the usefulness of any nonstructural attribute has been presented in the literature. In this section, we utilize LIMBO's ability to combine structural and nonstructural attributes seamlessly in order to develop a method that assesses the usefulness of nonstructural attributes to the reverse engineering process.

Let us assume that, for a given software system, we have a number of nonstructural attributes that we would like to evaluate. Our method creates inputs for LIMBO that contain all possible combinations of these attributes, together with the structural information. By running LIMBO on all these inputs and comparing the outputs to the authoritative decomposition, we can establish whether the inclusion of a particular nonstructural attribute to any combination produces improved decompositions or not. Consistent results across different software systems should be a good indication of the usefulness, or lack thereof, of the attribute in question.

In the following, we provide evidence for the validity of this method, by applying it to four attributes of established usefulness. The types of nonstructural attributes we considered are:

- *Developers (dev)*: This attribute gives the ownership information, i.e., the names of the developers involved in the implementation of the file. In case no developer was known, we used a unique dummy value for each file.
- *Directory Path (dir)*: In this attribute, we include the full directory path for each file. In order to increase the similarity of files residing in similar directory paths, we include the set of all subpaths for each path. For example, the set of features for this attribute for file `drivers/char/ftape/io.c` is the set {`drivers`, `drivers/char`, `drivers/char/ftape`} of directory paths.
- *Lines of Code (loc)*: This attribute includes the number of lines of code for each of the files. We discretized the values using two different schemes:

  1. The first scheme divides the full range of *loc* values into the intervals $(0, 100]$, $(100, 200]$, $(200, 300]$, etc. Each file is given a feature such as `RANGE1`, `RANGE2`, `RANGE3`, etc.
  2. The second scheme divides the full range of *loc* values into a number of intervals so that each interval contains the same number of values. Files are given features in a similar manner to the previous scheme.

  In our experiments, both schemes gave similar results. For this reason, we will only present results for the first scheme.
- *Time of Last Update (time)*: This attribute is derived from the time-stamp of each file on the disk. We include only the month and year.

These attributes were chosen because we had clear expectations about their usefulness. Directory structure is a very common way to organize a software system, so we expected *dir* to produce meaningful clusterings. Decomposition based on ownership (*dev*) has also been shown to have merit [10]. On the other hand, our hypothesis was that *loc* and *time* would produce decompositions further away from the authoritative one. Experimental confirmation of these hypotheses would indicate the validity of the usefulness assessment method presented in this section.

Our experiments were performed on Linux and Mozilla. We considered all possible combinations of the aforementioned nonstructural attributes added to the structural information. These combinations are depicted in the lattice of Fig. 3. At the bottom of this lattice, we have the structural dependencies and, as we follow a path upwards, different nonstructural attributes are added. Thus, in the first level of the lattice, we only add individual nonstructural attributes. Each addition is represented by a different type of arrow at

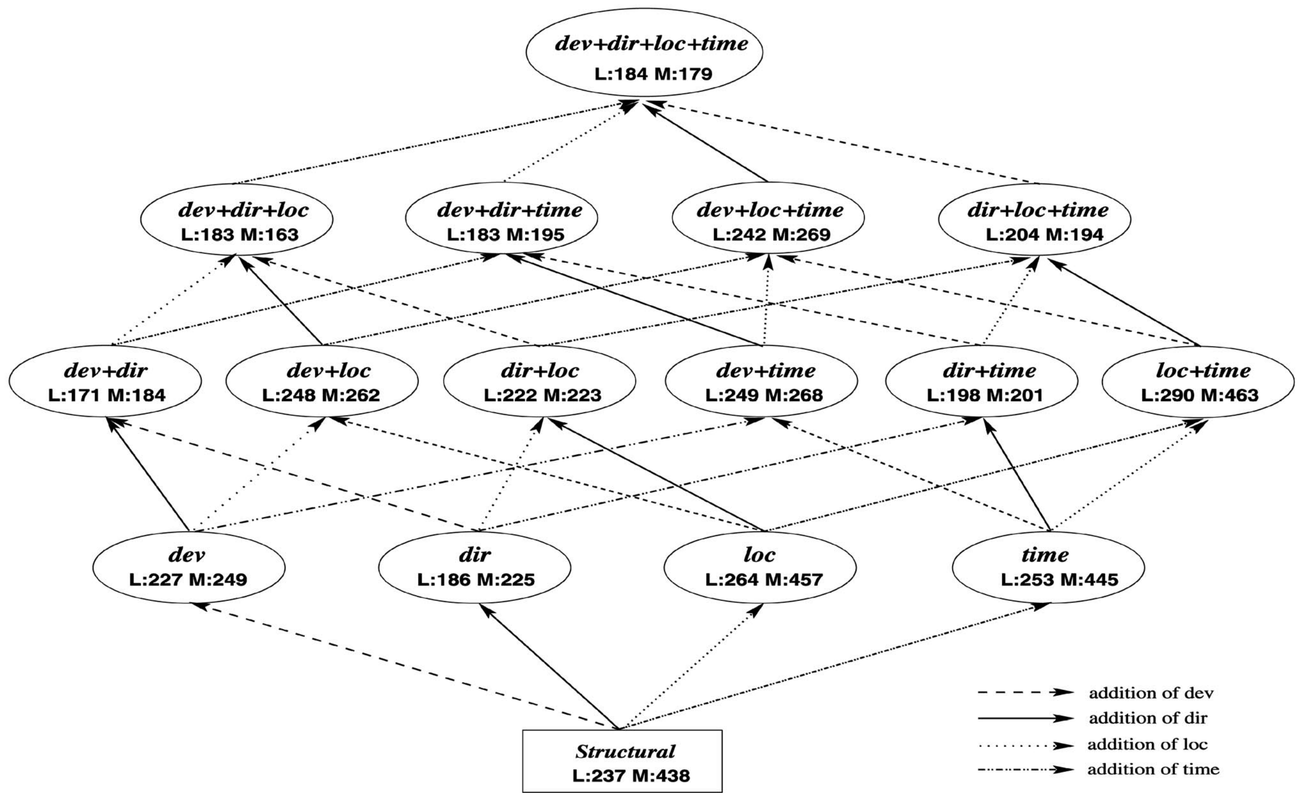

Fig. 3. Lattice of combinations of nonstructural attributes for Linux (L) and Mozilla (M).

each level of the lattice. For example, the addition of *dir* is given by a solid arrow. There are 15 possible combinations of nonstructural attributes that can be added to the structural information.

Each combination of nonstructural attributes in Fig. 3 is annotated with the MoJo distance between the decomposition created by LIMBO and the authoritative decomposition for Linux (L) and Mozilla (M). For example, for the combination *dev+dir+loc+time* at the top of the lattice, `L:184 M:179` means that the MoJo distance between the automatic decomposition and the authoritative one was 184 for Linux and 179 for Mozilla. The results are also given, in ascending order of MoJo distance, in Table 12. The table also includes the number of clusters that the proposed decomposition had in each case.

In order to avoid basing our results solely on the use of the MoJo distance measure, we measured the similarity of the proposed decompositions to the authoritative one using two other measures, namely, the Koschke-Eisenbarth (KE) measure [19] and EdgeSim [24]. Both of these are normalized to a percentage scale. The higher their value, the closer the two decompositions are. The values we obtained are also presented in Table 12.

Let us start our discussion by comparing the various evaluation measures. Although there are clearly differences of opinion between them, one can see that, in the case of Mozilla, all measures clearly agree that the four less effective combinations are *structural*, *time*, *loc*, and *loc+time*. Furthermore, as the MoJo value decreases, the values of the other two measures tend to increase. Things are not as clear in the case of Linux, but a similar trend can still be detected.

This observation implies that all measures would agree on the qualitative side of our results presented below, i.e., that our hypotheses on the usefulness of various nonstructural attributes have been confirmed. As a result, the rest of this discussion focuses primarily on the MoJo distance values.

As already hinted at above, an immediate observation is that certain combinations of nonstructural attributes produce clusterings with a smaller MoJo distance to the authoritative decomposition than the clustering produced when using structural input. However, in other cases, the MoJo distance to the authoritative decomposition has increased.

A closer look reveals some interesting trends:

- Following a solid arrow in the lattice always leads to a smaller MoJo value. This indicates that the inclusion of directory structure information produces better decompositions, an intuitive result.
- Following a dashed arrow leads to a smaller MoJo value as well, although the difference is not as dramatic as before. Still, this indicates that ownership information has a positive effect on the obtained clustering, a result that confirms the findings of Holt and Bowman [10].
- Following a dotted arrow decreases the quality of the obtained decomposition (in a few cases, at the top of the lattice, the quality remains practically unchanged). This confirms our expectation that using the lines of code as a basis for software clustering is not a good idea.

TABLE 12
Number of Clusters and Evaluation Measure Results for the Proposed Decompositions of Linux and Mozilla

Linux ($\phi = 0.0$)					Mozilla ($\phi = 0.2$)				
Combination	k	MoJo	KE	EdgeSim	Combination	k	MoJo	KE	EdgeSim
dev+dir	29	171	27	72.33	**dev+dir+loc**	63	163	72	63.84
dev+dir+loc	60	183	34	67.38	**dev+dir+loc+time**	34	179	62	69.77
dev+dir+time	27	183	32	72.82	**dev+dir**	30	184	58	71.90
dev+dir+loc+time	51	184	33	69.06	**dir+loc+time**	55	194	56	64.94
dir	61	186	35	67.13	**dev+dir+time**	58	195	64	64.09
dir+time	60	198	33	67.16	**dir+time**	41	201	52	63.81
dir+loc+time	92	204	38	66.44	**dir+loc**	37	223	51	64.46
dir+loc	96	222	41	66.42	**dir**	31	225	56	69.00
dev	70	227	25	66.61	**dev**	75	249	58	61.62
structural	56	237	24	66.65	**dev+loc**	66	262	55	63.94
dev+loc+time	45	242	24	68.08	**dev+time**	99	268	56	60.49
dev+loc	69	248	26	66.01	**dev+loc+time**	47	269	53	64.58
dev+time	73	249	25	67.43	**structural**	75	438	20	57.03
time	77	253	28	66.08	**time**	106	445	28	55.50
loc	96	264	31	66.26	**loc**	74	457	14	57.80
loc+time	76	290	21	66.88	**loc+time**	86	463	32	55.50

All combinations include structural information.

- Finally, following the arrows that indicate addition of *time* leads mostly to worse clusterings, but only marginally. This indicates that time could have merit as a clustering factor, but maybe in a different setting. It is quite possible that, if we obtain information about which files are being developed around the same time by examining the revision control logs of a system, we will get better results.

In summary, the experimental results confirm our hypothesis that directory structure and ownership information are important factors for the software clustering process, while lines of code are not. Temporal information might require more careful setup before it can be effective. These results present a strong indication that the method presented in this section can be used to assess the usefulness of other nonstructural attributes as well. Further research is, of course, required in order to determine whether these results hold true for a variety of software systems or are particular to open-source systems, such as Linux and Mozilla.

6 Evaluation of Weighting Schemes

One of the important properties of our approach is the fact that different weights can be applied to individual attributes (or their features) according to their importance for the clustering process. In this section, we present experimental results using established weighting schemes that assign importance based on the structure of the data set being clustered. In a software clustering scenario, software architects may assign weights based on their expert knowledge of the software system in question.

The weighting schemes we considered are described below. The weights derived by each weighting scheme can be applied directly to (7).

1. **Mutual Information** (MI). Given a set of features $\mathbb{B}$, the correlation of two features b_i and b_j can be computed using the mutual information $I(b_i; b_j)$. The higher the mutual information is, the stronger this correlation. We suggest computing the weight $w(b_j)$ for each feature b_j as the average mutual information between b_j and each other feature b_i, $1 \leq i \leq |\mathbb{B}|$, $i \neq j$, computed by

$$w(b_j) = \frac{1}{|\mathbb{B}| - 1} \sum_{i=1, i \neq j}^{|\mathbb{B}|} I(b_i; b_j).$$

   Intuitively, this scheme considers the features with high correlation to all other features as more important.
2. **Linear Dynamical Systems** (LDS). Dynamical Systems have been previously used in the clustering of attribute values in a database table [15]. Given a set of features, we assign an initial set of weights to them, called the *initial configuration* w_0 (commonly, the initial weights are equal). Then, the dynamical system applies a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ to create a new vector w_1. This process repeats until we reach a point where $f(w_i) = w_i$. This is called the *fixed point* of the dynamical system.

   A common example for function f is the summation operator (in this case, the dynamical system is called "linear"). This operator computes the new weight for a particular feature by summing the old weights of the features with which it cooccurs in the

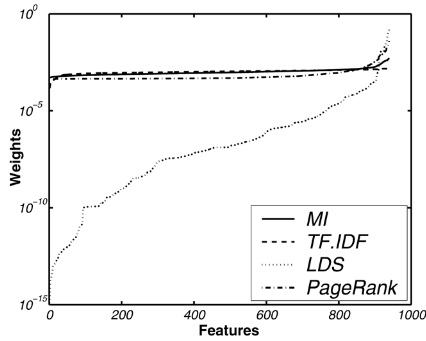

Fig. 4. TOBEY weights.

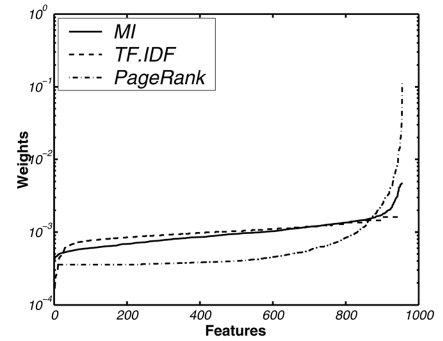

Fig. 5. LINUX weights.

artifacts. It has been shown that linear dynamical systems always converge [15].

In our case, we used a linear dynamical system with all features having equal weight in the initial configuration. The weight distribution used in the experiments is the weight vector at the fixed point of the dynamical system.

3. **TF.IDF**. This is a scheme widely used in information retrieval to assign weights to document terms [7]. In our case, the documents are the vectors representing the artifacts, and the terms are their features.

   Given an artifact $a_i \in \mathbb{A}$, the weight of feature $b_j \in \mathbb{B}$ is given by

$$w(b_j) = tf(b_j) \cdot \log\big(idf(b_j)\big),$$

   where $tf(b_j)$ (term frequency) is the frequency of feature b_j in the vector of a_i and $idf(b_j)$ (inverse document frequency) is the fraction n/n_{b_j}, where n is the number of artifacts, and n_{b_j} is the number of vectors containing the feature b_j.

   Intuitively, the *TF.IDF* weight of a feature is high if this feature appears many times within a vector and, at the same time, a smaller number of times in the collection of the vectors. The latter means that this feature conveys high discriminatory power. For example, in a data set of software artifacts, file `stdio.h`, which is used by a large number of software files, will have a lower weight compared to file `my_vector.h`, which is connected to a smaller fraction of files.

4. **PageRank**. This is a weighting scheme proposed and widely used in search engines [12] to compute a webpage's importance (or relevance). *PageRank* can be used when the relationships among different artifacts are given by a graph. The main idea behind *PageRank* is that a feature is deemed important if it has a large number of incoming edges from other important features.

   Early experiments with *PageRank* indicated that this weighting scheme was not beneficial to the clustering process. For this reason, we also experimented with *Inverse PageRank*, a weighting scheme that reverses the order in which *PageRank* assigns weights to the different features.

5. **Dynamic Usage**. Edges in a dependency graph indicate only potential relationships between the objects they connect. For example, a procedure call may or may not be executed when the system is run. Furthermore, it is quite common that particular edges are heavily used, while others are used only rarely.

   These observations indicate that the static picture of a dependency graph might belie what actually happens when the system it represents is in use. It is intuitive to conjecture that the amount of usage of a particular object is related to its importance.

   For this reason, the fifth weighting scheme we chose is based on dynamic usage. Assuming that each edge in the dependency graph of a software system is associated with a weight that represents its usage, each feature in the corresponding artifacts was assigned a weight equal to the weight of the edge that connects the node represented by the feature to the node represented by the artifact. The weights of the features in the same artifact were, of course, normalized prior to the execution of LIMBO.

Mozilla was the only software system that was used to evaluate the usage data weighting scheme. The main reason for this was the fact that, in order to extract meaningful dynamic usage data from a software system, one needs a comprehensive test suite that ensures good coverage of as many execution paths as possible. Such a test suite was not available for TOBEY or Linux. However, we were able to use the Mozilla "smoketests" for this purpose. The dynamic dependency graph we obtained contained information about 1,202 of the 2,432 source files that are compiled under Linux. The results presented in this section are based on the decomposition of these 1,202 files.

It is important to note that, since some of the weighting schemes we experimented with assume a graph-based data set, only structural features were included in our experiments.

Figs. 4, 5, and 6 present the weight distribution for the three software systems and the applicable weighting schemes. In all the figures, the y-axis is in logarithmic scale. Weights were sorted in ascending order to facilitate visualization. Since the various weighting schemes order features differently, a particular feature will have a different x-coordinate for each plot. For the same reason, the graph for *Inverse PageRank* would be exactly the same as that for *PageRank*.

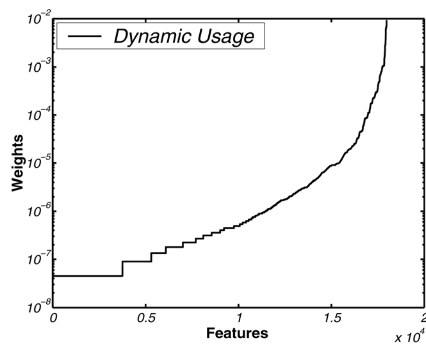

Fig. 6. MOZILLA weights.

In Fig. 4, we present the weight distribution of all four schemes applied to TOBEY. We observe that *MI* and *PageRank* produce weights in the same range. Although *TF.IDF* also produces weight values in the same range as *MI* and *PageRank* for the majority of the features, there is a number of them (30 or so) which were assigned smaller weight values. These are shown as points in the graph of the *TF.IDF* scheme, which correspond to the features closer to the origin of the x-axis. In the case of *LDS*, the weights produced are rather smaller and with a larger range. Larger weights correspond to nodes with large in and out-degrees.

The weight distribution in the Linux data set for all schemes follows the same pattern as in the TOBEY data set. To illustrate the differences among the weights of *MI*, *TF.IDF*, and *PageRank* schemes, we chose not to draw the distribution of *LDS* in Fig. 5. This figure shows that *MI* elicits the most conservative weights (all within one order of magnitude). On the other hand, *PageRank* gives a high weight to a number of nodes. These are nodes with a large number of incoming edges from other important nodes in the dependency graph of Linux. Finally, similar to the case of TOBEY, there are several features to which *TF.IDF* assigns smaller weight values.

Finally, Fig. 6 depicts the weight distribution produced by the dynamic usage weighting scheme. The main observation here is that there is a wide range of weights (the smallest weights are five orders of magnitude smaller than the largest ones). The distributions of the other four schemes for Mozilla are similar to the previous two data sets described above. *TF.IDF* assigns a smaller weight to approximately 3 percent of the features, as was the case with TOBEY and Linux.

The clustering results we obtained are shown in Table 13 (weighting schemes performing best are shown in bold).

An immediate observation is that the *TF.IDF* weighting scheme outperforms all others, including the scheme that uses no weights. This can be attributed to the fact that the way *TF.IDF* assigns weights corresponds well to the way software architects would assign importance to artifacts of their system. Artifacts used by the majority of the system are probably library functions that are not very important (low *idf*), while artifacts rarely used are unlikely to be central to the system's structure (low *tf*). Indeed, the low *TF.IDF* values assigned to a small percentage of the features (3 percent or so) as described before correspond to omnipresent nodes in the dependency graphs of the example software systems. The presence of these features in a large number of artifacts can result in unrelated artifacts being clustered together. The reduction of their importance through the low *TF.IDF* weight can only increase the quality of the obtained clustering.

A further observation is that the *LDS* weighting scheme performs quite poorly. Its weight distribution forecasted a deviant behavior, but the most likely explanation is the fact that it assigns large importance to nodes of large in and out-degree. This property is shared by the *PageRank* weighting scheme. Our results corroborate that this is not a desirable property for software data.

On the other hand, it was intriguing to observe that the *Inverse PageRank* weighting scheme produced results that were among the best. In fact, the difference between its results and those of *TF.IDF* is practically negligible. This indicates that importance for Web search engines does not imply importance for software clustering algorithms. In fact, quite the opposite seems to be the case.

Another interesting observation is that the dynamic usage weighting scheme performs rather well with the Mozilla data set. Even though it is outperformed by some weighting schemes, it still produces a decomposition of equal quality to the static dependency graph (represented by the None weighting scheme). Further experiments with a more complete test suite are required to evaluate this weighting scheme better.

It is interesting to note that the *MI* weighting scheme performs fairly well. With the exception of TOBEY, it is only

TABLE 13
Experimental Results for All Systems and Waiting Schemes

TOBEY ($\phi = 0.0$)		
Scheme	k	$MoJo$
None	84	316
MI	33	341
LDS	59	476
TF.IDF	**102**	**292**
PageRank	24	571
Inverse PageRank	86	297

LINUX ($\phi = 0.0$)		
Scheme	k	$MoJo$
None	56	237
MI	70	237
LDS	41	286
TF.IDF	**81**	**225**
PageRank	24	340
Inverse PageRank	79	226

MOZILLA ($\phi = 0.2$)		
Scheme	k	$MoJo$
None	75	438
MI	125	428
LDS	32	528
TF.IDF	**68**	**406**
PageRank	48	478
Inverse PageRank	98	407
Dynamic	61	440

slightly edged by *TF.IDF* and *Inverse PageRank*. It would be interesting to investigate whether *MI* is particularly appropriate for open-source systems, or that these results are only coincidental.

Finally, it should be noted that, in the interest of completeness, we performed experiments with the inverse version of all weighting schemes. The performance of *LDS* improved slightly, but not significantly, while the performance of the rest dropped. This indicates that the *MI*, *TF.IDF*, and dynamic usage weighting schemes in their pure form encapsulate well the properties of software decompositions.

The results presented in this section indicate clearly that a well-chosen weighting scheme can have a significant impact on the effectiveness of a software clustering approach. Further experimentation with more weighting schemes and software systems is, of course, required to establish the properties of weighting schemes that work well in the context of software clustering.

7 Conclusions

This paper presented the novel notion that information loss minimization is a valid basis for a software clustering approach. We developed an algorithm called LIMBO that follows this approach and showed that it performs as well, if not better, than existing algorithms.

Our approach has the added benefit that it can incorporate in the software clustering process any type of information relevant to the software system. This allowed us to develop a method for the assessment of the usefulness of nonstructural attributes. We validated this method by assessing the usefulness of four different nonstructural attributes of established usefulness.

We also presented a study on the applicability of weighting schemes to the clustering process. The results indicated that the *TF.IDF*, *MI*, and *Inverse PageRank* schemes can be quite effective.

Certain avenues for further research present themselves. Further experimentation with more software systems is one of our plans. The decompositions created by LIMBO will also have to be evaluated in an empirical study, where developers of the clustered systems provide feedback on them. We are definitely excited to investigate other attributes that are potentially useful to the software clustering process. Such attributes include date of creation, revision control logs, as well as concepts extracted from the source code using various concept analysis techniques. We would also like to collect dynamic usage data from more software systems in order to assess the usefulness of the dynamic usage weighting scheme better. We are, of course, excited to investigate other types of weighting schemes that are potentially useful to the clustering process.

Information theory concepts were also used by Anquetil and Lethbridge [6] to measure the mutual information and assess the redundancy between various features. We are interested to investigate how this work relates to ours, especially with respect to the *MI* weighting scheme.

Our weighting approach assigns weights that reflect the importance of each feature with respect to the rest of the features. One could argue that a feature with a high weight is unlikely to be uniformly that much more important when compared to all the other features in the system. We intend to investigate more localized weighting schemes in the future.

Finally, we plan to investigate the possible benefits from combining various nonstructural attributes with our weighting schemes. It is quite possible that choosing the appropriate nonstructural information together with an effective weighting scheme might be the best way to obtain a meaningful automatic decomposition for a given software system.

Acknowledgments

The authors would like to thank Nicolas Anquetil for providing the implementation of the cluster analysis algorithms, Rainer Koschke for the implementation of the Koschke-Eisenbarth measure, and Spiros Mancoridis and Brian Mitchell for the Bunch tool. They are also grateful to Ivan Bowman and R.C. Holt for the developer information of the Linux kernel. Finally, they thank Panayiotis Tsaparas, Renée J. Miller, and Kenneth C. Sevcik for their contribution to the development of LIMBO, as well as the anonymous reviewers for insightful comments on earlier drafts. This work was supported in part by the National Sciences and Engineering Research Council of Canada (NSERC).

References

[1] P. Andritsos and R.J. Miller, "Reverse Engineering Meets Data Analysis," *Proc. Ninth Int'l Workshop Program Comprehension*, pp. 157-166, May 2001.

[2] P. Andritsos, P. Tsaparas, R.J. Miller, and K.C. Sevcik, "LIMBO: Scalable Clustering of Categorical Data," *Proc. Ninth Int'l Conf. Extending DataBase Technology (EDBT)*, 2004.

[3] N. Anquetil and T. Lethbridge, "File Clustering Using Naming Conventions for Legacy Systems," *Proc. Ann. IBM Centers for Advanced Studies Conf.*, pp. 184-195, Nov. 1997.

[4] N. Anquetil and T. Lethbridge, "Experiments with Clustering as a Software Remodularization Method," *Proc. Sixth Working Conf. Reverse Eng.*, pp. 235-255, Oct. 1999.

[5] N. Anquetil and T.C. Lethbridge, "Recovering Software Architecture from the Names of Source Files," *J. Software Maintenance: Research and Practice*, vol. 11, pp. 201-221, May 1999.

[6] N. Anquetil and T.C. Lethbridge, "Comparative Study of Clustering Algorithms and Abstract Representations for Software Remodularisation," *IEE Proc. Software*, vol. 150, pp. 185-201, June 2003.

[7] R.B. Yates and B.R. Neto, *Modern Information Retrieval.* Addison-Wesley-Longman, 1999.

[8] M. Bauer and M. Trifu, "Architecture-Aware Adaptive Clustering of OO Systems," *Proc. Eighth European Conf. Software Maintenance and Reeng.*, pp. 3-14, Mar. 2004.

[9] I.T. Bowman and R.C. Holt, "Software Architecture Recovery Using Conway's Law," *Proc. Ann. IBM Centers for Advanced Studies Conf.*, pp. 123-133, Nov. 1998.

[10] I.T. Bowman and R.C. Holt, "Reconstructing Ownership Architectures to Help Understand Software Systems," *Proc. Seventh Int'l Workshop Program Comprehension*, May 1999.

[11] I.T. Bowman, R.C. Holt, and N.V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture," *Proc. 21st Int'l Conf. Software Eng.*, May 1999.

[12] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks*, vol. 30, nos. 1-7, pp. 107-117, 1998.

[13] S.C. Choi and W. Scacchi, "Extracting and Restructuring the Design of Large Systems," *IEEE Software*, pp. 66-71, Jan. 1990.

[14] T.M. Cover and J.A. Thomas, *Elements of Information Theory.* Wiley and Sons, 1991.

[15] D. Gibson, J.M. Kleinberg, and P. Raghavan, "Clustering Categorical Data: An Approach Based on Dynamical Systems," *Proc. Conf. Very Large Databases*, 1998.

[16] M.W. Godfrey and E.H.S. Lee, "Secrets from the Onster: Extracting Mozilla's Software Architecture," *Proc. Second Int'l Symp. Constructing Software Eng. Tools (CoSET)*, 2000.

[17] D.H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Eng.*, vol. 11, no. 8, pp. 749-757, Aug. 1985.

[18] R. Koschke, "Atomic Architectural Component Recovery for Program Understanding and Evolution," PhD thesis, Inst. for Computer Science, Univ. of Stuttgart, 2000.

[19] R. Koschke and T. Eisenbarth, "A Framework for Experimental Evaluation of Clustering Techniques," *Proc. Eighth Int'l Workshop Program Comprehension*, pp. 201-210, June 2000.

[20] C. Lindig and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis," *Proc. 19th Int'l Conf. Software Eng.*, pp. 349-359, May 1997.

[21] R. Lutz, "Recovering High-Level Structure of Software Systems Using a Minimum Description Length Principle," *Proc. 13th Irish Conf. Artificial Intelligence and Cognitive Science*, Sept. 2002.

[22] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *Proc. Int'l Conf. Software Maintenance*, 1999.

[23] E. Merlo, I. McAdam, and R.D. Mori, "Source Code Informal Information Analysis Using Connectionist Models," *Int'l Joint Conf. Artificial Intelligence*, pp. 1339-1344, 1993.

[24] B.S. Mitchell and S. Mancoridis, "Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements," *Proc. Int'l Conf. Software Maintenance*, pp. 744-753, Nov. 2001.

[25] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification," *J. Software Maintenance: Research and Practice*, vol. 5, pp. 181-204, Dec. 1993.

[26] N. Slonim, R. Somerville, N. Tishby, and O. Lahav, "Objective Classification of Galaxies Spectra Using the Information Bottleneck Method," *Monthly Notices of the Royal Astronomical Soc. (MNRAS)*, vol. 323, no. 270, 2001.

[27] K. Sartipi and K. Kontogiannis, "A Graph Pattern Matching Approach to Software Architecture Recovery," *Proc. Int'l Conf. Software Maintenance*, pp. 408-419, Nov. 2001.

[28] R.W. Schwanke, "An Intelligent Tool for Re-Engineering Software Modularity," *Proc. 13th Int'l Conf. Software Eng.*, pp. 83-92, May 1991.

[29] R.W. Schwanke and M.A. Platoff, "Cross References Are Features," *Second Int'l Workshop Software Configuration Management*, pp. 86-95, 1989.

[30] M. Siff and T. Reps, "Identifying Modules Via Concept Analysis," *Proc. Int'l Conf. Software Maintenance*, pp. 170-179, Oct. 1997.

[31] N. Slonim and N. Tishby, "Agglomerative Information Bottleneck," *Proc. Conf. Neural Information Processing Systems (NIPS-12)*, pp. 617-623, 1999.

[32] N. Slonim and N. Tishby, "Document Clustering Using Word Clusters via the Information Bottleneck Method," *Proc. 23rd Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pp. 208-215, 2000.

[33] N. Tishby, F.C. Pereira, and W. Bialek, "The Information Bottleneck Method," *Proc. 37th Ann. Allerton Conf. Comm., Control and Computing*, 1999.

[34] V. Tzerpos and R.C. Holt, "MoJo: A Distance Metric for Software Clusterings," *Proc. Sixth Working Conf. Reverse Eng.*, pp. 187-193, Oct. 1999.

[35] V. Tzerpos and R.C. Holt, "ACDC: An Algorithm For Comprehension-Driven Clustering," *Proc. Seventh Working Conf. Reverse Eng.*, pp. 258-267, Nov. 2000.

[36] Z. Wen and V. Tzerpos, "An Optimal Algorithm for MoJo Distance," *Proc. 11th Int'l Workshop Program Comprehension*, pp. 227-235, May 2003.

[37] C. Xiao and V. Tzerpos, "Software Clustering Based on Dynamic Dependencies," *Proc. Ninth European Conf. Software Maintenance and Reeng.*, to appear.

Periklis Andritsos received the BSc degree in electrical and computer engineering in 1998 from the National Technical University of Athens, Greece. In 2000, he received the MSc degree in computer science and, in 2004, the PhD degree in computer science, both from the Department of Computer Science at the University of Toronto. He is currently a postdoctoral fellow at the University of Toronto. His research interests include database systems, data mining, clustering, and reverse engineering. He is a member of the IEEE Computer Society and the ACM.

Vassilios Tzerpos received the BSc degree in electrical and computer engineering from the National Technical University of Athens, Greece, in 1992, and the MSc and PhD degrees in computer science from the University of Toronto in 1995 and 2001, respectively. He is currently an assistant professor in the Department of Computer Science and Engineering at York University, Toronto. His research interests are in reverse engineering, software clustering, and design pattern detection. He is a member of the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.