Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 244–254
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Clustering large software systems at multiple layers

Bill Andreopoulos [a,*], Aijun An [a], Vassilios Tzerpos [a], Xiaogang Wang [b]

[a] Department of Computer Science and Engineering, York University, Toronto, Ont., Canada M3J1P3
[b] Department of Mathematics and Statistics, York University, Toronto, Ont., Canada M3J1P3

Received 5 July 2006; accepted 25 October 2006
Available online 8 December 2006

Abstract

Software clustering algorithms presented in the literature rarely incorporate in the clustering process dynamic information, such as the number of function invocations during runtime. Moreover, the structure of a software system is often multi-layered, while existing clustering algorithms often create flat system decompositions.

This paper presents a software clustering algorithm called MULICsoft that incorporates in the clustering process both static and dynamic information. MULICsoft produces layered clusters with the core elements of each cluster assigned to the top layer. We present experimental results of applying MULICsoft to a large open-source system. Comparison with existing software clustering algorithms indicates that MULICsoft is able to produce decompositions that are close to those created by system experts.

Keywords: Software clustering; Multiple layer; Categorical; MULIC; Graph

1. Introduction

Reverse engineering is the process of analyzing a system's internal elements and its external behavior and creating a structural view of the system. Automatic construction of a structural view of a large legacy system can significantly facilitate the developers' understanding of how the system works. In legacy systems, the original source code is often the only available source of information about the system and it is very time consuming to study.

Software clustering techniques aim to decompose a software system into meaningful subsystems, to help new developers understand the system. Clustering is applied to large software systems in order to partition the source files of the system into clusters, such that files containing source code with similar functionality are placed in the same cluster, while files in different clusters contain source code that performs dissimilar functions. Software clustering can be done *automatically* or *manually*. Automatic clustering of a large software system using a clustering tool is especially useful in the absence of experts or accurate design documentation. It is desirable to have a software clustering tool that can consider both static and dynamic system information. Automatic clustering techniques generally employ certain criteria (i.e., low coupling and high cohesion) in order to decompose a software system into subsystems [13,12,17]. Manual decomposition of the system is done by software engineers. However, it is time consuming and it requires full knowledge of the system.

We propose the MULICsoft software clustering algorithm that is based on the MULIC categorical clustering algorithm that is described in [2]. MULICsoft differs from MULIC in that it incorporates both static and dynamic information (i.e., the number of function calls during runtime) in the software clustering process. MULICsoft handles dynamic information by associating weights with file dependencies and incorporating the weights in the clustering process through special similarity metrics. We showed that MULIC clustering results are of higher quality than those of other categorical clustering algorithms, such as *k*-Modes, ROCK, AutoClass, CLOPE and others [2]. Characteristics

* Corresponding author.
E-mail addresses: billa@cse.yorku.ca (B. Andreopoulos), aan@cs.yorku.ca (A. An), bil@cse.yorku.ca (V. Tzerpos), stevenw@mathstat.yorku.ca (X. Wang).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.10.010

of MULIC and MULICsoft include: *a*. The algorithm does not sacrifice the quality of the resulting clusters for the number of clusters desired. Instead, it produces as many clusters as there naturally exist in the data set. *b*. Each cluster consists of layers formed gradually through iterations, by relaxing the similarity criterion for inserting objects (files) in layers of a cluster at different iterations.

Section 2 gives an overview of previous software clustering tools. Section 3 describes the formulation of the input data for our clustering approach. Section 4 describes the MULICsoft clustering algorithm. Section 5 describes experimental results on the Mozilla system. Section 6 discusses inputting additional data to MULICsoft. Section 7 discusses evaluation of the results using an alternative measure. Section 8 discusses the runtime performance. Finally, Section 9 concludes and discusses future work.

2. Related work

Several clustering algorithms for software have been presented in the literature [3,5,6,9,11,12,14,15,17,20]. Some of the existing software clustering tools, such as LIMBO [3], consider dynamic information (i.e., the number of function calls during runtime) in the clustering process, while others, such as Bunch [12] and ACDC [17], produce clusters with a nested structure. MULICsoft both considers dynamic information and produces clusters with a layered structure.

In this section, we describe three algorithms: Bunch [12], ACDC [17] and LIMBO [3,4]. In Section 5, we will compare our proposed algorithm to these established software clustering algorithms.

Bunch is a clustering tool intended to aid the software developer and maintainer in understanding, verifying and maintaining a source code base [12]. The input to Bunch is a module dependency graph (MDG). Fig. 1 shows an MDG graph. Bunch views the clustering problem as trying to find a good partition of an MDG graph. Bunch defines a "good partition" as a partition where highly interdependent modules are grouped in the same cluster (representing subsystems) and independent modules are assigned to separate clusters. Fig. 1b shows a "good" partitioning of Fig. 1a. Finding a good graph partition involves systematically navigating through a very large search space of all possible partitions for that graph. Bunch treats graph partitioning (clustering) as an optimization problem. The goal of the optimization is to maximize the value of an objective function, called modularization quality (MQ) [12].

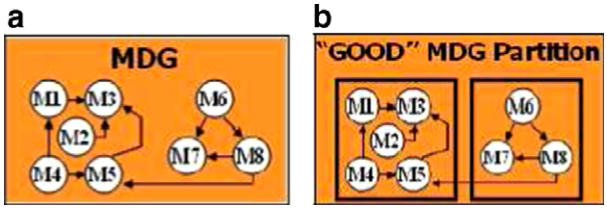

Fig. 1. (a and b) An MDG graph and its partition of maximum MQ [12].

ACDC works in a different way from other algorithms. Most software clustering algorithms identify clusters by utilizing criteria such as the maximization of cohesion, the minimization of coupling, or some combination of the two. ACDC performs the task of clustering in two stages. In the first stage, it creates a skeleton of the final decomposition by identifying subsystems that resemble established subsystem patterns, such as the body-header pattern and the subgraph dominator pattern [17]. Depending on the pattern used the subsystems are given appropriate names. In the second stage, ACDC completes the decomposition by using an extended version of a technique known as Orphan Adoption [19]. Orphan Adoption is an incremental clustering technique based on the assumption that the existing structure is well established. It attempts to place each newly introduced resource (called an orphan) in the subsystem that seems "more appropriate". This is usually a subsystem that has a larger amount of connectivity to the orphan than any other subsystem.

LIMBO is introduced in [4] as a scalable hierarchical categorical clustering algorithm that builds on the *Information Bottleneck* (*IB*) framework for quantifying the relevant information preserved when clustering. LIMBO has been successfully applied to the software clustering problem [3]. LIMBO's goal is to create clusters whose features contain as much information as possible about the features of their contents. LIMBO considers weights representing dynamic dependencies in the software clustering process.

3. Description of data sets

Static information on a software system represents dependencies between the objects to be clustered. In our case, the objects to be clustered are source files, while the dependencies are procedure calls and variable references. Static information on software systems is categorical, meaning that the objects have attribute values that are taken from a set of discrete values and the values have no specified ordering. We represent static information as a categorical data set by creating an $n \times n$ matrix M, where n is the number of files. Each row of M represents a file i of the software system. The categorical attribute value (CA) in cell (i,j) of M is '0' or '1', where '1' represents that file i calls or references file j and '0' represents that file i does not call or reference file j.

Dynamic information on a software system contains the results of a profiling of the execution of the system, representing how many times each file called procedures in other files during runtime. We represent dynamic information by associating a weight with each CA in the matrix, in the range 0.0–1.0, where 1.0 represents that file i called file j the maximum number of times during the runtime and 0.0 represents that file i did not call file j. Fig. 2 shows an example of a software data set in the form of a matrix.

The weights were derived by normalizing the number of procedure calls during an execution profiling, by dividing all numbers of calls in a column by the maximum number