SCuV: a Novel Software Clustering and Visualization Tool

Xiaomin Xu Sheng Huang Yanghua Xiao * Wei Wang

School of Computer Science, Fudan University, Shanghai, China

{10210240041,shhuang,shawyh,weiwang1}@fudan.edu.cn

Abstract

Decomposing a software system into smaller, more manageable clusters provides an insight for better comprehension of large systems for software engineers. However, invocation-awareness and dynamic view are two features which are not supported by existed software clustering visualization tools. In this paper, we presents a novel tool, named SCuV, to partition the ***S***oftware into invocation-aware clusters, ***C***l***u***ster them with nested containment & invocation hierarchy and ***V***isualize the clustering result in granularity-adjustable way.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

Keywords software clustering, static source code analysis

1. Introduction

Software clustering is widely used to gain insight into the softwares' architecture, which is critical for software maintenance activities such as architectural module reuse, legacy system reengineering and modification impact analysis.

In general, current related works on software clustering could be roughly classified into two groups according to the data source from which the clusters are extracted. The first group is documentation based approaches which extract clusters from design documentation such as UML graphs [1]. These approaches have an obvious drawback since they rely on existing design conventions. The second group is based on source code analysis which extracts system modules according to the dependency graph built from source code files. Dependencies considered include function invocation [2, 4], variable reference or directory structure of source code files [5] etc. Then, top-down or bottom-up clustering frameworks are usually adopted to build clustering hierarchy. Generally speaking, source code based approaches are more promising when system design documentations are unavailable. However, there still exists a number of challenges remaining unsolved for software clustering.

1. Previous works aren't aware of the invocation order among modules. For example, given a target module, users may be interested to find out which modules are invoked by it and how they are invoked. Unfortunately, existed visualization tools for clustering result make it difficult for users to obtain such information directly.
2. In real applications, dynamic views of clustering result are preferred. A good visualization tool should support zoom-in and zoom-out operations to explore modules of interest. In this way, users can not only have an overview of the system but also gain the insight into the detailed invocation relations between modules.

In this work, we developed SCuV, which is invocation aware and supports granularity-adjustable visualization of clusters. The tool can easily be ported to iPAD or large touch-screen system for better interactive usability in individual software comprehension and better software engineering experience in team discussion.

2. . Clustering Approach Implementation

The architecture of our tool consists of three parts: the Source Code Analyzer, Hierarchical Clustering and Visualization. Our approach takes source code as input, clusters source code on-line and provides users with a granularity-adjustable and hierarchical view of modules.

2.1 Source Code Analyzer

The source code analyzer extracts call graph of function from source code. It serves as the input of the clustering procedure. Currently, the tool only support java and it could be easily extended to support other programming languages. Notice that we just consider explicit function call when building call graph in this version. In the next version, implicit dependencies such as classloading dependencies and reflection will be considered.

2.2 Hierarchical Clustering

In this part, functions in the call graph are clustered into higher-level modules on which module invocation-dependency graph (MIDG) is built level by level iteratively until a single

* Correspondence author. We thank Tao Yu for implementing visualization of this demo. This work was supported by NSFC under grant Nos 61003001, 61170006 and 61033010. Specialized Research Fund for the Doctoral Program of Higher Education No. 20100071120032.

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

cluster is formed. Our clustering approach has two phases: *partition* and *clustering*. In the partition phase, we sperate the input call graph into connected components, each of which will be fed into the clustering phase. In the clustering phase, a hierarchical cluster is generated for each connected component. The clustering phase contains two steps: *Entry-based clustering* and *PageRank-based clustering*.

In *Entry-based clustering* phase, the following steps are executed iteratively for each connected component C. We first detect the *entry* functions in C. Then, for each entry function f, we find all functions only invoked by f, which form the *dominated set* of f and merge them into a larger module. Then we delete these functions from C iteratively until there is no more entry functions. After functions in C are merged into larger modules, we build module invocation-dependency graph $MIDG_C$ as follows. $MIDG_C(V, E)$ is a weighted directed graph with each $v \in V$ representing a module and each directed edge $e_{i,j} \in E$ representing the dependency of module i on module j. The edge weight, $w_{i,j}$ on $e_{i,j}$ represents the coupling degree between the two endpoints and is defined as the fraction of inner-modules in v_i that invoke at least one inner-module in v_j.

In *PageRank-based clustering* phase, for each $MIDG_C$, we run a variation of the PageRank algorithm [3] on $MIDG_C$ to determine the coupling cost for each vertex $v \in MIDG_C$ by $PageRank(v)$. Then we process each vertex in the decreasing order of vertices' PageRank score as follows: We find its k out-neighbors with the largest coupling degree to merge them into a higher-level module and then delete them from $MIDG_C$. Then we build a higher-level MIDG on these newly merged modules by the approach described in *Entry-based clustering* phase. The above two clustering steps are executed alternatively and iteratively until $MIDG_C$ are merged into a single cluster. Next, we introduce the naming strategy of new modules.

Naming Strategy We use package and class names to name new modules. First, functions are labeled by their signatures which include names of packages and class as prefix. Modules generated in *entry-based clustering* phase are named by the labels of their *entry* functions. For each module generated in *PageRank-based clustering* phase, its name is derived from names of its internal lower-level modules by adopting textural mining techniques proposed in [1].

2.3 Visualization

Visualization of the clustering result provides users with a hierarchical and dynamic adjustable view of the software systems. The left part of Fig. 1 shows the highest level of the cluster result of Java based system Weka[1](ver. 3.0) which contains 10 packages, 147 class files with a total of 95 KLOC of source code. In this view, rectangles represent the currently visible internal modules contained in higher-level modules represented by shadow parts. The label on rectangle is module name derived by our naming strategy.

[1] http://www.cs.waikato.ac.nz/ml/weka/

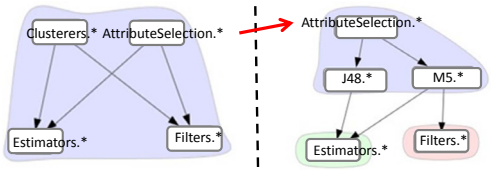

Figure 1. Screenshot of Clustering Result for Weka.

The arrows reveal invocation dependency between modules. Visible modules are displayed in a top-down fashion according to their invocation order. For example, in the left part of Fig. 1, because module labeled by *"AttributeSelection.*"* invokes module labeled by *"Estimators.*"*, the former is displayed on top of the latter one. When the user zoom in to the module labeled by *"AttributeSelection.*"*, its internal modules will be displayed in the right part of Fig. 1.

3. Demonstration

The goal of demonstration. Through the demo, we plan to show the following two aspects of SCuV : (1) our tool can produce meaningful invocation-aware hierarchial clustering; (2) our tool supports dynamic visualization with the flexibility to navigate clustering in different granularity.

The way of demonstration. We will let participants use SCuV to understand the target java project-Weka, with the comparison to understand Weka by using IDE like Eclipse. First of all, the presenter will introduce the implementation of SCuV with a deck. Then the Weka project will be loaded into Eclipse. The presenter will show participants how to understand the Weka project using Eclipse. In contrast, the presenter will load Weka into SCuV and show the comprehension process: (1)navigate from higher level modules for a global view of cluster result; (2)zoom in to explore the detailed invocation & containment relationship; (3)zoom out to hide the detail and display the higher level clustering. Meanwhile, we encourage participants to browse the visualized result to experience the benefits of SCuV. Besides, we hope to discuss with participants to see how the implementation of SCuV can be improved.

References

[1] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, pages 201–221, 1999.

[2] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98*, page 45.

[3] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *ICSE*, pages 848–858, 2012.

[4] C. Patel, A. Hamou-Lhadj, and J. Rilling. Software clustering using dynamic analysis and static dependencies. CSMR '09, pages 27–36, 2009.

[5] V. Tzerpos and R. C. Holt. Acdc : An algorithm for comprehension-driven clustering. WCRE '00, pages 258–267.