Using Decomposition to Produce High-Level System Organization of Software Source Code

Violeta T. Bojikova[1]

Abstract – **Software clustering techniques are useful for extracting architectural information about a system directly from its source code structure. In this paper is discussed the evaluation problem of clustering algorithms.**

Keywords – **software clustering algorithms, program decomposition, program restructuring**

I. Introduction

Software architecture is a critical asset to a project due to the ever increasing complexity and the demand to reduce maintenance cost for evolution. One of the areas in software architecture is architecture recovery through reverse engineering of existing implementations.

Clustering techniques have been used in many disciplines to support grouping of similar objects of a system. Clustering analysis is one of the most fundamental techniques adopted in science and engineering. The primary objective of clustering analysis is to facilitate better understanding of the observations and the subsequent construction of complex knowledge structure from features and object clusters. Examples include botanic species and mechanical parts. The key concept of clustering is to group similar things into clusters, such that intra-cluster similarity or cohesion is high, and inter-cluster similar or coupling is low. Coupling has great impact on many quality attributes, such as maintainability, verifiability, flexibility, portability, reusability, interoperability, and expandability. The main objective of clustering is similar to that of software partitioning.

Most existing clustering approaches often are limited to architecture recovery activity in the reverse engineering process only. But clustering techniques can be applied to software during various life-cycle phases. Clustering techniques can be used to effectively support both software architecture partitioning at the early phase in the forward engineering process and software architecture recovery of legacy systems in the reverse engineering process. Lung demonstrated that clustering techniques can also be used to effectively facilitate software architecture restructuring instead of simply being used for software architecture recovery of existing systems.

Because the structure of software systems are usually not documented accurately, researchers have expended a great deal of effort studying ways to recover design artifacts from source code. Since many software systems are large and complex, appropriate abstractions of their structure are needed to simplify program understanding and restructuring.

For small systems, source code analysis tools can easy extract the source level components (e.g., modules, classes, functions) and relations (e.g., method invocation, function calls, inheritance) of a software system. For large systems there is significant value in identifying the abstract (high-level) entities, and then modeling those using architectural components such as subsystems and their relations.

Subsystems generally consist of a collection of source code resources that collaborate with each other to implement a feature or provide a service to the rest of the system. Typical resources found in subsystems include modules, classes, and possibly, other subsystems. Subsystems facilitate program understanding by treating sets of source code resources as high-level entities.

The entities and relations needed to represent software architectures (high-level component) are not found in the source code. Thus, without external documentation, we seek other techniques to recover a reasonable approximation of the software architecture using source code information. Researchers in the reverse engineering community have applied a variety of software clustering approaches to address this problem.

Many of the clustering techniques published in the literature can be categorized by the way they create clusters. These techniques determine clusters (subsystems) using source code component similarity concept analysis, optimization [144,145], or information available from the system implementation such as module, directory, and/or package names.

II. Fundamental Questions Pertaining to the Software Clustering Problem

1. How can a software engineer determine – within a reasonable amount of time and computing resources – if the solution produced by a software clustering algorithm is good or not?

2. Can an algorithm be created that guarantees a solution – within a reasonable amount of time and computing resources – that is close to the ideal solution?

From a practical aspect, the answers to these questions are important because they provide increased confidence to software engineers who analyze systems. From a theoretical aspect, these answers are important because they provide an approximation algorithm to a known NP-Hard problem, in addition to a method for comparing any solution, even those produced by other algorithms that use the same clustering

[1]Violeta Bojikova is with the Department of Computer Science Varna Technical University, Bulgaria, e-mail: vbojikova@yahoo.com, bojikov@nat.bg

criterion we do (i.e., coupling-cohesion tradeoff), to the optimal solution.

III. Sub-Optimal Decomposition Algorithm – SOAD

In this paper is presented an evaluation of a clustering algorithm, which first version is presented in [5-7] and which uses heuristic search technique to determine the subsystems of a software system. The goal of the software clustering process is to partition the graph model – MDG of the system into a set of clusters such that the clusters represent subsystems.

Since graph partitioning is known to be NP-hard, obtaining a good solution by random selection or exhaustive exploration of the search space is unlikely. SOAD overcomes this problem by using heuristic-search techniques.

Formalization: The MDG $= (X, U)$ is a directed graph where the source code components are modeled as nodes, and the source code dependencies are modeled as edges:

– X is finite set of components (nodes), where $N = |X|$ is the number of components – classes, modules, files, packages, etc.;

– $U \subseteq X \times X$ is the set of ordered pairs $\langle x_1, x_2 \rangle$ that represent the source-level relationships between module x_1 and module x_2 (inherit, import, include, call, instantiate, etc.)

Once the MDG is created, SOAD produce an initially partition of the MDG and evaluates the "quality" of this partition using a fitness function that is called Modularization Quality (k) [doklad]. After producing the initially solution from the search space, SOAD improves it using iterative algorithm. Given that the fitness of an individual partition can be measured, heuristic search algorithms are used in the iterative clustering phase in an attempt to improve the MQ of the initially generated partition. SOAD implement a hill-climbing algorithm.

The Goal of SOAD is to "Find a good partition of the MDG."

A partition is the decomposition of a set of elements (i.e., all the nodes of the graph) into mutually disjoint clusters.

A "good partition" is a partition where:

– highly interdependent nodes are grouped in the same clusters;

– independent nodes are assigned to separate clusters The k function is designed to penalize excessive inter-cluster coupling. k increases as the inter-edges (i.e., external edges that cross cluster boundaries) increase.

Modularization Quality and restrictive condition. Modularization Quality (k) is a measurement of the "quality" of a particular MDG partition.

The assumption is: "Well designed software systems are organized into cohesive clusters that are loosely interconnected".

The weight – W_k of each cluster $g \in$ MDG with $x_i \in X$ components (nodes) corresponds to the restrictive condition W_0, where w_i – is the label of node x_i and present the number of node's elements (i.e. functions):

$$W_k = \sum_{x_i \in X_k} w_i \Longleftarrow W_0.$$

The value of "k", where k_{ij} is the number of inter-edges (i.e., external edges that cross cluster boundaries) between nodes x_i and x_j is calculated as follow:

$$k = \sum_{i=1,\ldots,M} \sum_{j=1,\ldots,M} k_{ij} = \min, \quad \forall i \neq j.$$

M represents the number of clusters in the current partition of the MDG.

IV. Comparing the Results Produced by Software Clustering Algorithms

Now that a plethora of approaches to software clustering exist, the validation of clustering results is starting to attract the interest of the Reverse Engineering research community. Numerous clustering approaches have been proposed in the reverse engineering literature, each one using a different algorithm to identify subsystems. Since different clustering techniques may not produce identical results when applied to the same system, mechanisms that can measure the extent of these differences are needed.

Many of the clustering techniques published in the literature present case studies, where the results are evaluated by the authors or by the developers of the systems being studied. This evaluation technique is very subjective. Recently, researchers have begun developing infrastructure to evaluate clustering techniques, in a semi-formal way, by proposing similarity measurements [1-3]. These measurements enable the results of clustering algorithms to be compared to each other, and preferably to be compared to an agreed upon "benchmark" standard. Note that the "benchmark" standard needn't be the optimal solution in a theoretical sense. Rather, it is a solution that is perceived as being "good enough".

Existing clustering techniques neither provide a guarantee on the quality of their solutions nor any indication of a solution's proximity to the optimum. Bunch [1,3], for example, uses several methods to find solutions, such as hill-climbing and genetic algorithms. Hill-climbing only guarantees local optimality, but makes no guarantees of global optimality.

Genetic algorithms are another type of search, like hill climbing, that does not guarantee the quality of its solution, not even with respect to local extreme. Neither method indicates how good a solution is with respect to the optimal solution. Not being able to meet either of these criteria is unsatisfactory.

Researchers have begun formulating ways to measure the differences between system decompositions.

For example, Anquetil et al. developed a similarity measurement based on Precision and Recall.

Much of the research on measuring the similarity between decompositions only considers the assignment of the system's modules to subsystems. Mancoridis and al. [1] argue that a more realistic indication of similarity should consider how much a module depends on other modules in its subsystem, as well how much it depends on the modules of other subsystems.

Mojo measures the distance between two decompositions of a software system by calculating the number of operations

needed to transform one decomposition into the other [4]. The transformation process is accomplished by executing a series of Move and Join operations. In MoJo, a Move operation involves relocating a single resource from one cluster to another, and a Join operation takes two clusters and merges them into a single cluster.

Tzerpos and Holt also introduce a quality measurement based on MoJo. The MoJo quality measurement normalizes MoJo with respect to the number of resources in the system. Given two decompositions, A and B, of a system with N resources, the MoJo quality measurement is degfined as:

$$\text{MoJoQuality}(A; B) = (1 - \text{MoJo}(A; B)/N) * 100\%$$

Koschke and Eisenbarth present in 2000 a framework for experimental evaluation of clustering techniques [2]. The goal this evaluation is to have an oracle to compare the results of automatic techniques.

Let software engineers detect modules $\longrightarrow$ references R

Let automatic techniques propose modules $\longrightarrow$ candidates C

Let compare candidates to references

– identify immediate corresponding candidates and references $\longrightarrow$ good match

– identify corresponding submodules; i.e., a module corresponds only to a part of another module $\longrightarrow$ O.k. match

– measure accuracy of correspondences $\longrightarrow$ detection quality

Types of matchs:

1. Good match $C \approx pR$:
Iff $|\text{elements}(C) \cap \text{elements}(R)|/|\text{elements}(C) \cup \text{elements}(R)| >= p$, where p is a tolerance parameter.

- if $p = 1$, C and R must be the same
- more pragmatically, $p = 0.7$
  – C and R overlap at 70%
  – $\{a, b, c, d\} approx 0.7\{b, c, d\}$, overlap is $3/4 = 0.75$
  – $\{a, b, c, d\}$ no $\approx 0.7\{b, c, d, e\}$, overlap is $3/5 = 0.6$

2. Part-of matches
Matching relation $S \subseteq pT$:
Iff $|\text{elements}(S) \cap \text{elements}(T)|/|\text{elements}(S)| >= p$, where p is tolerance parameter as above S is part of T

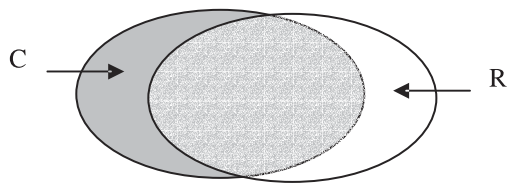

Fig. 1. Mutually part-of matches

3. Mutually part-of matches

- $C \approx pR \Rightarrow C \subseteq pR \wedge R \subseteq pC$
- but not: $C \subseteq pR \wedge R \subseteq pC \Rightarrow C \approx pR$
- yet, there is a distinct correspondence between C and R (fig. 1): if $C \subseteq pR \wedge R \subseteq pC$

$$\text{Overlap}(C, R) = (C \cap R)/(C \cup R) < 70\%$$

C and R are a mutually part-of match:

iff $C \subseteq pR \wedge R \subseteq pC$

– mutually part-of matches are part-of matches

– good matches are mutually part-of matches

Part-of and mutually part-of are O.k. matches

4. The accuracy of each match is:

$$T(C, R) = \text{overlap}(C, R),$$

where: $\text{overlap}(C, R) = (C \cap R)/(C \cup R)$.
Accuracy $T(M)$ for class of matches is:

$$T(M) = \frac{\sum_{C_i, R_i \in M} T(C_i, R_i)}{|M|}.$$

5. There are multiple aspects of detection quality [2]:
– Numbre of false positives and true negatives
– Granularity of matches = good matches/all matches
– Accuracy of each match and the class of matches

6. SOAD is evaluated using the Koschke similarity technique and the Mojo distance evaluation. The result, produced by SOAD are stable. Ideally, the results produced by SOAD could be compared to an optimal reference solution, but this option is not possible since the graph partitioning technique used by SOAD for software clustering is NP-Hard.

In the case is used the benchmark standard. SOAD has been tested for open-source software systems, which has

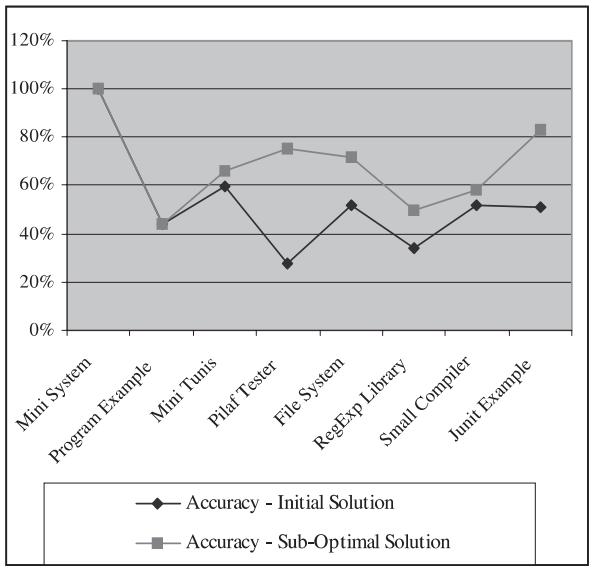

Fig. 2. Accuracy results

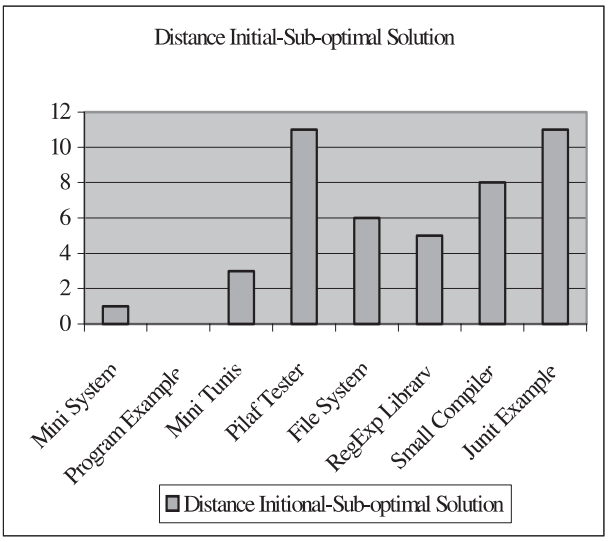

Fig. 3. Distance evaluation

been used by other clustering algorithms [3]: Apache Regular Expression Class Library (RegExp Package), Mini Tunis, Small Compiler, File System, PlafTester Program etc. All systems are small or middle size. We dispose with the MDG graph for these systems and with the reference partition (i.e. a partition presented by the experts, or by other techniques for clustering, or in the design documentation).

Figures 2,3 show the clustering results and the value of the accuracy of each class of matches (initial partition and sub-optimal partition). The results depend from the graph strength (number of nodes and edges).

V. Conclusion

Most interesting software systems are large and complex and, as a consequence, understanding their structure is difficult. Such software systems are composed of many resources. The structure of these systems can be represented as a graph where the nodes are the resources and edges are the relations between the resources.

Without automated assistance the software structure graph provides little value when being used to understand the design of practical systems because of its large number of nodes and edges.

Decomposing source code components and relations into subsystem clusters is an active area of research. The primary goal of clustering tools is to propose subsystems that expose abstractions of the software structure. However, the various clustering tools use different algorithms, and make different assumptions about how subsystems are formed.

Now that many clustering techniques exist, some researchers have turned their attention to evaluating their relative effectiveness. There are several reasons for this: Many of the papers on software clustering formulate conclusions based on case studies, or by soliciting opinions from the authors of the systems presented in the case studies.

Much of the research on measuring the similarity between decompositions only considers the assignment of the system's modules to subsystems. We argue that a more realistic indication of similarity should consider how much a module depends on other modules in its subsystem, as well how much it depends on the modules of other subsystems.

When decompositions are compared, all source code resources tend to be treated equally. Conclusions are often formulated based on the value of a similarity or distance measurement.

In the paper we examine two similarity measurements that have been used to compare decompositions.

We show the results of our study of similarity measurements. The conclusion is that SOAD shows good results referring to these measurements.

References

[1] Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements, Spiros Mancoridis and Brian Mitchell IEEE Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01). IEEE.

[2] Rainer Koschke and Thomas Eisenbath, "A Framework for experimental evaluation of clustering techniques", *International Workshop on Program Comprehension*, June, 2000.

[3] Search Based Reverse Engineering, by B. S. Mitchell, S. Mancoridis, M. Traverso. In the *ACM Proceedings of the 2002 International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, Ischia, Italy, July, 2002. pp. 431-438.

[4] Mojo: A distance metric for software clustering. V. Tzerpos and R. C. Holt. In Proc. Working Conf. on Reverse Engineering, Atlanta - 1999, pp.187-193.

[5] Software Architecture Decomposition, Violeta Bojikova, M. Mitev, Proceedings of the 14th Int'l Conference SAER'2000, Varna, 2000, pp. 173-177.

[6] An Approach to measure the Cost of Program restructuring, Violeta Bojikova, M. Karova, Proceedings of papers, Volume 2, pp 669-671, 2002, Nish, Yugoslavia

[7] Elementary Operations and Program Restructuring, V.Bojikova, M.Karova, Proceedings of the Int'l Conference Tehnonav 2002, Constanca, June-2002, pp.192-197.