Cooperative Based Software Clustering on Dependency Graphs

by

Ahmed Fakhri Ibrahim

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

© Ahmed Fakhri Ibrahim 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The organization of software systems into subsystems is usually based on the constructs of packages or modules and has a major impact on the maintainability of the software. However, during software evolution, the organization of the system is subject to continual modification, which can cause it to drift away from the original design, often with the effect of reducing its quality.

A number of techniques for evaluating a system's maintainability and for controlling the effort required to conduct maintenance activities involve software clustering. Software clustering refers to the partitioning of software system components into clusters in order to obtain both exterior and interior connectivity between these components. It helps maintainers enhance the quality of software modularization and improve its maintainability.

Research in this area has produced numerous algorithms with a variety of methodologies and parameters. This thesis presents a novel ensemble approach that synthesizes a new solution from the outcomes of multiple constituent clustering algorithms. The main principle behind this approach derived from machine learning, as applied to document clustering, but it has been modified, both conceptually and empirically, for use in software clustering. The conceptual modifications include working with a variable number of clusters produced by the input algorithms and employing graph structures rather than feature vectors. The empirical modifications include experiments directed at the selection of the optimal cluster-merging criteria. Case studies based on open source software systems show that establishing cooperation between leading state-of-the-art algorithms produces better clustering results compared with those achieved using only one of any of the algorithms considered.

Acknowledgment

I would like to thank the Electrical & Computer Engineering Department at the University of Waterloo for funding this work.

Dedication

I dedicate my dissertation work to my family and many friends. A special feeling of gratitude to my loving parents, whose words of encouragement and push for tenacity ring in my ears. My wife has never left my side and are very special.
I also dedicate this dissertation to my friends in the ECE department who have supported me throughout my study. I will always appreciate all they have done, especially Susan King for help and support.

Contents

List of Tables viii

List of Figures ix

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Proposed Software Clustering Approach . . . 3
- 1.3 Thesis Statement . . . 3
- 1.4 Research Goals . . . 4
- 1.5 Organization of The Thesis . . . 5

2 Background and Related Work 6
- 2.1 Software Clustering Analysis . . . 6
  - 2.1.1 Fact Extraction . . . 6
  - 2.1.2 Filtering . . . 7
  - 2.1.3 Similarity Computation . . . 7
  - 2.1.4 Clustering Algorithms . . . 7
    - 2.1.4.1 Graph-theoretical Algorithms . . . 7
    - 2.1.4.2 Optimization Algorithms . . . 8
    - 2.1.4.3 Hierarchical Algorithms . . . 8
    - 2.1.4.4 Meta-Heuristic Algorithms . . . 9
  - 2.1.5 Results Visualization . . . 9
  - 2.1.6 User Feedback Collection . . . 10
- 2.2 Software Clustering Validation . . . 10
  - 2.2.1 Quality of Partitioning . . . 11
  - 2.2.2 Distance Between Decompositions . . . 12
  - 2.2.3 Stability . . . 12
- 2.3 Related Work . . . 12
- 2.4 Summary . . . 13

3 Cooperative-Based Software Clustering 14
- 3.1 Software Dependencies . . . 16
- 3.2 Dependency Structure Extraction . . . 16
- 3.3 Dependency Structure Partitioning . . . 17
- 3.4 Cooperative-Based Dependency Structure Partitioning . . . 18

3.4.1 Agreement Phase 18
3.4.2 Merging Phase 19
3.5 CC/G Complexity 20
3.6 Limitations of CC/G 20
3.7 Clustering Evaluation 21
3.8 Summary 21

4 Experimental Work 23
4.1 Experimental Procedure 23
4.2 Benchmark Programs 24
4.3 Constituent Clustering Techniques 25
4.4 Choosing an Efficient Merging Criterion 26
4.5 Clustering Evaluation 28
4.5.1 Modularization Quality 28
4.5.2 Similarity Measures 29
4.5.3 Stability of Clustering Solutions 31
4.6 Summary 33

5 Summary and Future Work 37
5.1 Research Summary 37
5.2 Current Research Achievements 38
5.3 Future Work: Enhancing Bug Classification using CC/G 38
5.3.1 Introduction 38
5.3.2 Clustering Defect Reports using CC/G 39
5.3.3 Experimental Setup 40
5.3.4 Conclusion 41

Bibliography 42

List of Tables

4.1 Benchmark Programs (First Set) 24
4.2 Benchmark Programs (Second Set) 25
4.3 TurboMQ Values for Acegi 0.5.0 28
4.4 TurboMQ Values for Cocoon 1.7 29
4.5 TurboMQ Values for Jabref 1.1 29
4.6 TurboMQ Values for Proguard 1.0 30
4.7 TurboMQ Values for Struts 0.5 30
4.8 TurboMQ Values for xwork 1.0 31
4.9 Improvement (%) in *TurboMQ* by CC/G 31
4.10 Similarity values (%) obtained by the NAHC, SAHC, Lattix, and CC/G with respect to the original partitioning (First set of software programs) 32
4.11 Similarity values (%) obtained by the NAHC, SAHC, Lattix, and CC/G with respect to the original partitioning (Second set of software programs) 32
4.12 Changes in consecutive versions for each benchmark programs . . . 32

List of Figures

3.1 Cooperative Clustering for Graph Structure (CC/G) Approach . . . 15
3.2 Dependency Structure Matrix 17

4.1 DSM Partitioning Example 27
4.4 Distance between two consequence version of benchmark programs 33
4.2 Comparison between CC/G-min and CC/G-max for Acegi0.5, Proguard 1.0, Xwork1.0 system 35
4.3 Quality of partitioning 36

5.1 Building a Bug Report Predication Model 40

Notations

ACDC	Algorithm for Comprehension-Driven Clustering
AIB	Agglomerative Information Bottleneck algorithm
CA	Combined algorithm
CC	Cooperative Clustering
CF	Cluster Factor
DSM	Dependency Structure Matrix
MoJo	Distance Matrics
MQ	Modularization Quality
original	Package Structure
SR	Stability Ration
SV	Software Visualization
TurboMQ	Turbo Modularity Quality

Chapter 1

Introduction

Software supports many business, government, and social institutions. As the requirements of these institutions change, so must the software that supports them; however, changing software systems that govern complex actions is much more easily said than done. Creating an adequate model of the structure of a system and keeping that model coherent with respect to modifications that occur as the system evolves are among the many critical problems that software developers are facing. Without an automated method of acquiring insight into the system design, a software maintainer often modifies the source code without a complete understanding of its organization. For such systems, it is apparent that adopting an ad hoc approach to software maintenance will have a negative impact on the quality of the structure of the system. In fact, the quality may diminish to the point where the system organization is so disorganized that it must be modernized or abandoned.

Software clustering techniques have been applied as a method of solving crucial software engineering problems in the context of reflexion analysis, software evolution, and reverse engineering. The basic idea behind reflexion analysis is the creation of a hypothesized static architecture from existing documentation or from interviews with architects, followed by the mapping of the elements of the implementation to the architecture [61]. For large software systems, such a manual process usually entails substantial work [47]. Combining automatic software clustering with prior knowledge about the system's architecture would have a significant impact on reflexion analysis. In practice, software clustering can identify concrete entities for which an automatic mapping decision is relatively simple and can also provide support for the user with respect to manual mapping through the detection of hypothesized entities that are possible correct entities [56].

The evolution of software systems occurs through the addition of new functionality, the correction of existing faults, and improvements in [56]. Software clustering tools attempt to improve the structure of the software through software restructuring or reduce the complexity of large modules source code decoupling. Software clustering is also useful for identifying duplicate code [68]. Decoupling source code can help simplify complex modules or functions, where complexity of a module or a function is determined based on software metrics. Xu et al. [68] have presented a case study in which software clustering is applied for source code decoupling at the

procedure level. With software clustering, related statements are grouped together to produce a dependency ranking among the groups. The authors suggested that a module or function be divided based on the results of the software clustering.

The fundamental goal of reverse engineering is to recover components or to extract system abstractions [15]. The literature includes a number of approaches and techniques that are supported by software clustering and that can facilitate the recovery of information from a software system [10, 12, 39, 56]. Two examples of typical reverse engineering are module recovery [12, 18, 23, 29] and architecture recovery [10, 39, 48, 58, 70]. Module recovery software clustering methods focus on recovering modules based on an analysis of dependencies extracted from the software system, such as function calls. A number of tools, such as Schwanke's Arch tool [53], Bunch [37], and ACDC [61], have been used to help software engineers understand a system through the decomposition of the system into subsystems. In practice, architecture recovery methods focus on determining the system architecture through the breaking down of abstractions distilled from the source code, such as components (modules), subsystems, and design patterns.

Most software clustering algorithms have been implemented for specific software systems, with substantial success. It is consequently crucial to examine new approaches that will enable the development of a cooperative strategy for successfully combining current algorithms to create robust solutions applicable for different types of systems.

1.1 Motivation

In the last 20 years, software clustering has been the target of an enormous amount of research [56]. The clustering techniques described in the literature use a number of criteria for establishing the clusters. As mentioned, most of these algorithms have been successfully applied in specific software systems. However, the question of how a software engineer can identify the software clustering algorithm best suited for a particular software system remains unanswered.

The importance of evaluating software clustering algorithms was first pointed out by Lakhotia and Gravley [32]. Evaluation methods involve an appraisal of software clustering approaches based on an assessment of the quality of an automatic decomposition using an authoritative decomposition as a reference. The drawback of these methods is the assumption that such a decomposition exists, given that its construction for a mid-sized software system is quite challenging, even with the help of an expert [55]. Numerous research studies have addressed the creation of an authoritative decomposition [31, 43]. However, questions related to determining the correct process for creating an authoritative decomposition and to why it is useful to create such authoritative compositions remain unanswered.

From a practical perspective, the answers to these questions are important because they increase the confidence of the software engineers who analyze systems. From a theoretical point of view, these answers are significant because they provide an approximate method for comparing any solutions, even those produced by other

algorithms that are based on the same clustering criterion.

1.2 Proposed Software Clustering Approach

Rather than relying on a single algorithm, researchers and practitioners in the document-clustering field have devised a new ensemble approach that synthesizes a solution from the results of an aggregation of constituent clustering algorithms. This ensemble approach is known as cooperative clustering (CC) [28]. The goal of using CC is to achieve a common benefit (e.g., a global clustering solution) based on agreement among peer techniques.

In this research, we propose a new clustering approach inspired by document-based cooperative clustering We designate our approach CC/G because it is based on graphs (i.e., software dependency graphs) rather than on numerical feature vectors, as in the original CC. The CC/G algorithm includes the three steps of the original CC: first, run each of the constituent clustering algorithms; second, from the output of the clustering algorithms, find the common elements that they agree on; third, merge these elements into existing (or new) clusters.

The particulars of how these steps are performed represent an additional difference between CC and CC/G. Conceptually, CC works with numerical feature vectors and includes assumptions about the number of clusters produced at each step. In contrast, CC/G works with graph structures, with a relaxing of the assumptions about the number of clusters generated. Empirically, in any domain a variety of merging criteria could be used, and the best criteria for each domain are discovered through experimentation. Such experiments have been performed for CC/G as applied to software clustering, with the conclusion that the best overall results are obtained by merging clusters that have low cohesion.

Case studies using open source software systems have demonstrated that combining state-of-the-art approaches based on cooperative methodology and merging produces better results than any of the individual approaches considered. The case studies were focused on a combination of the bottom-up clustering techniques embodied in the Bunch [37] and Lattix [52] tools. MoJo [60] and Turbo Modularization Quality ($TurboMQ$) [43] metrics were employed for the assessment of the quality of the results, and the stability of the solutions was measured by monitoring the changes between subsequent versions of the benchmark programs.

1.3 Thesis Statement

This dissertation will establish the following thesis statement:

> Combining multiple state-of-the-art software clustering algorithms with a cooperative strategy produces measurably better and more stable results than any of those algorithms does individually.

The state-of-the-art software clustering algorithms that we combine cooperatively are Hill Climbing and Matrix Co-clustering in the Bunch framework, and the De-

pendent Structure Matrix (DSM) partitioning strategy used by the Lattix tool. We use Mitchell's TMQ metric to assess cluster quality and Tzerpos's MoJo metric to assess stability. These are well accepted metrics in software clustering, as we discuss in the related work (section 2.3).

1.4 Research Goals

This research investigates the tolerance of dependency-based structured software systems, the aggregation of multiple clusterings with a variable number of clusters, the discovery of common patterns in each clustering in the form of sub-clusters, and the determination of the best merging criteria that will result in efficient clusters. The scope of this research is wide, with the primary focus on how to build a software clustering algorithm based on the cooperative methodology inspired by the traditional cooperative categorization for text documents described by Kashef [28]. The ultimate goals of any software clustering algorithm include the production of high-quality partitions from raw software systems within an acceptable time frame, and the maintenance of the stability of the partitioning for changing versions of the software. The quality of the global clusters generated with the use of the proposed approach is to be measured based on the merging strategy adopted and the clustering algorithms invoked. Quality can achieved through the maintenance of high internal cohesion in the clusters and low coupling between them. In the context of cluster entities means that the produced partitions should be of real significance and represent the structures inherent in the software. Stability means that solutions remain to be maintained when similar clustering decompositions are produced for similar versions of a software system. The following are the characteristics of our proposed clustering algorithm:

1. The algorithm does not sacrifice the quality of the resulting clusters by creating the number of clusters desired. Instead, it produces as many clusters as naturally exist in the data set.

2. Each cluster consists of layers formed gradually through iterations.

3. An appropriate choice of merging criteria results in high-quality clustering solutions with stable compositions.

These goals could be further expanded to include the following features:

1. Management of dynamic information through the association of weights with file dependencies and the incorporation of these weights into the clustering process through special similarity metrics.

2. Selection of the best combination of heterogeneous clustering algorithms that have different characteristics and configurations.

3. Facilitation of the hybrid aggregation of both feature-based and dependency-based structured software.

4. Ability to choose a variety of objective functions for the generation of the sub-clusters.

5. Addition of quantitative measures of quality based not only on internal information (i.e., dependencies) but also on external information (e.g., class labels).

1.5 Organization of The Thesis

Chapter 2 reviews the software clustering literature, focusing mainly on representation extraction methods, software clustering algorithms, clustering evaluation approaches and related work. Chapter 3 presents the proposed software clustering approach in detail. Preliminary experimental results are discussed in chapter 4. Finally, a summary of the work, and a discussion of the research plan are given in chapter 5.

Chapter 2

Background and Related Work

In this chapter, we consider a number of broad areas related to our research, including software clustering analysis, data representation, and techniques for assessing the quality of a software clustering. In the following sections, we will present the state-of-art in software clustering analysis, describe the evaluation criteria used in this study, and review existing work related to combing multiple partitioning.

2.1 Software Clustering Analysis

Referring to the process of analyzing the relationships between software entities in such a way that the system organization can be reconstructed in a certain level of modularity. In general, this process is undertaken either for the purpose of gaining insight into the identity of the system components (modules), or for the purpose of obtaining a compressed representation of the system organization. This process can be feasible only when the system structure has a certain level of modularity.

In the context of software clustering, Shten [55] suggests that software cluster analysis can be done in the following stages: (1) fact extraction, (2) filtering, (3) similarity computation, (4) cluster creation, (5) results visualization, and (6) user feedback collection. The process typically repeats until satisfactory results obtained. Each stage discussed in detail in the following sub-sections.

2.1.1 Fact Extraction

Fact extraction is the term used to describe the program analysis to be used as input to the cluster algorithm. [56]. It starts with recognizing the set of entities to cluster. After entities have been identified, the next phase is attribute selection. Selecting an appropriate set of attributes for a given clustering task is critical for its success. There are different sources that software artifacts could be extracted from such as source code, binary modules and software documentation. Then, extracted artifacts are stored in a language-independent model, often called factbase [49]. The models can then be studied and manipulated to analyze the architecture of the software being built.

2.1.2 Filtering

Filtering is performed after fact extraction. Its aims are to remove unnecessary information from the factbases, to calculate facts that are a composition of existing facts, and apply a weighting scheme to the attributes. Lung et. [33] showed that using a different methodology for the building of the final factbase may directly affect the results. Ideally, the factbase should be small and have enough information to make the obtained clusters meaningful. Several tools have been developed for fact manipulation, such as Gork [25] and Crocopat [11]. Some algorithms expect that the factbase includes only relations between modules [54]; other algorithms do not make any assumptions about the facts [7].

2.1.3 Similarity Computation

Similarity measures between entities are the core component of any clustering algorithm [27]. The measures reflect the degree to which the elements of entities belong together. Thus, similarity measures reflect how strongly-related each part of functionality of a software module is.

Schwanke [53] introduced the concept of using design principles, such as low coupling and high cohesion. Schwanke's concept has been extended by Koschke [31] to metric-based hierarchical clustering techniques. The Koschke similarity functions consider global declarations, function calls and name similarities between identifiers and file names. Choi and Scacchi [16] also describe a similarity function based on maximizing the cohesiveness of clusters.

2.1.4 Clustering Algorithms

Software clustering, by definition, is an exploratory and descriptive software analysis technique, which has gained a lot of attention in software maintenance. The organization of a software system has a major impact on its maintainability. To improve maintainability, software systems are usually organized into subsystems using the constructs of packages or modules using clustering algorithms. Clustering algorithms can be categorized based on their cluster model. According to Wiggerts [65], software clustering algorithms can be categorized into graph-theoretical algorithms, optimization algorithms, hierarchical agorithms, and meta heuristic algorithms.

2.1.4.1 Graph-theoretical Algorithms

This set of algorithms is based on graph properties where nodes represent entities and edges represent relations. Aggregation algorithms and minimal spanning tree algorithms are the most common types of graph-theoretical clustering algorithms. Aggregation algorithms decrease the number of nodes in a graph by merging them into aggregate nodes. The aggregates can be used as clusters or can be the input for a new iteration resulting in higher-level aggregates. The minimal spanning tree (MST) algorithms start by discovering a MST of the given graph. Due to the nature of software, the classic MST algorithm is not suitable for software clustering

because it tends to create a few large clusters where contain many entities while several other entities isolated [58]. Bauer and Trifu [10] modified the classic MST algorithm for using in software clustering. This extended MST algorithm uses two passes. The first pass, which follows the classic MST concept, iteratively joins the two closest nodes into a cluster, while the second pass assigns the remaining un-clustered entities to the cluster they are the closest to.

2.1.4.2 Optimization Algorithms

A typical optimization algorithm starts with an initial solution and tries to enhance it by iterative adaptations, according to some fitness function [56]. Its effectiveness reliy on the successful preliminary choice of values for seven parameters that control factors, including the number of expected clusters, the minimum number of objects in the cluster, and the maximum number of iterations. These algorithms produce both hierarchical [35] and non-hierarchical [18] clustering. A classic non-hierarchical clustering optimization method starts with an initial partition derived based on some heuristic. Then, entities are replaced with other clusters to improve the partition according to some criteria. Hill-climbing is a search and optimization technique which has been successfully employed in various software clustering algorithms [18]. Mitchell [43] shows promising results in terms of the quality and performance of hill-climbing search methods. His approach has been implemented as part of the Bunch software clustering tool [44], [36]. Bunch is a clustering tool which supports software developers and maintainers in understanding, verifying and maintaining a source code base [37], [36], [23], [61]. Bunch interprets the clustering problem as an optimization problem, and tries to find a partition maximize an objective function where approximating the goodness of a partition. A good partition gathers highly interdependent modules in the same cluster (representing subsystems) and assigns independent modules to distinct clusters. Finding a good graph partition involves systematically navigating through a very large search space of all conceivable partitions using techniques such as hill-climbing [36] and genetic algorithms [57].

2.1.4.3 Hierarchical Algorithms

Hierarchical algorithms [38] employ a similarity measure to calculate pair-wise resemblances for all entities to be clustered. The most similar entities are gathered together iteratively until some heuristic rule decides that a satisfactory set of clusters has been reached. Hierarchical algorithms can be categorized into agglomerative (bottom-up) and divisive (top-down) [26]. Divisive algorithms begin with one cluster that holds all entities and splits the cluster into a number of disjoint clusters at each successive step. Agglomerative algorithms start at the bottom of the hierarchy by iteratively grouping like entities into clusters. Each step combining, the two clusters that are most like to each other, reducing the number of clusters by one. Divisive algorithms propose an advantage over agglomerative algorithms because most users are concerned about the main structure of the data which con-

sists of some large clusters found in the first steps of divisive algorithms [26]. But, agglomerative algorithms have been extensively used in software clustering. For example, Lung et al. [33] have shown applications of the UPGMA (Unweighted Pair Group Method with Arithmetic mean) method in the software clustering context. Andritsos and Tzerpos [7] presented the Scalable Information Bottleneck (LIMBO) algorithm, an agglomerative hierarchical algorithm that employs the Agglomerative Information Bottleneck algorithm (AIB) for clustering.

On the other hand, ACDC [61] is a hierarchical clustering algorithm that does not follow the classical schema of hierarchical algorithms. It cannot be assigned to the agglomerative or divisive category because the algorithm does not have an explicit iterative split or merge stage. ACDC uses patterns that have been shown to have good program comprehension properties to determine the system decomposition. This results in most modules being placed into hierarchical categories (subsystems).Then, ACDC uses an orphan adoption algorithm [59] to assign the remaining modules to the appropriate subsystem. ACDC [61] completes the task of clustering in two stages. In the first stage, it generates a skeleton of the final decomposition by finding subsystems that look like established subsystem patterns. Depending on the pattern used the subsystems are given appropriate names. In the second stage, ACDC finalizes the decomposition by using an extended version of the Orphan Adoption Algorithm. It tries to place each unclustered software entity in the subsystem that depends most on it.

2.1.4.4 Meta-Heuristic Algorithms

Many meta-heuristic methods have been successfully applied to software module clustering. Most proposed approaches generate clusters by analyzing only the structural dependencies between software entities [36],[39], [46],[8], [7]. The field was established by Mancorridis et. al. [36]. They implement an automatic clustering tool called Bunch [37]. This tool used a search-based clustering algorithms such as hill climbing for automating software module clustering. Several other meta-heuristic search technologies have been applied, including simulated annealing and genetic algorithms [22], [34], [44]. However, experiments show that hill climbing dominents other techniques in both result quality and execution time. To prepare software engineering problems as clustering problems, the representation and fitness function need to be defined [18], [22].

2.1.5 Results Visualization

Software Visualization (SV) denotes the use of visual representations to enhance the understanding and comprehension of the different aspects of a software system. Caserta et al. [14] classify architecture related approaches which involves visualizing relationships between software entities. Visualizing relationships in the software is a harder task because software entities can have a much larger number of relations of many kinds, such as inheritance, method calls, etc.

Graphs as vertices and edges (node-link) have all the characteristics needed

to represent relationships between software components [24]. However, visualizing large software systems using graphs can be very confusing, with plenty of edge congestion, overlapping, and occlusions, its context making it almost impossible to investigate an individual node or edge.

Dependency Structure Matrix (DSM) has been proposed in the literature as an alternative to the node-link representation graphs [52]. DSM is a square matrix with identical row and column labels. Matrix entries are the number of relations between a row element and a column element is shown in the matrix.

2.1.6 User Feedback Collection

Some software clustering algorithms accept (or require) user input, whereas others run totally automatically [55]. Algorithm which accept user input are called semi-automatic clustering algorithms. A software engineer may explore different aspects of a software system by validating clustering results and providing feedback to the semi-automatic clustering algorithm.

Semi-automatic algorithms aims to enable a collaboration between clustering algorithms and the user with the goal of constructing quality results. Koschke [31] proposed a semi-automatic clustering framework based on modified versions of the fully automatic techniques he investigated. Christl et al. [17] present a semi-automatic algorithm that maps the hypothesized high-level entities to source code entities.

Unfortunately, it is usually difficult to select between automatically/semi-automatically clustering algorithms due to the size and the lack of formatting of the extracted information. Therefore, being able to combine both approaches will help the software engineer to explore different aspects of the software system.

2.2 Software Clustering Validation

There are three types of validation studies in data clustering [26]. An external evaluation of validity matches up the recovered decomposition to a reference decomposition (often produced by an expert). An internal evaluation of validity tries to determine if the decomposition is intrinsically appropriate for the data by measuring the quality of a clustering based on the type of criterion being considered. A relative test compares two decompositions and measures their relative values.

Software clustering, Turbo Modularization Quality (TurboMQ) [43] and MoJo [60] are the measures that have been used in the literature to apply for software clustering evaluation. TurboMQ used to analyze the quality of partition (internal evaluation). MoJo calculate the minimum number of move or join operations one needs to perform in order to transform one clustering to another or vice versa. MoJo used for relative evaluation, by compare between two clusterings of the same software system. Also, MoJo could be used for external evolution, when asked to measure the distance between the clustering of a system and its reference decomposition. The following subsections discuss the three types in more detail.

2.2.1 Quality of Partitioning

Based on the assumption that well-designed software systems are organized into cohesive partitions (clusters) that are loosely inter-connected, two modularization criteria have been proposed in the literature to evaluate the quality of partitioning: Basic Modularization Quality (MQ) and Turbo Modularization Quality (MQ) [43].

The Basic MQ was the first modularization quality measure, introduced by Mancoridis et. al, [37]. It measures the connections between constitutes of two distinct clusters (coupling) and the connections between the constituents of the same cluster (cohesion) independently. MQ is designed to reward the creation of highly cohesive clusters while penalizing excessive inter-cluster coupling. However, Mitchell noted two significant drawbacks in basic [43]. First, the performance of this measurement restrict usage to small systems (i.e., fewer than 75 modules). The second problem with the Basic MQ measurement is that its design cannot support weighted graphs.

TurboMQ [42] was designed to overcome the two limitations of Basic MQ. TurboMQ supports weighted graphs, and has much lower computational complexity than Basic MQ. Turbo MQ has been used to evaluate many of the clustering techniques published in the literature including [9], [35] and [46].

To calculate TurboMQ, we need to compute two quantities: intra-connectivity $\mu_{i,j}$ and inter-connectivity $\varepsilon_{i,j}$. μ_i is the sum of all relationships that exist between classes in cluster i. A higher intra-connectivity suggests high cohesion. $\varepsilon_{i,j}$, is defined as the sum of all relationships that exist between classes in two distinct clusters i and j. Using these two quantities, a cluster factor CF_i is calculated for each cluster i and then $TurboMQ$ of the system is given by the sum of CF for all clusters. The cluster factor is calculated as:

$$CF_i = \begin{cases} 0 & \mu_i = 0; \\ \frac{\mu_i}{\mu_i + \frac{1}{2} \sum\limits_{i=1 j \neq i}^{k} (\varepsilon_{ij} + \varepsilon_{ji})} & otherwise. \end{cases} \quad (2.2.1)$$

The objective function approximates the cohesion and coupling of each cluster with respect to other clusters using CF definition.

TurboMQ is given by:

$$TurboMQ = \sum_{i=1}^{k} CF_i \quad (2.2.2)$$

TurboMQ measures the quality of CC/G generated clusters. It therefore allows us to compare the generated clusters from different algorithms using consistent criteria.

2.2.2 Distance Between Decompositions

The MoJo distance [60] measures the distance between a decomposition created by a clustering algorithm and an authoritative decomposition as the minimum number of operations one has to perform to transform a software decomposition to the authoritative one. In practice, it computes the number of move and join operations that needed to be performed in order to change one software decomposition to another. Intuitively, the smaller the distance of a decomposition to the authoritative one, the more effective the algorithm that produced it.

2.2.3 Stability

Stability reflects how sensitive is a clustering approach to perturbations of the input data. Stability in software clustering means that similar clustering decompositions should be produced for similar versions of a software system [55]. Thus, under conditions of small and incremental change between consecutive versions, an algorithm should be stable. To calculate the relative stability of two constituent clustering algorithms, Wu et al. [67] suggest constructing sequences of MoJo [60] values calculated based on comparing two consecutive members of the sequence of decompositions obtained from a software system. This measure allows one to say that one algorithm is more stable than another with regard to a software system. Tzerpos and Holt [62] define a stability measure based on the ratio of the number of "good" decompositions to the total number of decompositions produced by a clustering algorithm. A decomposition obtained from a slightly modified software system is defined as "good" if and only if the MoJo distance between the decomposition and the decomposition obtained from the original software system is at most 1% of the total number of entities.

2.3 Related Work

Kebir et al. [30] compare and combine two algorithms for software component identification from object-oriented source code. They started by defining a mapping model between objects and components and a measurement model for evaluating semantic correctness of a software component. Then, they continue combining a hierarchical and a genetic clustering algorithms in a collaborative manner. They choose a population that contains a local minimal solution obtained by hierarchical clustering to initial the genetic clustering algorithm.

Patel et al. [48] proposed a two-phase clustering approach that combines both dynamic (trace based) and static dependency analysis. They begin by building the core skeleton decomposition of the system, using software features as a clustering criterion. Then, they analyzed their static dependencies with the formed clusters. Zhang et al. [70] proposed a new hybrid clustering algorithm and partition clustering for recovering high-level software architecture from Weighted Directed Class Graph (WDCG). based on hierarchical clustering and partition clustering for software architecture recovery. In particular, they started by using hierarchical cluster-

ing to find out the kernels of clusters, and then they partition other vertices into the kernels. Daun et al. [21] proposed a consensus-based requirements clustering algorithm in which a predefined number of candidate clusterings (in their experiment 25) and then integrates them into a final clustering using voting scheme. Saeed et al.[51] developed a new linkage algorithm, thr Combined Algorithm (CA). The CA algorithm works by computing a new feature vector for each formed cluster. This new feature vector is built by taking the binary OR of the feature vectors of the entities that are clustered. When two entities are merged together, the combined cluster includes the types, globals and routines of its constituent entities. Hence, the new feature vector correctly reflects the merge.

2.4 Summary

The intention was to highlight these issues related and important to this research. Next chapter presents the deployment of the proposed approach in software engineering.

Chapter 3

Cooperative-Based Software Clustering

The idea of cooperative clustering [28] is to synthesize a clustering from the outputs of an ensemble of constituent clustering algorithms. It searches for a clustering that is better than those produced by the constituent input algorithms, using their outputs as starting point. In the worst case cooperative clustering reverts to simply selecting the best of the constituent outputs.

The original cooperative clustering approach for text documents [28] has three steps. First, it runs each of the constituent clustering algorithms. Second, it finds the points of agreement between these constituent clusterings in form of sub-clusters. Finally, a coherent merging of sub-clusters performed to reveal the expected number of clusters.

There are four aspects of cooperative clustering that should be modified to fit the paradigm of software systems rather than documents data sets. First, heterogeneous software clustering approaches will produce different number of clusters; the assumption in the document domain was that each constituent clustering technique would produce the same number of clusters. Second, the merging operation should handle a graph structure of the software rather than a numerical feature vector; source files should not clustered based only on lines of code but, rather, by the way in which they interact with each other. Third, the criteria used to decide what to merge should be tailored for software systems. We empirically explored a number of alternative criteria, as discussed in the experiments section. Finally, the original cooperative clustering had an assumption that the number of output clusters was known a prior, which is usually not the case in software systems. We have modified the cooperative clustering to address all of these points, and we named our modified version CC/G to indicate that it is based on graphs rather than numerical feature vectors. The four steps of CC/G are illustrated by the flowchart in Figure 3.1 and described in Figure 3.1.

These four steps are the three steps of the original CC plus an extra step at the beginning to extract the graph structure from the software.

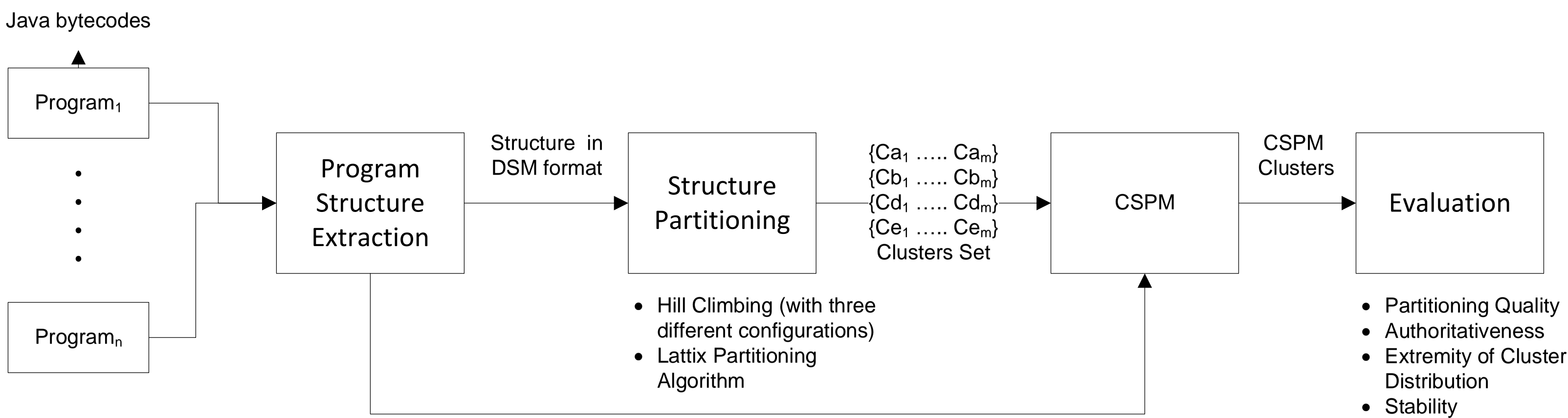


Figure 3.1: Cooperative Clustering for Graph Structure (CC/G) Approach

Algorithm 3.1 Pseudo-code for clustering program dependency structures using cooperative strategy

Input: The Dependency Structure Matrix, and a set of C clustering algorithms, $i = 1, 2, .., C$
Output: The desired clusters k
For all k

1. Generate the DSM Matrix among n software entities
2. Apply each of the input clustering algorithms on DSM to generate k_i clusters, $i = 1, 2, .., C$.
3. Repeatedly merge sub-clusters until k clusters are reached using the merging factor (mf).
4. Evaluate the quality of the k clusters using modularity quality measure.

End

3.1 Software Dependencies

Software dependencies has a great impact on software understandability, reusability, maintainability, and testability [20]. Dependencies are relations in which a change to item A can affect item B even though there is no direct dependency between A and B. This could happen, for example, when both A and B access data provided by item C. The likelihood of a change affecting other modules increases with the number of dependencies a module has.

A variety of dependency types have been presented and analyzed in previous research [66]. Dependencies can be classified by their abstraction level, by their static or dynamic nature, or by their impact weight. Two properties of dependencies can be identified: dependency extraction time and dependency significance Both apply to implementation level dependencies. Dependencies can be simple direct dependencies with one dependent and one dependee, or transitive dependencies that create a dependency path between multiple items, or cyclic dependencies, in which the dependency path starts and ends with the same item. There are various ways to model and examine the dependencies in software systems, ranging from informal box-and-arrow diagrams, UML diagram, and Dependency Structure Matrix (DSM). DSM has a clear benefit when dealing with large architectures. More about DSM is comping in the next section.

3.2 Dependency Structure Extraction

As a first step in our CC/G algorithm, we extract the static dependencies between the input software artifacts (Java byte-code programs in our case) and construct a compact representation of these dependencies using a Dependency Structure Matrix

(DSM) [52]. The DSM approach uses a matrix of rows and columns to show how each software entity depends on others within a project. Different tools are used to construct DSMs but only Lattix[52] suite is used in extracting and partitioning software architectures.

		1	2	3	4	L
$Element_1$	1	0	0	0	0	1
$Element_2$	2	1	0	1	0	2
$Element_3$	3	1	1	0	0	2
$Element_4$	4	0	0	1	0	3

(a) Simple DSM with 4 elements and 5 dependency marks.

		1	2	3	4	L
$Element_4$	1	0	0	0	0	1
$Element_1$	2	1	0	1	0	2
$Element_3$	3	1	1	0	0	2
$Element_2$	4	0	0	1	0	3

(b) Partitioned DSM

Figure 3.2: Dependency Structure Matrix

Figure 3.2a provides an example of the general formulation of the DSM of four software elements. In the DSM, each row and column represent a file in the target system. Each cell DSM (i, j) represents a static dependency (such as a reference to a variable or a data type, a call to a function) from one file to another. A cell DSM(i, j) with value 1 means there is a dependency between element i and element j otherwise DSM$(i, j) = 0$. L in figure 3.2b represents cluster label. In this figure, the DSM elements grouped into three clusters.

3.3 Dependency Structure Partitioning

In this step, we employ a number of clustering algorithms. Each algorithm i generates a set of k_i clusters. We use Dependency Structure Matrix (DSM) component partitioning by Lattix[52] and Hill climbing partitioning algorithms with different settings based on their availability and their discussion in the literature.

The DSM partitioning algorithm tries to re-arrange rows of the DSM matrix to have clusters of elements near the diagonal. Several algorithms and heuristics have been offered to aid in determining appropriate objective functions and optimization [52]. Genetic algorithms, distance (from the diagonal) penalty computed for each interaction and other algorithms have been used to cluster DSMs [13].

DSM partitioning is a ‘matrix reordering’ which discovers a permutation of matrix rows and columns such that the resulting matrix is as ‘compact’ as possible. The ordering of row and columns in the matrix is important because it dictates what the DSM will look like and how it should be interpreted. Figure 3.2b shows an example of DSM partitioning.

In Hill Climbing, a graph known as Module Dependency Graph (MDG) is used instead of the DSM. The MDG represents all of the software entities (classes) in the system as nodes, and all of the dependencies between nodes as edges. The edges in

the MDG may be weighted to emphasize the relative strength of the dependency among nodes. Candidate clusters are determined by moving nodes between the clusters, or in some cases creating new clusters, to maximize an objective function which is defined by Mitchell [45] as Turbo Modularization Quality ($TurboMQ$). This function has the property of rewarding cohesive clusters, while penalizing extreme coupling between clusters.

Every software system is clustered using the preceding algorithms, which generate a flat decomposition of the benchmark systems. Every individual solution of the software partitioning problem is represented by a vector of integers L. This vector is generated by assigning a cluster number to each entity. For example, in Figure 3.2b, elements are considered to be grouped in 3 clusters.

Each tool has its own objective function, but all based on the software engineering concepts of coupling and cohesion. Generally, cohesion refers to the degree to which the elements of a module belong together [69]. Methods of measuring cohesion vary from qualitative measures to quantitative measures based on the scope of source code being analyzed. In this study, we use quantitative cohesion to classify the source code. We measure how tightly classes in the same component are connected. Cohesion metrics usually cooperate with coupling metrics to measure quality of software clusters [19]. Subsystems with high cohesion are considered preferable because high cohesion is associated with several desirable features of software including robustness, reliability, reusability, and understandability. Coupling is the measure to which each subsystem relies on other subsystems. It is believed that subsystems exhibiting high cohesion and low coupling form well designed systems[19]. Hence, the resulting decompositions should have more intra-cluster relationships (i.e. dependencies in clusters) and fewer of inter-cluster relationships (i.e. dependencies between clusters).

3.4 Cooperative-Based Dependency Structure Partitioning

This step is the core of the proposed approach. It takes the DSM and a set of partitioning algorithms outcomes as inputs and generate the new clusters of the CC/G. The approach uses two main phases: sub-cluster generation and merging. First phase employs an agreement strategy between the multiple clustering algorithms to find the set of intersections between the different clusterings. Extracted sub-clusters are then represented by a vector of agreement memberships. In the merging phase, sub-clusters are merged based on cohesiveness and coupling between elements. The merging process converges when the desired number of clusters is obtained.

3.4.1 Agreement Phase

An agreement strategy is employed to discover common patterns between different clusterings and to produce a set of agreement sub-clusters. In the original version

of the cooperative approach, the authors assume that each clustering techniques produces the same number of clusters (i.e. k_i is equal) [28]. They generate a number of sub-clusters as k_i^C , where k_i is the number of clusters generated by a partitioning algorithm i, and C is the number of partitioning techniques. However, due to the nature of the software clustering domain, software systems may have different authoritative decompositions [55]. So, we generalized sub-cluster generation by allowing unequal number of clusters and generate sub-clusters based on k_{max} , where $k_{max} = max(k_i), i = 1, 2, .., C$.

To find associations between the corresponding sets of sub-clusters, a membership vector $ComemVector$ of size n, where n is number of software entities, generated based on calculating a sub-cluster membership value assigned to each software entity (in our case a Java class). This clusterings-mapping recognizes the set of disjoint sub-clusters generated by the intersection of the c partitioning techniques. The new cooperative sub-cluster membership for a given software entity x is defined as:

$$mem(x) = mem(x)_{P_1} + ... + mem(x)_{P_{(C)}} * k_{\max}^{C-1} \qquad i = 1, ..., C \tag{3.4.1}$$

Where $mem(x)_{Pi}$ is a cluster label for each software entity assigned by a partitioning algorithm P_i, The membership function satisfies the following condition: for any two Java classes x and y, if $mem(x) = mem(y)$, then x and y belong to the same cluster (or sub-cluster). Consequently, ComemVector is defined as:

$$ComemVector(i) = mem(x_i), \qquad i = 1, ..., n \tag{3.4.2}$$

Using k_{max} for generating the sub-clusters memberships causes an enormous growth in the number of generated sub-clusters especially for large values of k_{max}. Furthermore, the cooperative membership vector most probably has sparse entries. Therefore, only non-zero values of ComemVector are only passed into the next phase of CC/G. Future directions include the investigation of finding better membership function for variable number of clusters.

3.4.2 Merging Phase

The intuition behind this phase is attained the clustering solution through merging of generated sub-clusters. The most suitable sub-clusters for merging are those that obtain maximum intra-connectivity among their own cluster elements and minimum inter-connectivity with respect to other clusters.

Equation 3.4.3 is used to merge sub-clusters. This novel equation measures the benefit of merging two sub-clusters based on the value of the merging factor (mf). The merging factor can be calculated as follows:

$$mf = \frac{\mu_i + \mu_j + \varepsilon_{ij}}{N_i N_j (N_i N_j - 1)} \tag{3.4.3}$$

Where μ_i and μ_j are the intra-connection between elements of sub-clusters i and j, respectively. μ_i is the sum of all relationships that exist between classes in sub-cluster i. A higher value of intra-connectivity is suggestive to high cohesion. $\varepsilon_{i,j}$is defined as the sum of all relationships that exist between classes in two distinct sub-clusters i and j. $\varepsilon_{i,j}$ has values between 0 and 1. The 0 value mean, no subsystem level relations between subsystem i and subsystem j exist and value 1 means all modules in subsystem i are related to all modules in subsystem j and vice-verse. A low value for inter-connectivity means low coupling. N_i and N_j are the number of entities in each sub-cluster i, j respectively. The proposed merging factor is used as measure of the cohesiveness of merging two clusters. Thus, the two sub-clusters that have maximum mf are merged into a new cluster. Then the sub-clusters list is updated based on the new added cluster. This step is repeated until no additional merging is needed when reaching the desired number of clusters.

3.5 CC/G Complexity

Assume we are using c clustering algorithms in our cooperative approach, and the DSM is already extracted. Thus the computational time of these algorithms will be bounded by the algorithm of highest complexity.

The computational overhead of the cooperative aggregation and merging can be divided in two parts, 1) Sub-clusters generations complexity, and 2) Sub-clusters merging complexity. In order to generate the common sub-clusters, a linear membership assignment is needed which is $O(n)$, where n is the number of classes in the software system.

Merging of sub-clusters in an iterative scenario takes $\mathrm{O}(n_{sb}^2)$, where n_{sb} is the number of sub-clusters, n_{sb} is greater than or equal to k. The complexity and extra computation cost of this merging process should be weighed against the choice of the merging function at each iteration. For example for CC/G-min, choosing the least coherent sub-cluster takes $O(1)$ and then find the pair sub-cluster for merging may take $\mathrm{O}(n_{sb})$ in its worst case scenario. Updating the cohesion and coupling after each merging step takes $O(n)$.

If fully agreement is obtained such that $n_{sb} = k$, then the CC/G takes a linear time complexity, where $k^2 <<<< n$. However, for a complete disagreement, $n_{sb} = n$, thus CC/G takes in its worst case scenario $\mathrm{O}(n^2)$.

3.6 Limitations of CC/G

We don't expect that any real large software system is composed of clusterings that are identical or totally dissimilar with different algorithms. A challenge is to

determine in what situations a cooperative clustering is appropriate. Intuitively, there are three scenarios for agreements between software engineering clusterings: full, non and partially agreements. For full-agreement ensemble scenario, the cooperative clustering algorithm will add no additional information, and the clustering solution will be identical to any of its constituents. For non-agreement ensemble scenario, each sub-cluster becomes a singleton element by itself and the total complexity will be $O(n^2)$. In general, the cooperative clustering could improve upon its constituent inputs if there exist a partially agreement between them. This can be done by employing a proper merging scheme that may yield improvements in clusterings. How much agreement should exist to provide effective cooperative clustering is an open research question?

3.7 Clustering Evaluation

Evaluation is an essential part of the proposed algorithm. Thi s step decides "how good the clustering solution is". We compare the clustering solutions obtained by the input constituents to those generated by CC/G. The comparison is based on three evaluation measures: TurboMQ, MoJo Distance, and Stability. Each of these measures were previously described in detail in Chapter 2.

Turbo Modularity Quality ($TurboMQ$) is used by [45] as a partitioning quality function and it is used in this research as an evaluation measure as it has been applied successfully on hill-climbing algorithms to measure the quality of generated partitions as discussed in [37].

In our work, we also use another measure, MoJo distance to measure the similarity of the CC/G results compared to that of the various software clustering algorithms. Moreover, it help us to determine if the generated CC/G clusters are biased to the clusterings generated by the input constituents or to the original partitioning of the benchmark . The CC/G clusterings should not be biased to the original partitioning nor to the clusterings enrolled to generate the CC/G partitions. In case CC/G generates clusters that are biased to the original partitioning, this means CC/G failed to provide a different solution to the original system partitioning (which may not be the desired decision in some scenarios). In this case, CC/G succeeded only in distinguish between the clustering algorithms engaged to generate CC/G clusters.

The Stability [62] is also used in our evaluation step to measure the percent change between decompositions of successive versions of an evolving software system. Under conditions of small changes between consecutive versions, an algorithm should produce similar clustering.

3.8 Summary

In this chapter, a new cooperative software clustering algorithm, CC/G presented and discussed. The proposed algorithm contains four main steps, starting by extracting the dependency structure, applying individual partitioning of the DSM,

employing the cooperative partitioning; merging of the common sub-clusters, and finally evaluating the final clustering solutions. Experimental analysis and results over a number of software systems are illustrated and discussed in the following chapter, showing the effectiveness of the CC/G algorithm compared to the leading algorithms in software clustering.

Chapter 4

Experimental Work

This chapter presents the results from a number of experiments showing the effectivness of the proposed cooperative methodology and approach compared to the individual clustering techniques in terms of clustering quality and stability. Following Mitchell [45] and Tzerpos [59, 61], we consider the following types of dependencies in our experiments: function calls, variable reference or directory structure of source code files.

4.1 Experimental Procedure

In this section , we outline the necessary steps and procedure of preparing, partitioning and evaluating the benchmark software programs as follows:

1. Obtain the Benchmark Programs in a JAR file format by downloading them from [64] .
2. Load system JAR file into Lattix [52].
3. Extract the dependencies, export its structure into Excel [41] or flat file and store the file as original system partitioning.
4. Apply Lattix partitioning algorithm on extracted dependencies, then store the generated partitioning as a Lattix system decomposition format.
5. Begin labeling the original and Lattix partitioning (manually) in the stored files.
6. Convert both original and Lattix system partitioning into the proper format (modular dependency graph 'MDG') for the Bunch tool [37].
7. Run the Bunch tool [37] against the extracted data by applying hill climbing algorithms to system dependencies. The files produced can be easily imported into MoJo tool.
8. Import the generated hill climbing system partitionings into Matlab [40].

9. Use the clusterings of original, Lattix and hill climbing algorithms to perform cooperative clustering by first finding agreement between constituent inputs then merge the sub-clusters back into the required number of cluster.

10. Use the normal MoJo metrics version 1.2 [60] to find the similarity between the constituent inputs and the produced cooperative clusterings.

11. Use the modularization quality [37]to measure the quality of the generated clusters for every solution.

Gathering the benchmark programs, the use of the constituents inputs , choosing an efficient merging criteria, and evaluating the quality of the generated clusterings are discussed and analyzed in the following sub-sections.

4.2 Benchmark Programs

In this section, we analyze six open source Java projects selected from a list of the Helix benchmark programs. The Helix benchmark suite is a compilation of release histories of a number of non-trivial Java Open Source Software projects. This suite is discussed in Vasa's PhD thesis [63].

The first and second sets of software systems used in our experiments are produced in a variety of sizes, functionalities, and development philosophies, as listed in Tables 4.1 and 4.2, respectively. In particular, the first and second columns in Tables 4.1 and 4.2 show the name of the system and its version. The third, fourth, and fifth columns indicate the number of classes, interfaces, and methods in each program graph, respectively. Program size listed in the last column. It should be noted that Table II provides information about a later version of each software system listed in Table I.

Name	Version	Classes	Interfaces	Methods	Size-In-Bytes
Acegi	0.5.0	106	29	94	283955
Cocoon	1.7	84	21	72	224593
JabRef	1.1	103	5	40	348449
Proguard	1.0	90	22	127	275918
Struts	0.5	102	4	3	287969
Xwork	1.0	74	26	45	252546

Table 4.1: Benchmark Programs (First Set)

Our first step was acquiring all necessary data for the initial list of selected projects, including executable versions (byte code) as well as copies of the version archives. The software systems we used in our experiments came in a variety of sizes, functionality and development philosophies. Those systems are listed as follows:

Name	Version	Classes	Interfaces	Methods	Size-In-Bytes
Acegi	0.5.1	113	30	103	299708
Cocoon	1.8	90	23	73	340237
JabRef	1.2	150	8	47	517042
Proguard	1.1	95	23	129	282309
Struts	1.0	183	2	20	587258
Xwork	1.1	105	31	61	407835

Table 4.2: Benchmark Programs (Second Set)

1. Acegi Security Framework [1]: a highly popular enterprise security framework for web applications. It provides many authentication and authorization features.

2. Cocoon (Apache) [2]: it is a web application framework built around the concepts of pipeline, separation of concerns and component-based web development. The framework focuses on XML and XSLT publishing.

3. JabRef [3]: it is a reference management software that uses BibTEX as its native format. JabRef provides an easy-to-use interface for editing BibTEX files, for importing data from online scientific databases, and for managing and searching BibTEX files.

4. Proguard [4]: it is free Java classes file optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes, and it optimizes byte code and removes unused instructions.

5. Struts (Apache) [5]: it is an open-source web application framework for developing Java EE web applications. It uses and extends the Java Servlet API to encourage developers to adopt a model–view–controller (MVC) architecture.

6. Xwork [6]: it is a generic command pattern implementation with no dependencies on web specific libraries.

In particular, the first and second columns in Table 4.1 show the name of the experimented system and its version. The third, fourth, fifth and the sixth columns show the numbers of classes, interfaces, methods and fields in each program graph, respectively. The programs sizes are shown in the last column. Table 4.2 presents a later version from every software system in Table 4.1. Software systems in Table 4.2 are used in studying the stability of the CC/G algorithm.

4.3 Constituent Clustering Techniques

Lattix [52] and Bunch [37] software tools used in our experiments. Lattix is a suite of software that is used to analyze software system architecture in detail, to edit the structure to improve the design, and to specify rules to formalize and communicate

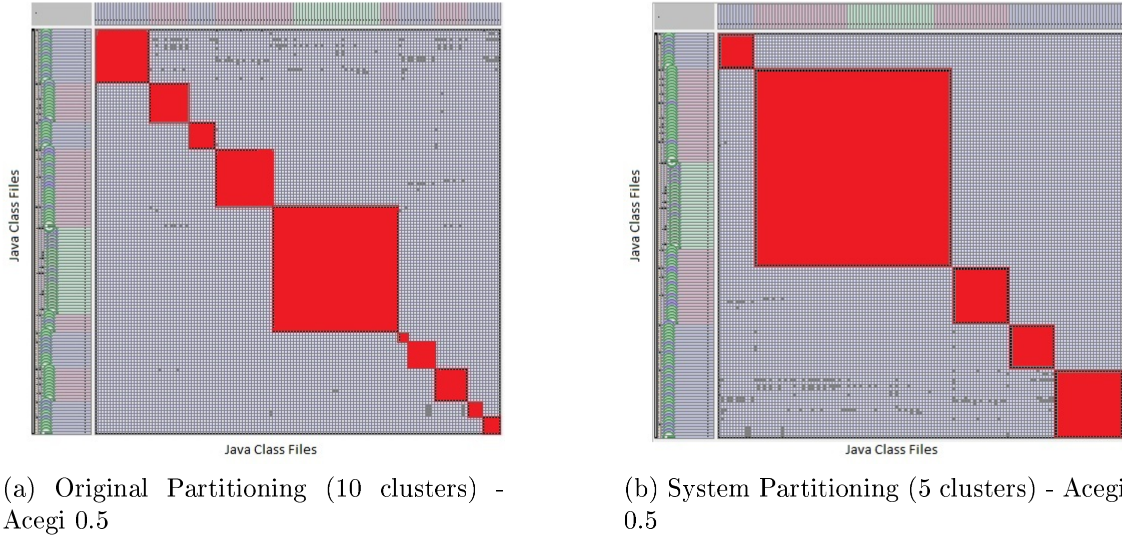
the architecture to the entire organization. Lattix uses the Dependency Structure Matrix (DSM) to show the structure among software entities. The DSM approach uses a matrix of rows and columns to show how each software artifacts depend on each other within a project.

Lattix has two interesting feature to us, the inclusion of the DSM extractor for Java source code and the embedded algorithm for partitioning a software structure. For dependency extraction, Lattix uses a standard notion of dependency, in which a module A depends on a module B if there are explicit references in A to syntactic elements of B. For DSM partitioning, Lattix uses Sangal approach [52], in which rows are re-arranged to form a triangular structure. This structure helps architects to determine whether or not specified architectural styles, such as layers, are implemented correctly. We used this partitioning algorithm on all the DSMs that were produced by the Lattix. Figure 4.1a shows the original DSM for the Acegi software system. Figure 4.1b shows the DSM after the partitioning algorithm applied.

Bunch is a suite of algorithms that attempt to find a decomposition that optimizes a quality measure based on high-cohesion and low-coupling. In particular, Bunch uses randomization in its optimization approach to form clusters, therefore, it is unlikely repeated runs will produce the exact same decomposition of a software system. Bunch has three algorithms for partitioning, the hill-climbing, the genetic algorithms and the exhaustive algorithms, each with different configurations. Bunch begins its partitioning process by creating a random partition of the structure Module Dependency Graph (MDG). In our experiment, we used the unweighted version of the MDG. Although, Bunch uses graph diagram for representing software structure which may be not efficient for complex system but it has several unique features that lend themselves well to our study. We used Bunch version 3.3.6 and we established our experiments with two versions of a hill-climbing algorithm, we will refer to as NAHC (Nearest Ascent Hill Climbing) and SAHC (Steepest Ascend Hill Climbing) [37].

4.4 Choosing an Efficient Merging Criterion

Two different merging criteria have been examined for selecting two sub-clusters i and j to be merged. We will refer to the two merging criteria as CC/G-min and CC/G-max. The first criterion CC/G-max is based on selecting the sub-cluster i with the maximum cohesion (i.e. maximum μ_i), then we choose the sub-cluster j such that the ratio mf (defined in equation 3.4.3) is maximum for $i, j = \{1, \ldots, k^c_{max}\}$, $i \neq j$. This selection procedure is repeated iteratively until the desired number of compositions is obtained. However, this technique was not successful for some of the benchmark systems. We assume that this criterion did not perform as expected for all the systems as enriching a sub-cluster with maximum dependencies between its objects with another sub-cluster with low cohesion may deform the total number of dependencies in each of the two sub-clusters if they become one cluster.



(a) Original Partitioning (10 clusters) - Acegi 0.5

(b) System Partitioning (5 clusters) - Acegi 0.5

Figure 4.1: DSM Partitioning Example

Another issue that affected the results is the existence of singleton sub-clusters (i.e. sub-clusters with only one object) as they have high cohesion $\mu_i = 1$, these set of singleton sub-clusters bias the merging selection to combine the small sub-clusters first.

We have overcome this drawback by first selecting the sub-cluster with minimum cohesion (i.e. lowest number of connection between its elements), this method of selection for merging is called CC/G-min. The main reason for this selection is to enhance the cohesion of this sub-cluster through merging, next we choose the second sub-cluster also such that the ratio mf is maximum. This selection criterion allows us to generate the final set of clusters with the highest cohesion and lowest coupling among each other. This technique provides an enhanced solution to generating optimal solution for all benchmark systems.

Figure 4.2 shows how these criteria perform for benchmark systems. The horizontal axis represents the number of clusters for each benchmark system. The vertical axis is for the $TurboMQ$ values. Each curve represent the $TurboMQ$ values of both examined merging criteria. The red curve correspond to the criterion of selecting the sub-clusters of maximum cohesion, while the blue curve represent the opposite criterion. It can be shown from figure 4.2 that the CC/G-min and CC/G-max have almost the same $TurboMQ$ (TMQ for simplicity) value for Acegi and Xwork systems, but it is clear that for the Proguard , the CC/G-min has higher values of the $TurboMQ$ than the CC/G-max for number of clusters $= 3, 4, 5, .., 10$.

4.5 Clustering Evaluation

To evaluate the quality of the software partitioning algorithms, we adopted various criteria including the quality of generated cluster, distance (or similarity) between the decompositions, and stability of solutions. Three experiments are presented in this section to cover each of those criteria. The target of the first experiment is to compare the values of *TurboMQ* of the clusterings solutions generated by the proposed cooperative approach and by the individual partitioning algorithms for the same software system. Next experiment aims at measuring the similarity between different partitioning approaches using MoJo distance, finally the stability of the clustering solution obtained by the CC/G is presented and discussed in the third experiment.

4.5.1 Modularization Quality

The values of the *TurboMQ* of different decomposition of each test system for the CC/G, Lattix, NAHC ,SAHC, and original system partitioning are listed in tables 4.3 to 4.8. The "original" in each of the following tables and figures refers to the package structure of the benchmark systems. We consider the package structure to be flat partitioning.

This improvement in the quality of clusterings through CC/G is due to the choosing of proper merging mechanism, namely CC/G-min. The CC/G-min generates optimum exterior and interior connectivity between cluster elements and remove weak sub-clusters. In tables 4.3 to 4.8, the symbol "-" at a specific number of clusters k means that the corresponding partitioning algorithms do not generate k clusters.

Table 4.3: TurboMQ Values for Acegi 0.5.0

	k=5	k=8	k=10
Original	-	-	6.259
Lattix	3.1616	-	-
NAHC	3.8859	-	-
SAHC	-	6.2504	-
CC/G	4.4203	6.6756	8.5162

Table 4.9 and figure 4.3 show the percentage of improvement in the *TurboMQ* using the CC/G compared to the leading approaches for the benchmark systems. The CC/G achieves an improvement in the *TurboMQ* of up to 53% compared to that of the original decomposition for xWork 1.0 and up to 51% compared to the Lattix for Proguard 1.0. In addition it obtains an enhancement in the *TurboMQ*

Table 4.4: TurboMQ Values for Cocoon 1.7

	k = 5	k = 7	k = 10	k = 12	k = 18	k = 22
Original	-	5.497	-	9.401	-	-
Lattix	3.357	-	-	-	-	-
NAHC	-	-	-	-	11.7726	-
SAHC	-	-	7.830	-	-	14.471
CCG	3.839	5.839	8.687	10.013	16.013	19.760

Table 4.5: TurboMQ Values for Jabref 1.1

	k = 6	k = 7	k = 15	k = 17
Original Partitioning	4.972	-	-	-
Lattix	3.139	-	-	-
NAHC	3.642	-	9.123	-
SAHC	-	4.716	-	10.555
CCG	4.997	5.614	11.934	12.919

value by 16% compared to the NAHC algorithm for the Acegi 0.5. Furthermore, CC/G reaches up to 59% compared to that of the SAHC decomposition for Struts0.5 and up to 38% compared to the Lattix for Jabref 1.1.

Finally, we can also see that the CC/G outperforms the SAHC by an increase in the *TurboMQ* of up to 42% for the Proguard 1.0. In summary, we can see that the CC/G yields better results for all systems with an improvement of 27% on average in the TurboMQ values for the used benchmarks systems .

4.5.2 Similarity Measures

Generally, a similarity measure gives us the idea of how good (effective) the resulting decomposition is. This can be done by comparing the decomposition produced by the clustering algorithm against the benchmark or expert decomposition. In this thesis to achieve this goal, we use MoJo[60] distance metric for evaluation, which is also commonly used by the work in [67] and[12]. The similarity between two partitionings solution, C_i and C_R is defined as follows:

$$mSim(C_i, C_R) = \left(1 - \frac{MoJo(C_i, C_R)}{n}\right) \times 100\% \qquad (4.5.1)$$

Where C_R is the assumed reference decomposition, C_i is the partitioning produced by the algorithm i and n is the number of software entities to be clustered. Intuitively, the higher the $mSim(C_i, C_R)$, the closer clustering results to the reference decomposition. As, we have no authoritative decomposition (reference) to the software systems that we cluster, we build the following results based on the assumption that the original system partitioning is a reference decomposing. Hence,

Table 4.6: TurboMQ Values for Proguard 1.0

	k = 3	k = 4	k = 5	k = 9
Original	-	2.1165	-	-
Lattix	1.3049	-	-	-
NAHC	-	-	-	5.4928
SAHC	-	-	2.3465	-
CCG	1.9734	2.6231	3.3215	6.3944

Table 4.7: TurboMQ Values for Struts 0.5

	k = 4	k = 6	k = 8
Original	2.798	-	-
Lattix	1.852	-	-
NAHC	-	2.776	-
SAHC	-	-	2.918
CCG	3.167	5.275	7.114

we examine how close the different system decompositions to the original decomposition. Our aim is to prove that the CC/G will not be biased to either the original system decomposition nor to the other partitionings.

Tables 4.10 and 4.11 present the similarity (*mSim*) results for two consequence versions of programs under study. The *mSim* values are percentage values. The larger the value of *mSim*, the closer decomposition to its original. In table 4.10, we have noticed that the CC/G partitioning is not close to the original partitioning as compared to Lattix for Acegi 0.5, Proguard 1.0, and Xwork 1.0; nor far as compared to NAHC and SAHC for Acegi 0.5 and Xwork 1.0.

For the Proguard 1.0 and Struts 0.5 systems in table 4.10, the *mSim* value of CC/G is much closer to that of the SAHC. This indicates that the SAHC decomposition of these two benchmarks datasets is the closest to the solution generated by the CC/G from the family of solution provided. So, both decompositions are accepted as a partitioning solution for both benchmark systems using the *mSim* formulations.

In 4.11, CC/G is able to find better solution for Acegi 0.5.1, Cocoon 1.8, Jabref 1.2, Proguard 1.1, Xwork 1.1 compared to Lattix, NAHC, and SAHC. Furthermore, Struts 1.0 system, the CC/G and SAHC generate the same partitioning as compared to the orignal decomposition.

In conclusion, the CC/G produces unbiased results for four out of six benchmark systems in table 4.10 and five out of six benchmark systems in table 4.11. These unbiased results are augmented with the enhancement of quality as shown in section 4.5.2, showing the major advantages of using CC/G as a consensus cooperative clustering approach. The results of these experiment indicate that the idea of

Table 4.8: TurboMQ Values for xwork 1.0

	k = 2	k = 5
Original	-	2.6721
Lattix	1.0341	-
NAHC	-	4.0507
SAHC	-	3.789
CCG	1.5282	4.3114

Table 4.9: Improvement (%) in *TurboMQ* by CC/G

	Acegi 0.5	Cocoon 1.7	Jabref 1.1	Proguard 1.0	Struts 0.50	Xwork 1.0
Original	36	6	0.5	24	12	53
Lattix	43	13	38	51	42	48
NAHC	16	27	25	11	47	1
SAHC	13	18	17	42	59	8

using common patterns as a basis for software clustering has a distinct advantage, as it is clear that it can create decomposition that are not biased to the original decomposition nor to the used ones.

4.5.3 Stability of Clustering Solutions

The stability of a clustering solution concerns the persistence of the partitioning structure of two consecutive versions of an evolving software system [62], [50]. Previous studies by [62], [50] used Mojo to evaluate software clustering algorithms' stability. These studies, as well as our research study, focus on stability in the context of small incremental changes to the program. Therefore, stability is considered to be inversely proportional to the MoJo metric: a small MoJo value indicates a stable clustering algorithm, whereas a large MoJo value designates an unstable clustering algorithm.

Given a clustering algorithm i and a program x, if there is a minor change in the program x and a correspondingly small change in the output of the clustering algorithm i, then i is considered a stable clustering algorithm. In our preliminary studies we have used a modified version of the MoJo metric which focuses on the common elements that exist in both versions of the program. We call our new metric the "Stability Ratio (SR)". We define the SR based on the ratio of the number of movements (split or join) between clusters to the total number of shared classes occurred between successive versions of an evolving software system.

The goal of the following experiment is to show that the proposed CC/G approach has a significant SR with respect to the variation in a program.

Table 4.12 shows the change that occurred between two consecutive versions of benchmark programs in terms of added classes, removed classes, and change in number of dependencies. Each benchmark program versions have different changes in various aspects (as listed in Table 4.12). Some software have a major modification

Table 4.10: Similarity values (%) obtained by the NAHC, SAHC, Lattix, and CC/G with respect to the original partitioning (First set of software programs)

	NAHC	SAHC	Lattix	CC/G
Acegi 0.5	37.78	37.04	77.78	47.65
Cocoon 1.7	73.27	45.045	44.55	60.73
Jabref 1.1	62.26	41.035	34.435	57.31
Proguard 1.0	67.52	70.09	99.91	74.15
Struts 0.5	80.19	66.98	51.89	64.78
Xwork 1.0	39	35	97	65.5

Table 4.11: Similarity values (%) obtained by the NAHC, SAHC, Lattix, and CC/G with respect to the original partitioning (Second set of software programs)

	NAHC	SAHC	Lattix	CC/G
Acegi 0.5.1	55.92	44.63	96.69	67.11
Cocoon 1.8	50.00	53.08	100	61.49
Jabref 1.2	55.92	44.63	96.69	67.11
Proguard 1.1	45.61	47.37	82.46	64.62
Struts 1.0	48.46	54.8	74.23	56.69
Xwork 1.1	64.38	62.75	82.46	75.88

in dependencies between its classes, and others have a significant change in addition or deletion of its classes.

Table 4.12: Changes in consecutive versions for each benchmark programs

Program	Used Versions		Added Classes	Removed Classes	Change in dependencies
acegi	0.5.0	0.5.1	9	1	30
proguard	1.7	1.8	1	4	19
xwork	1.1	1.2	3	1	35
Struts	1.0	1.1	50	16	230
Cocoon	0.5	1.0	20	12	6
Jabref	1.0	1.1	16	3	93

Figure 4.4 illustrates the similarity ratio of CC/G clusterings values among two consecutive versions of benchmark programs. For any two consecutive versions of the given benchmark programs (x-axis), the y-axis represents the number of elements moved between programs clusters of CC/G over the total number of shared elements in these clusters as measured by the SR.

From Figure 4.4, we can see small changes in the successive versions of versions of Acegi, Proguard, and Xwork programs. CC/G obtains nearby values of the SR, showing the stability of clustering solutions obtained by the cooperative approach.

For the Jabref program, the CC/G SR is high because there is a significant large change among its two consecutive versions.

For the Struts program, the change in dependencies is slightly higher than the remaining benchmark programs, however the number of movements between its used consecutive versions are close to the Acegi, Proguard, Xwork programs. Furthermore for the Cocoon program, the changes in number of classes and dependencies are slightly low but the SR of the CC/G is is quite large compared that of other benchmark prgrams. This discrepancy seems rather strange at first glance since the changes in Struts and Cocoon do not reflect correctly the system structure obtained by the CC/G. We believe that this behavior is probably caused by the distribution of dependencies in those programs abstraction. However, a closer investigation should be established to reveal this behavior under various testing of other stability measures in addition to the stability ratio measure.

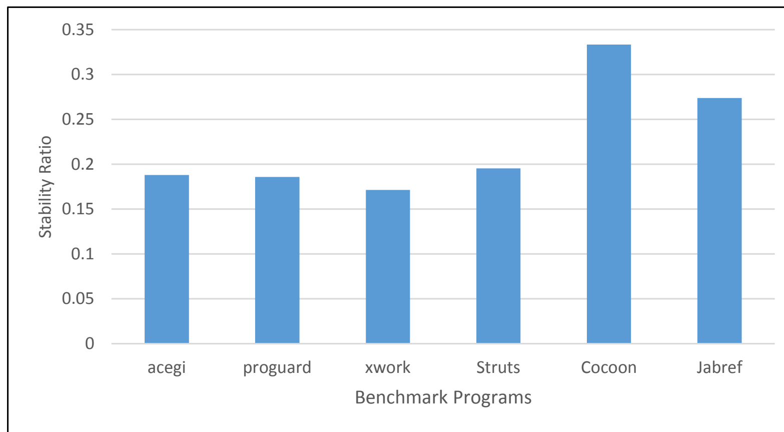


Figure 4.4: Distance between two consequence version of benchmark programs

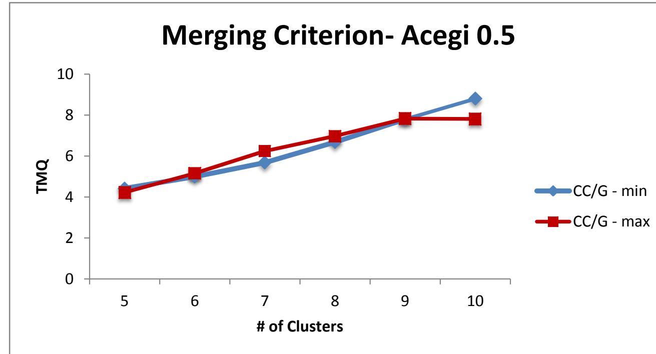
Future extension to the experimental work on stability testing can be established by

1. Using more consecutive versions of the benchmark programs on a daily basis, such that we can combine the clusterings of the CC/G on a former day to decompositions of input constituent on a latter day, this incremental addition of the CC/G decisions can be repeated for a predefined number of days, and a the stability of the final CC/G clusterings is then tested;
2. Comparing the stability ratio obtained by the CC/G to that of individual constituent for the same benchmarks programs.

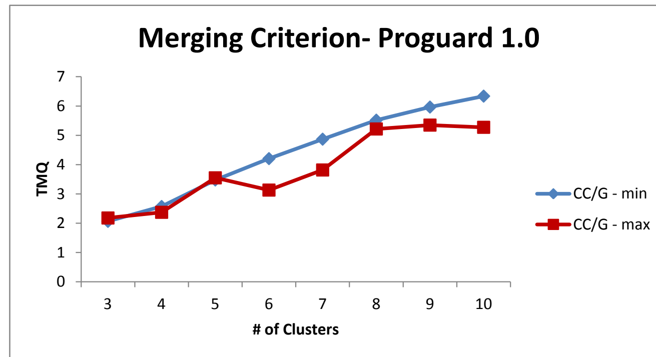
4.6 Summary

In this chapter, various experiments have been conducted over a number of software benchmarks data sets. Experimental results using different validation measures yield that the proposed CC/G algorithm outperforms the individual software

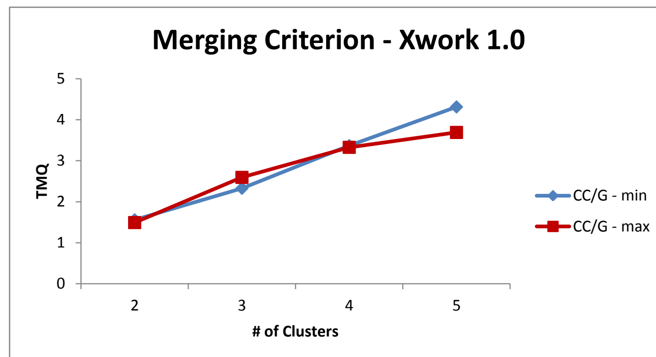
clustering algorithms. The clustering solutions obtained by the CC/G have high cohesion and low coupling measured by the TurboMQ quality measures showing the efficiency of the CC/G algorithm compared to the input constituents. Using the MoJo distance measure, the CC/G clusterings are shown to be unbiased to the original partitioning nor to the decompositions obtained by any of the input algorithms. The experimental work also illustrates that the CC/G is a well defined stable software clustering algorithm.



(a) Acegi 0.5.0

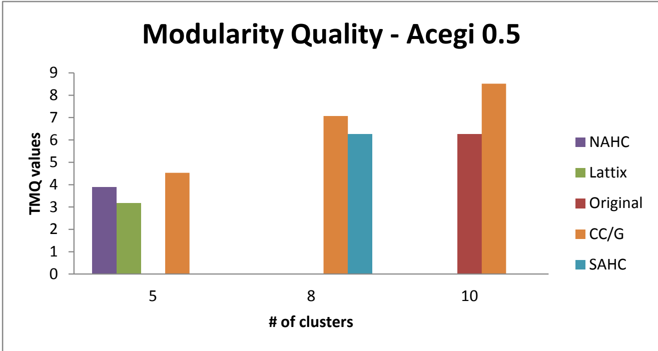


(b) Proguard 1.0

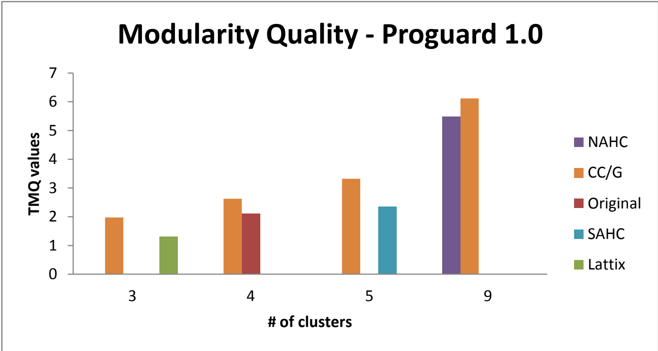


(c) Xwork 1.0

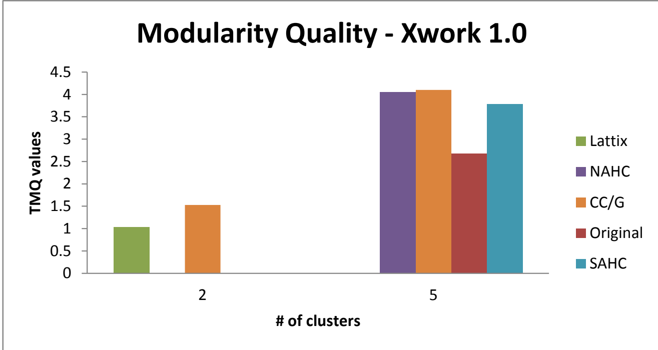
Figure 4.2: Comparison between CC/G-min and CC/G-max for Acegi0.5, Proguard 1.0, Xwork1.0 system



(a) Improvement in TurboMQ Values for Acegi 0.5.0



(b) Improvement in TurboMQ Values for Proguard 1.0



(c) Improvement in TurboMQ Values for Xwork 1.0

Figure 4.3: Quality of partitioning

Chapter 5

Summary and Future Work

The size and complexity of modern software systems necessitate suitable abstractions of their structure in order to make them more understandable and thus easier to maintain. Software clustering techniques are effective for creating such abstractions because they can generate architectural-level views of a system's structure directly from its source code. Software clustering involves the partitioning of software system components into clusters so that optimal exterior and interior connectivity among system components is achieved. However, a major shortcoming of the current practice is that existing techniques are often either reinvented or require augmentation. The goal of this research is to advance the state of the art in software clustering through the exploration of potential research solutions that will overcome this drawback. This chapter provides a summary of the work that has been achieved thus far, along with a description of the planned continuation of the research.

5.1 Research Summary

The core of this research is the proposal of a new software clustering approach (CC/G) that builds on the notion of cooperative clustering (CC), as demonstrated for document clustering by Kashef and Kamel [28]. The CC approach combines a variety of software clustering approaches in order to provide measurably better clustering compared with that produced by any single individual approach. Cooperative clustering is divided into two phases: a sub-cluster generation phase and a merging phase. In practice, the implementation of the concept of cooperative partitioning for systems software has required modification. The original CC approach [28] is based on the assumption that an identical number of clusters is produced by each of the input algorithms. This assumption does not hold true for current state-of-the-art approaches to software clustering, and it is also based on vectors of quantifiable features. Software architectures, on the other hand, are constructed of structures and dependencies, represented in graphs that describe the relationships of the various entities in the software. The proposed method overcomes the disadvantage of the original CC approach by introducing a new membership function

for generating sub-clusters as well as a new merging condition appropriate for the software paradigm. The CC/G algorithm has been tested with a variety of merging criteria, a process that has led to the conclusion that merging low-cohesion clusters creates better clustering results, as assessed according to Michell's TurboMQ [43]. Experiments using real benchmark programs have shown that the proposed CC/G approach works effectively and efficiently so that practitioners can use the new approach for partitioning software and revealing the natural, inherited data structure.

5.2 Current Research Achievements

The work that has already been completed represents five principal research contributions:

- The introduction of a novel formulation of the software clustering problem using a cooperative strategy derived from document clustering, an approach that, to the best of my knowledge, has not previously been reported in the software engineering literature.
- The ability to customize the software clustering platform by extending cooperative clustering to function with a variable number of clusters.
- The accommodation of software dependencies when intermediate clusters are merged.
- The introduction of novel merging criteria based on cohesion and coupling.
- The use of a variety of quality and stability measures for evaluating the efficacy of the partitions obtained compared to state-of-art software clustering algorithms, with results that demonstrate that cooperative clustering is more efficient and effective than dependency structure matrix (DSM) partitioning or hill-climbing clustering algorithms.

5.3 Future Work: Enhancing Bug Classification using CC/G

5.3.1 Introduction

Many software projects depend on bug reports through its maintenance activity [?]. In open source software projects, bug reports are regularly submitted by software contributors and collected in a database by a bug tracking tool such as Bugzilla. Storing bug reports in a repository allows contributors to report and potentially help fixing bugs to improve software quality [?]. In Bugzilla, the number of defect reports usually exceeds the available development resources and becomes a laborious intensive task. For example, in 2005, one Mozilla's developer claimed that,

"everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [?]. That large number of defect reports is usually due to the large number of contributors discovering the same defects in the same time and, consequently, report them. Previous studies report that as many as 36% of bug reports were duplicate reports [?]. Thus, it is a challenging problem to examine all existing defect reports to detect duplication.

Bug classification is one of the proposed techniques to detect duplicated defect reports in [?], [?], and [?]. A classifier is usually able to combine different types of defect report features to identify duplicates. This approach can serve as a filter between contributors and arriving defect reports. However, bug classification typically constructed using a large number of defect reports stored in a bug tracking system. This is also a challenging task due to the training complexity, high memory requirements and slow convergence. Therefore before building a classifier, a cluster analysis on its training data is needed. This stage will find the optimal number of clusters that are needed for training the classifier. Also, a cluster analysis for inputs can help on finding groups of homogeneous bug reports features that may deliver additional accuracy for classification. Moreover, It will help in discovering the bug reports features that may have a negative impact on the accuracy of the models.

5.3.2 Clustering Defect Reports using CC/G

Bug reports contain free-form textual descriptions and titles, and most duplicate bug reports share many of the same words. Bug reports titles and descriptions can be used to define a textual distance metrics. This metrics will be used to identify the duplication between bug reports. A typical approach for building a textual similarity metric is using the "bag of words" approach [?].

Each bug report is represented by a vector v of size n, with $v[i]$ related to the total number of times that word i occurs in that bug report textual information. The precise value at position $v[i]$ is found from a formula that can have the number of times word i appears in that bug report, the number of times it appears in the bug report corpus, the length of the bug report textual information, and the size of the bug report corpus.

When we got the set of vectors representing all the bug reports corpus, we can compute the similarity between them by using cosine similarity [28]. Cosine similarity is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them. The closer two vectors are to co-linear, the more the corresponds between two bug reports.

Finally, the generated textual similarity metric is used to induce a graph in which the nodes are defect reports and edges link reports with similar text. Thus, a clustering algorithm is applied to partition this graph to obtain a set of clustered reports. The clustering algorithm will tease out the structure between defect reports and will illustrate the duplication between them. Every cluster exposed is showing the duplication of a bug report in the corpus. Furthermore, if the incoming bug report was not included in any cluster. Intuitively, this report is a singleton. Thus,

it is less likely to be a duplicate.

The main research question we investigate in this work is whether the CC/G, can improve duplicate bug reports classification results. Furthermore and according to our knowledge, no work has been done for enhance duplicate bug reports classification using consensus cooperative partitionings approaches (CC/G).

5.3.3 Experimental Setup

The experiments for this work will be based on the bug reports from the Mozilla project [?]. The reports will span from January 2009 to May 2012. The generated data set will include reports from four programs of them: Firefox, Thunderbird, Eclipse and NetBeans. For our study, we are going to retrieve a punch of the full history of all bug reports via the Bugzilla API of the respective projects. In particular, our analysis will focus on a subset of those bug reports that had a final status indicating that they were resolved. We limit our analysis to these bug reports because the bug handling community already completed the categorization process and thus reached a decision on how they were processed. Those chunk of reports will be used in training the classifier after applying CC/G to find reports without duplication.

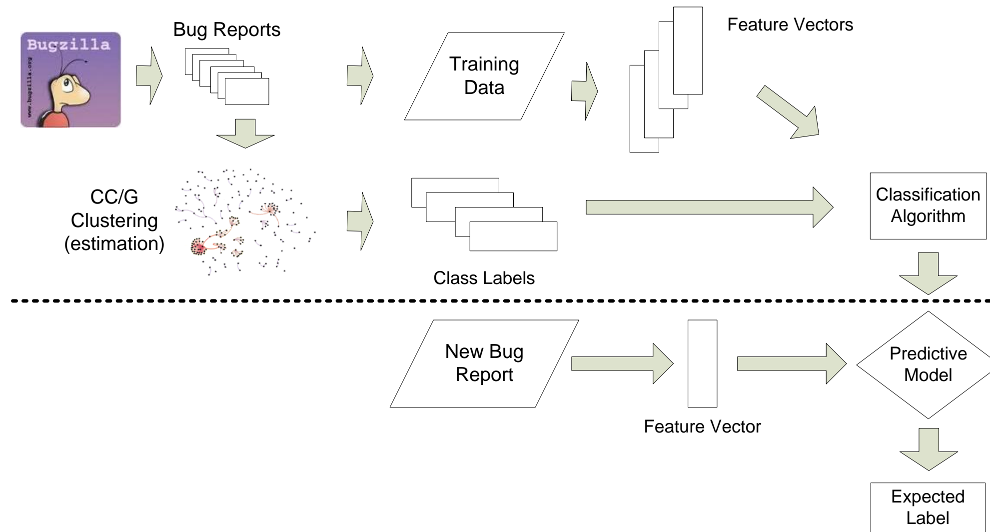


Figure 5.1: Building a Bug Report Predication Model

Figure 5.1 shows the flow diagram of the intended building system. This system will be dividing into two main parts: training and testing. Training will include the extraction of bug reports, extracting textural features from those reports, dividing the data set into 25% reports for training and leave the remaining to the testing part, applying CC/G, then building the classifier. For testing, testing data set will be provided into the classifier to test its prediction and performance.

5.3.4 Conclusion

We propose a software engineering experiment in which the CC/G is supposed to enhance the automatic classification of duplicate bug reports as they arrive to save developer time. We empirically evaluated CC/G using the available data set of bug reports from the Mozilla project. We believe that CC/G is able to enhance the classification of duplicate bug reports.

Bibliography

[1] Acegi – role based security framework, 2010. 25

[2] Cocoon – web application framework, 2010. 25

[3] Jabref – bibliography management tool, 2010. 25

[4] Proguard – java obfuscator, 2010. 25

[5] Struts – web application framework, 2010. 25

[6] Xwork – generic command pattern framework, 2010. 25

[7] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *Software Engineering, IEEE Transactions on*, 31(2):150–165, 2005. 7, 9

[8] N. Anquetil and T.C. Lethbridge. Experiments with clustering as a software remodularization method. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 235–255, 1999. 9

[9] Mahir Arzoky, Stephen Swift, Allan Tucker, and James Cain. Munch: An efficient modularisation strategy to assess the degree of refactoring on sequential source code checkings. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 422–429, Washington, DC, USA, 2011. IEEE Computer Society. 11

[10] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 3–14, 2004. 2, 8

[11] Dirk Beyer and Claus Lewerentz. Crocopat: Efficient pattern analysis in object-oriented programs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 294–, Washington, DC, USA, 2003. IEEE Computer Society. 7

[12] R.A. Bittencourt and D.D.S. Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 251–254, 2009. 2, 29

[13] T.R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *Engineering Management, IEEE Transactions on*, 48(3):292–306, 2001. 17

[14] P. Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 17(7):913–933, 2011. 9

[15] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, 1990. 2

[16] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Softw.*, 7(1):66–71, January 1990. 7

[17] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Automated clustering to support the reflexion method. *Information and Software Technology*, 49(3):255 – 274, 2007. <ce:title>12th Working Conference on Reverse Engineering</ce:title>. 10

[18] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEE Proceedings -*, 150(3):161–175, 2003. 2, 8, 9

[19] Jian Feng Cui and Heung Seok Chae. Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems. *Information and Software Technology*, 53(6):601–614, 2011. Special Issue: Best papers from the APSEC. 18

[20] Cleidson Ronald Botelho De Souza. *On the relationship between software dependencies and coordination: field studies and tool support.* PhD thesis, California State University at Long Beach, Long Beach, CA, USA, 2005. AAI3200278. 16

[21] Chuan Duan, Jane Cleland-huang, and Bamshad Mobasher. A consensus based approach to constrained clustering of software requirements. In *International Conference on Information and Knowledge Management*, pages 1073–1082, 2008. 13

[22] Mark Harman, Robert M. Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1351–1358, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 9

[23] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1029–1036, New York, NY, USA, 2005. ACM. 2, 8

[24] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, January 2000. 10

[25] Richard C. Holt. Software architecture abstraction and aggregation as algebraic manipulations. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 5–. IBM Press, 1999. 7

[26] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999. 8, 9, 10

[27] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. 7

[28] Rasha Kashef and Mohamed S. Kamel. Cooperative clustering. *Pattern Recognition*, 43(6):2315 – 2329, 2010. 3, 4, 14, 19, 37, 39

[29] A. Kazem and S. Lotfi. An evolutionary approach for partitioning weighted module dependency graphs. In *Innovations in Information Technology, 2007. IIT '07. 4th International Conference on*, pages 252–256, 2007. 2

[30] Selim Kebir, Abdelhak-Djamel Seriai, Allaoua Chaoui, and Sylvain Chardigny. Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, pages 1–8, 2012. 12

[31] R. Koschke. Atomic architectural component recovery for program understanding and evolution. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 478–481, 2002. 2, 7, 10

[32] A. Lakhotia and J.M. Gravley. Toward experimental evaluation of subsystem classification recovery techniques. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 262–269, 1995. 2

[33] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *J. Syst. Softw.*, 73(2):227–244, October 2004. 7, 9

[34] K. Mahdavi, M. Harman, and R.M. Hierons. A multiple hill climbing approach to software module clustering. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 315–324, 2003. 9

[35] A.S. Mamaghani and M.R. Meybodi. Clustering of software systems using new hybrid algorithms. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 1, pages 20–25, 2009. 8, 11

[36] 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 *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 45–52, 1998. 8, 9

[37] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn R. Chen, and Emden R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 50–59, 1999. 2, 3, 8, 9, 11, 21, 23, 24, 25, 26

[38] O. Maqbool and H.A. Babri. The weighted combined algorithm: a linkage algorithm for software clustering. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 15–24, 2004. 8

[39] O. Maqbool and H.A. Babri. Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE Transactions on*, 33(11):759–780, 2007. 2, 9

[40] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010. 23

[41] Microsoft. Micosoft excel, 2010. 23

[42] B. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 181–190, 2001. 11

[43] Brian S. Mitchell. A heuristic approach to solving the software clustering problem. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 285–288, 2003. 2, 3, 8, 10, 11, 38

[44] Brian S. Mitchell and Spiros Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1375–1382, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 8, 9

[45] Brian S. Mitchell and Spiros Mancoridis. Craft: a framework for evaluating software clustering results in the absence of benchmark decompositions [clustering results analysis framework and tools]. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 93–102, 2003. 18, 21, 23

[46] B.S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *Software Engineering, IEEE Transactions on*, 32(3):193–208, 2006. 9, 11

[47] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM. 1

[48] C. Patel, A. Hamou-Lhadj, and J. Rilling. Software clustering using dynamic analysis and static dependencies. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 27–36, 2009. 2, 12

[49] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 137–148, 2007. 6

[50] M. Risi, G. Scanniello, and G. Tortora. Architecture recovery using latent semantic indexing and k-means: An empirical evaluation. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 103–112, 2010. 31

[51] M. Saeed, O. Maqbool, H.A. Babri, S.Z. Hassan, and S.M. Sarwar. Software clustering techniques and the use of combined algorithm. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 301–306, 2003. 13

[52] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *SIGPLAN Not.*, 40(10):167–176, October 2005. 3, 10, 17, 23, 25, 26

[53] R.W. Schwanke. An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering*, pages pp.83,92, 1991. 2, 7

[54] A. Shokoufandeh, S. Mancoridis, and M. Maycock. Applying spectral methods to software clustering. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 3–10, 2002. 7

[55] Mark Shtern. *Methods for evaluating, selecting and improving software clustering algorithms.* PhD thesis, York University, Canada, 2010. 2, 6, 10, 12, 19

[56] Mark Shtern and Vassilios Tzerpos. Clustering methodologies for software engineering. *Advances in Software Engineering*, 2012(792024):18, 2012. 1, 2, 6, 8

[57] J. Stender. Introduction to genetic algorithms. In *Applications of Genetic Algorithms, IEE Colloquium on*, pages 1/1–1/4, 1994. 8

[58] Adrian Trifu. Using cluster analysis in the architecture recovery of object-oriented systems. Master's thesis, Forschungszentrum Informatik, 2001. 2, 8

[59] Vassilios Tzerpos and Richard C. Holt. The orphan adoption problem in architecture maintenance. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 76–82, 1997. 9, 23

[60] Vassilios Tzerpos and Richard C. Holt. Mojo: a distance metric for software clusterings. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 187–193, 1999. 3, 10, 12, 24, 29

[61] Vassilios Tzerpos and Richard C. Holt. Acdc: an algorithm for comprehension-driven clustering. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 258–267, 2000. 1, 2, 8, 9, 23

[62] Vassilios Tzerpos and Richard C. Holt. On the stability of software clustering algorithms. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 211–218, 2000a. 12, 21, 31

[63] Rajesh Vasa. *Growth and change dynamics in open source software systems.* PhD thesis, Swinburne Univesity of Technology, 2010. 24

[64] Rajesh Vasa, Markus Lumpe, and Allan Jones. Helix – software evolution data set, 2010. 23

[65] T.A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 33–43, 1997. 7

[66] Norman Wilde. Understanding program dependencies, 1990. 16

[67] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 525–535, Washington, DC, USA, 2005. IEEE Computer Society. 12, 29

[68] Xia Xu, Chung-Horng Lung, M. Zaman, and A. Srinivasan. Program restructuring through clustering techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 75–84, 2004. 1

[69] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1979. 18

[70] Qifeng Zhang, Dehong Qiu, Qubo Tian, and Lei Sun. Object-oriented software architecture recovery using a new hybrid clustering algorithm. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, volume 6, pages 2546–2550, 2010. 2, 12