Journal of Universal Computer Science, vol. 24, no. 12 (2018), 1731-1757
submitted: 1/10/17, accepted: 15/11/18, appeared: 28/12/18 © J.UCS

Clustering for Software Remodularization by Using Structural, Conceptual and Evolutionary Features

Amit Rathee
(Department of Computer Engineering, National Institute of Technology, Kurukshetra, India
amit1983_rathee@rediffmail.com)

Jitender Kumar Chhabra
(Department of Computer Engineering, National Institute of Technology, Kurukshetra, India
jitenderchhabra@gmail.com)

Abstract: During various phases of software development lifecycle, the internal structure of the software degrades which finally results in increased maintenance efforts and hence cost. One quick solution to this problem is software remodularization in which restructuring of different software elements such as classes/ packages/ methods is done (without changing their original meaning and functionality). Several researchers have proposed different techniques for software remodularization. Each technique considers two points of view: dependency measurement among software elements (based on structural, conceptual and/or change history based relations) and performing clustering using different algorithms. So, in this paper, first of all, an empirical evaluation is carried out in order to test the role of different dependency relations in modeling dependency among different software elements. From an empirical evaluation, it is observed that the change history of a software system plays a major role in modeling dependency relations and hence must be used along with other relations for more accurate measures. Then, a new weighted dependency measurement scheme is proposed by combining structural, conceptual and change history based relations among software elements together, with more importance to evolutionary dependency relations. Finally, different dependency schemes are evaluated with six clustering algorithms by applying them to four standard open source Java software of variable sizes and belonging to different domains. The obtained results show that our proposed approach is capable of accurately determining dependence relations among various software elements as compared to other similar approaches present in literature and thus increases restructuring accuracy.

Keywords: Software Remodularization, Restructuring, Cohesion, Coupling, Structural/ Conceptual/ Evolutionary/ Logical Dependency, Reverse Re-engineering, Clustering
Categories: D.0, D.2, D.2.8, D.2.13

1 Introduction

Modularity is one of the key concepts in software engineering that helps in implementing the principle of Separation of Concern by dividing a large, complex software into smaller, more manageable modules where each module is expected to serve a particular business domain functionality [Baldwin, 03]. A well-modularized system is easier to maintain and evolve due to the lesser interdependence among modules. Frequent maintenance activities like new requirements, defect fixing, etc. make the internal structure more complex, harder to understand and makes further

modification difficult and hence increased maintenance cost [Chapin, 01; Lehman, 96].

In research domain, this problem is tackled by incrementally improving the internal structure (minimizing coupling and maximizing cohesion) of the software system to lower its complexity, and the technique is known as software remodularization or restructuring [Chikofsky, 90; Nierstrasz, 03]. The key idea behind this technique is to redistribute various software elements such as classes, variables, and functions across packages and/ or class boundaries in order to have future adaptations and extensions of the software system. According to [Fowler, 99], it must alter only the internal structure and should not affect the external behavior of the software system in any way.

In literature, a large number of approaches are proposed to support software remodularization and quality inspection [Mishra, 09] of a software system. Many automated and semi-automated approaches have been proposed by [Wiggerts, 97; Mancoridis, 98; Anquetil, 99; Mitchell, 06; Mitchell, 08; Maqbool, 07; Abdeen, 09; Shtern, 09; Bavota, 10, 13; Beck, 16; Chhabra, 17; Amarjeet, 17; Hwa, 17]. Most of these studies utilize one or two out of three different relations viz structural, conceptual and evolutions of a software. However, no approach has utilized and tested the feasibility of taking all of these three relations together to the best of our knowledge. Further, different researchers have taken different clustering algorithm, but they did not mention which clustering algorithm will perform better during restructuring and under what given situations. However, from the software developer point of view, it is very important to know whether dependency measurement criteria, clustering approach or both affect the most in remodularization process.

Hence, in this paper, a new weighted dependency measurement scheme based on the combined use of structural, conceptual and evolutionary relations is proposed, together with an empirical evaluation with 42 test cases (7 dependency schemes X 6 clustering criteria) in order to justify our approach and to find a best clustering algorithm for remodularization. The proposed weighted dependency measurement scheme is capable of accurately predicting the actual dependency relations among software elements in a software system. This is because it combines all kinds of dependency relations with more importance given to the evolutionary relations. As the evolutionary relations represent co-change coupling between software elements and denotes the set of elements that undergo simultaneous modification. The co-change coupling directly correlates with software defects [D'Ambros, 09]. Therefore, the remodularization based on the proposed dependency scheme is expected to improve the quality issues in software development and maintenance. The primary contribution of this paper includes:

1. To empirically evaluate the role of various dependency relations in remodularization by formulating six different dependency schemes by considering them as an individual and in pairs.
2. To propose a new weighted dependency measurement scheme based on the combined use of all three relations present among different software elements.
3. To justify the proposed scheme by empirically evaluating it with other dependency schemes considered in step 1.

4. To study the effect of different clustering criteria under different dependency schemes on software remodularization.

The rest of this paper is structured as follows: Section 2 gives the background study. Section 3 & 4 specifies the literature work and the proposed methodology. Section 5 describes the case study & Section 6 gives the experimentation details. Section 7 gives threat to validity and Section 8 provides the conclusion and future work.

2 Background Study

The software remodularization process being a two-step process: 1) measuring the dependency relations and, 2) to regroup software elements using clustering. This section explains these terminology & techniques. Since a lot of research volume is already present in literature, we have not covered each topic exhaustively; rather, the details relevant to our study are presented in the following subsections:

2.1 Structural Dependency

Structural dependency analysis has remained a key research field in software re-engineering and remodularization. Different researchers have applied the structural dependency analysis at different granularity levels of the software such as statements, methods, classes, and architecture [STAFFORD, 01; Ferrante, 87; Vieira, 01]. It describes how the interaction between the software units (statement, methods, classes or packages) takes place that encompasses the architecture of a software system. In literature a total of sixteen types of structural relations is identified that may exist in an object-oriented software system [Briand, 99; Erdemir, 14; Maffort, 15]. Two software units P and Q are said to be structurally related if at least one of these relations exists among them. The structural relations are always directional in nature, i.e. if P is structurally related to another unit Q then it is not necessary that Q is also related to P by the same relation. Out of these sixteen structural relations, only eight types of relations are considered in this paper, as they majorly contribute in the measurement of dependency relations among software artifacts namely implements, extends, calls, returns, Is Of Type, Has PARAM, THROWS, and REFERENCES [Chhabra, 17].

2.2 Conceptual Dependency

Conceptual dependency tries to capture the domain-specific relations among software elements by analyzing the semantic information present in the source code in the form of lexemes. While coding a software system, different programmers try to use the domain specific names in form of names of classes, member variables, member functions, parameters, comments & other documentation related to software [Deissen, 06; Caprile, 99; Arnaoudova, 10]. It means that if two classes belong to the same domain, then they possess higher conceptual similarity. In literature, lots of works have been proposed that aim at remodularization of a software system based on the information extracted from different parts of source code such as comments, class names, identifier names, method signature, etc. All such schemes are based on the

principle of using the concept of various Information-Retrieval (IR) techniques such as Vector Space Model (VSM), Latent Semantic Analysis (LSA) [William, 92].

2.3 Evolutionary Dependency

Nowadays, every software system is developed using some kind of Version Control System (CVS) that manages and stores the information about the changes made to a software system in the form of logs. These logs at any instant of time represent the snapshot of the current version of the software and what changes have been done so far. In literature, these logs are used to find out the evolutionary/logical/ co-change dependency among software. Based on the change history, two software elements are said to be evolutionary coupled if they have been simultaneously changed many times. To understand it considers a system consisting of five elements A, B, C, D, and E together with commit information from change history as C1, C2, C3, C4, and C5 of Figure 1. Each commit represents which elements are changed together at a particular instant of time (marked with X symbol). Here, element B and E are supposed to have a higher evolutionary dependency as both have a higher co-change relation as compared to other elements of the system. This relation is depicted with blue rectangles in the Figure 1.

	C1	C2	C3	C4	C5
A	X		X	X	X
B		X		X	X
C			X		
D	X				X
E		X	X	X	X

Figure 1: Evolutionary Dependency of Software Elements

The evolutionary dependency is identified by means of association rule mining approach of data mining where the association among elements is represented as X => Y [Agrawal, 93]. Here, X is called Antecedent and Y as Consequent. The association rule signifies that if event X happens then the event Y will also happen at the same time due to their higher possibility of occurring together. In software engineering, it denotes the possibility that if X changes in a commit, then Y will also undergo modification.

To evaluate different association rules, two criteria are generally used, namely *Support* and *Confidence*. *Support* denotes how frequently a given set of items appears together in the dataset and *Confidence* denotes how many times a given association rule is found to be true. Choice of optimal minimum threshold values for support and confidence has a significant impact over evolutionary dependency.

2.4 Clustering

Clustering may be defined as a process of dividing a set of data points/population into different groups. It is an important step in software remodularization because it aims at segregating groups having similar traits and features by assigning them to different clusters. Clustering can be *hard,* means that at any instant of time a given data point belongs to only one group (most suitable to remodularization) or *soft,* means a given data point can belong to multiple groups at any time. A lot of clustering techniques are discussed in literature viz Partition-Based, Hierarchical Agglomerate, Density-Based & Search-Based Methods. Each of these clustering algorithms is based on an abstract model that represents the interdependencies among various elements extracted from the system to be re-engineered. The clustering process can be represented as an "*Artifact Dependency Graph "(*ADG), which is formalized as a quadruple (*A, R, K, t),* where-

- *A* is a finite set of software artifacts
- *R* is the finite multiset of dependencies among the software artifacts
- *K* is a set of labels such as class, function, variable, function call, uses etc.
- *t* is a mapping function $A \cup R \rightarrow K$, which provides a label for a given software artifact and dependency value.

3 Literature Survey

The key aim of this research paper is many folds: to investigate the effect of various dependency approaches and clustering criteria on the software remodularization by conducting an empirical study and to propose a new dependency measurement scheme. In literature, a large number of research approaches for software remodularization are already proposed. So, in this paper, the related work by different researchers is systematically divided into the following subsections based on the clustering criteria used:

3.1 Partition-Based Software Remodularization

In literature, many different approaches are proposed that aims at grouping software elements with high cohesion and lower coupling quality parameters [Antoniol, 01; Shaw, 03; Cimitile, 95]. Most of these proposed approaches make use of clustering based on detection of strongly connected components in the Module Dependency Graph (MDG) representation of the software system. The MDG representation of a software system gives information about the dependency strength among different modules of a software system. Wiggerts et al. studied remodularization as a clustering problem and proposed various techniques to measure the similarity criteria among different software elements [Wiggerts, 97]. Wu et al. conducted a comparative study of various clustering algorithms in the context of the software remodularization [Wu, 05]. They used MoJoSim metric for the measurement of the authoritativeness of restructuring results. Bavota et al. utilized the structural and semantic relations among classes to construct a dependency graph [Bavota, 10; Bavota, 13]. They further performed the software remodularization by identifying strongly related chains of classes and putting them in a separate package. Corazza et al. also proposed an

approach to partition object-oriented software based on K-Medoids clustering approach and the extracted semantic information [Corazza, 10]. [Yu, 12] used the complexity metrics, based on the structural relations, for building a regression model for predicting the fault-prone software module. Yassin et al. proposed a combined machine learning approach that makes use of both K-Means clustering and Naive Bayes Classifier to reduce false detection and increasing the accuracy rate [Yassin, 13]. [Scitovski, 14; Tzortzis, 14] have studied the k-means algorithm and its variants in data clustering.

3.2 Hierarchical-Based Software Remodularization

Maqbool and Babri have applied hierarchical clustering in the context of remodularization by recovering the underlying architecture [Maqbool, 07]. They also investigated the different measures that are helpful in this context by categorizing various similarity and distance measures into groups according to their characteristics. Mancoridis et al. proposed a collection of hierarchical clustering based algorithm to automatically recover the underlying modular structure using the source-code [Mancoridis, 98]. [Anquetil, 99] performed a comparative analysis of various agglomerative hierarchical clustering algorithms along with describing the entities and the coupling computation among different entities. Corazza et al. proposed an approach for dividing the software system into semantically related classes by using Hierarchical Agglomerative Clustering [Corazza, 11; Zhong, 16]. [Mitchell and Mancoridis, 01] proposed guidelines to compare the performance of different clustering algorithms for source code decomposition, while a more recent work by [Shtern, 09] introduced a method for the selection of a suitable clustering algorithm, given specific needs. Beck et al. proposed a visualization approach that is capable of determining the current modularization criteria of a software system and simultaneously capable of comparing it with other software clustering criteria's commonly being used in the literature [Beck, 16]. Sanner et al. Used hierarchical clustering to determine the effective placement of controllers in software-defined networks [Sanner, 16]. Bishnoi et al. proposed an approach for software remodularization that makes use of a hierarchical clustering algorithm optimized with PSO [Bishnoi, 16].

3.3 Search-Based Software Remodularization

According to the study conducted by Wu et al. when a large-sized software system is remodularized using classical clustering algorithms, then they are not easily extensible [Wu, 05]. So, in literature, the software remodularization is also studied by modeling the problem as search based and applying various metaheuristics to solve the problem. Many researchers studied the problem by modeling as a single-objective problem to improve the cohesion and coupling among software elements [Abdeen, 09; Bavota, 11; Bavota, 13; Praditwong, 11]. [Abdeen, 09] proposed and implemented a heuristic search-based approach to automate the process of optimizing the dependencies. The authors also targeted minimizing the cyclic dependency among packages. [Chhabra, 17] proposed an approach to improve the modular structure by considering lexical and structural aspects and modeling the remodularization problem as single and multi-objective search-based problem. Mitchell and Mancoridis

proposed a search-based meta-heuristic tool called Bunch that supports for various algorithms, namely, Hill-Climbing, Genetic and Simulated Annealing [Mitchell, 06]. The authors further used the Bunch clustering tool to perform the reverse engineering task in order to infer the underlying system abstraction [Mitchell, 08]. They used a search-based strategy in order to check the enormous set of graph partitions. [Ferrucci, 13] applied the multi-objective search-based approach to help balance the project risk and duration against overtime planning. [Barros, 14] performed a case study to evaluate the applicability of search-based remodularization techniques in software architecture recovery context with large sized open source software. Bavota et al. proposed an interactive genetic algorithm (IGA) to perform the software remodularization by putting the software developers in the loop [Bavota, 14]. Their work was motivated by the work of [Praditwong, 11; Bavota, 12; Abdeen, 13]. Similarly, Mkaouer et al. proposed a many-objective based remodularization approach that makes use of NSGA-III [Mkaouer, 15]. [Amarjeet, 17] concludes that metaheuristic algorithms are beneficial in software remodularization and proposed a harmony search-based remodularization solution. The authors in the paper [Hwa, 17] proposed a multi-factor search-based module clustering approach. The authors also introduced two different search-based variations of multi-factor module clustering and compared them with single-factor remodularization.

This paragraph of the paper discusses the pros and cons of different software remodularization approaches discussed in the above sub-sections. The partition-based approaches (K-Means and K-Medoids) are fast, robust, and easier to understand. But, the main limitations of these approaches are- the total number of clusters (modules) in the remodularized system must be known in advance, sensitivity to noise and the outliers, unable to detect hierarchical clusters (clusters within clusters) and sometimes these result in local optima while performing clustering. The Hierarchical-based remodularization approaches unlike the partition-based approaches are free from an initial selection of a total number of clusters in the system, and can easily detect nested clusters. However, these approaches are still sensitive to the noise present and outliers in the input data, and they are dependent on the arbitrary decision regarding selection of the distance metric and the linkage criteria used. Finally, the search-based approaches are metaheuristic in nature and they are expected to provide near optimal results, and they are free from local optima. However, the main limitations of these approaches include a dependency on the quality of the chromosomes used and this technique also suffer from degeneracy (multiple chromosomes representing the same solution).

The most of the remodularization approaches proposed in the literature consider one specific dependency relation and/ or one particular clustering algorithm. Moreover, the literature clearly suggests and highlights the use of importance of many types of dependencies. Many researchers have reported the usefulness of different clustering approaches as well as co-change dependency relations for better modularization quality. However, none of the existing approaches have investigated the integrated effect of dependencies along with a suitable clustering technique. Further, no study has been carried out to identify the weights for the integration of three types of dependencies. These research gaps and facts motivated us to fill this research gap by proposing a new weighted dependency measurement approach in which more importance is given to evolutionary relations.

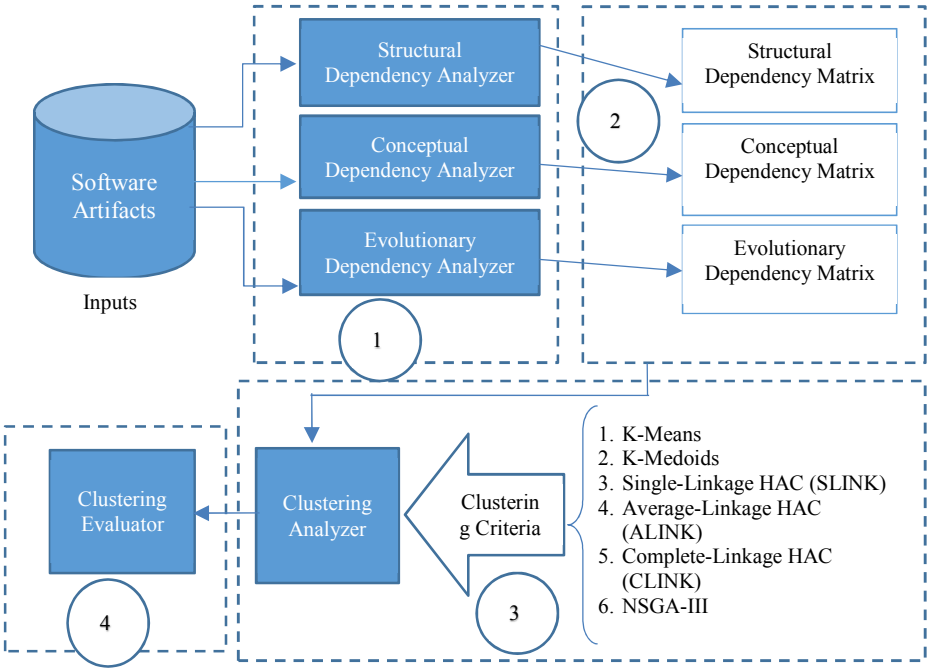

Figure 2: Proposed Methodology to Evaluate Effect of various Dependencies & Clustering Strategies in Remodularization

4 Proposed Methodology

A software remodularization process aims at rearranging the software elements into packages and sub-packages in order to increase the software modularity by decreasing coupling and increasing cohesion quality parameters. The remodularization process helps in minimizing the maintainability efforts at the developer's end. Choice of effective dependency measurement technique and clustering criteria can be very useful to get a good remodularization.

In this paper, the first step of remodularization i.e. measurement of the dependency relation is first studied from six coupling-schemes, namely Structural (ST), Conceptual (CT), Evolutionary (EV), Structural + Conceptual (ST+CT), Structural + Evolutionary (ST+EV) and Conceptual + Evolutionary (CT+EV). This study helps us determine that none of the six designed coupling-schemes are perfect in determining the actual dependency relations among different software elements with higher precision and recall score. This motivates us to combine all three dependency relations together and formulate a new weighted dependency measurement metric viz Structural + Conceptual + Evolutionary (ST+CT+EV). Later on, the proposed metric is also considered for evaluation along with other six coupling-schemes. To evaluate the second step of remodularization i.e. the effect of clustering criteria, we studied it from six clustering-schemes, namely K-Means, K-Medoids, (SLINK) Single Linkage Hierarchical Agglomerate Clustering (HAC),

(ALINK) Average Linkage HAC, (CLINK) Complete Linkage HAC and search based approach (NSGA-III). In this paper, each of the clustering criteria is again evaluated against seven coupling-schemes formulated above for determining the dependency among various software elements in a software system. It helps to evaluate the suitability of each of the clustering criteria for each of the dependency method for a given software system. Figure 2 diagrammatically shows the procedure adopted during the evaluation of the effect of various dependencies schemes and clustering criteria (as formulated above) regarding software remodularization.

During the evaluation procedure, the following four steps are followed in sequence and finally their results are analyzed.

Step 1: Extraction of the structural, conceptual and evolutionary relations among software elements using different software artifacts. These relations are later used to model dependencies among different software elements.

Step 2: The extracted dependency relations are represented in the form of software dependency matrix (SDM) by considering different relations as an individual, pairwise and all together. In these SDM's, every element of the matrix represents the coupling/ dependency strength between the corresponding pair of software elements. Here, the different combination of dependency relations helps in modeling their ability to capture the actual dependency relations. Also, while combining different dependency relations together, it is very important to determine, in what ratio their weights should be considered in order to get optimal results. In this paper, a hit and trial method is used to determine the weight proportion of different dependency relations while combining them together. The detailed description of this method is presented in the subsection entitled "Determining Optimal Weights for Combining Dependency Relations" below this section.

Step 3: The software system is remodularizd based on different clustering techniques as shown in Figure 2 and utilizing the dependency matrices obtained in step-2. The aim of performing this step is to find out which clustering scheme will outperform others when studied on different software systems under different coupling schemes. This helps the researchers in determining which clustering algorithm should be used and under what situations.

Step 4: The obtained clustering results of step-3 are evaluated using expert criteria approach to have their effect on software remodularization.

4.1 Structural Dependency Analyser

Structural dependency analyzer helps to calculate the dependency among different elements by analyzing the structural relations such as method calls, variable use etc. present in the source code of the software. In the present study, total eight structural relations are considered as shown in Table-1. Each of this relation is assigned relative weights w_1, w_2 … w_8 that varies between [0…1]. The weight w_i is given by the formula:

$$Relative\ weight\ w_i = \frac{Tota\ No.of\ i^{th}\ type\ of\ relation \in the\ System}{Total\ No.of\ relation \in System} \rightarrow \quad (1)$$

Relation ID	Relation Name	Relative Weight	Relation ID	Relation Name	Relative Weight
r_1	IMPLEMENTS	w_1	r_5	THROWS	w_5
r_2	EXTENDS	w_2	r_6	Is Of Type	w_6
r_3	CALLS	w_3	r_7	REFERENCE	w_7
r_4	RETURNS	w_4	r_8	Has PARAM	w_8

Table 1: Different Structural Relations under Study

Using the frequency of each relation, r_i and its corresponding weight w_i, the structural dependency (StD) is calculated for each pair of software element E_i and E_j as:

$$StD\ (E_i, E_j) = \sum_{i=1}^{8} f_i * w_i \longrightarrow (2)$$

Here, f_i is the frequency of structural relation of type r_i between E_i and E_j. The overall structural dependency matrix (StDM) is computed using the value of $StD(E_i, E_j)$ as:

$$StDM(i,j) = \sum_{i=1}^{|C|} \sum_{j=1}^{|C|} StD\ (E_i, E_j) \longrightarrow (3)$$

Here, $|C|$ is the total number of the software elements in the system. Since, the matrix StDM is not symmetric due to the directional nature of structural relations. So, to do this, StDM (i, j) is calculated as mentioned below just to possess symmetry relation:

$$StDM(i,j) = max(StDM(i,j), StDM(j,i)) \longrightarrow (4)$$

4.2 Conceptual Dependency Analyser

The conceptual dependency analyzer helps to calculate the dependency among different software elements by analyzing the underlying conceptual domain information embedded by the software developers. Nowadays, different software is developed by following proper standards and guidelines for software development. That means if two or more software elements belong to a similar domain, then, they are supposed to have similar vocabulary [Rathee, 17, Bavota, 13]. In this study, the conceptual information is extracted by tokenizing the source code and considering tokens from six parts as shown in table- 2.

The extracted tokens are first normalized using LSI followed by removing the language specific keywords, removing all stopwords of English language and dividing tokens into their root form by applying Porter's Stemmer Algorithm. In this paper, a weighting scheme for tokens is tf-idf based and the weight of the term t present in the i^{th} part of the document, d Ɛ D is calculated using the following formula:

$$tf - idf(t,i,d) = tf(t,i,d) * log \frac{N}{1 \quad +|d \in D: t \in (d,i)|} \rightarrow \quad (5)$$

S.No.	Source- Code Part Name	Description
1	Comments	includes comments and Javadoc part of the source code
2	Class Names	includes class declaration part of the source code
3	Attribute Names	includes attribute definition part of the source code
4	Method Names	includes all the method signature statements of the code
5	Parameters	attribute names passed as parameters to methods in the code
6	Source Code Statements	statements present in the definition of each method of code

Table 2: Different parts of source code from which tokens are extracted for a given source element

Here, D = { d_1, d_2, …., d_N} is the set of all the software elements, t Ɛ T where T is the set of all tokens present in the system and $N = |D|$ is the count of a total number of software elements. Also, tf (t, i, d) represents the term frequency of token t present in the software element d belonging to i^{th} part of the source code of d. After extracting and weighing tokens, each software element d is represented in the form of a vector representing all the tokens and their corresponding weights. Here, if the token is not present, then its weight is taken as zero. The conceptual dependency matrix (CtDM) is calculated using cosine-based similarity measure as shown below:

$$CtDM(i,j) = \frac{V_i \quad . \quad V_j}{\|V_i\| \quad * \quad \|V_j\|} = \frac{\sum_{k=1}^{n} (w_{k,i} \quad * \quad w_{k,j})}{\sqrt{\sum_{k=1}^{n} w_{k,i}^2} \quad \sqrt{\sum_{k=1}^{n} w_{k,j}^2}} \rightarrow \quad (6)$$

Here, V_i = (w_{i1}, w_{i2}… w_{in}) and V_j = (w_{j1}, w_{j2}… w_{jn}) is the vector representation of element E_i and E_j respectively. The CtDM is a square matrix and its values vary between [0…1]. This matrix is always symmetric in nature.

4.3 Evolutionary Dependency Analyser

The evolutionary dependency analyzer parses the information present in the change history of a software and calculates the evolutionary dependency among software elements. It works in two steps: 1) performing different pre-processing tasks on the change history log and 2) building a co-change graph from the pruned information by applying association rule mining as discussed in Section 2.

Step 1: Pre-Processing Tasks

To effectively study the dependency relations following pre-processing tasks is performed on the information extracted from the repository:

1. *Considering commits associated with maintenance only.* As a VCS repository contains a large amount of information related to changes made to software artifacts. However, not every information present represent the dependency relations. Therefore, we removed all such transactions that are not related to software maintenance. During the study, all the considered commits are related to feature change, bug fixes or other improvements in the system.
2. *Removing commits containing transactions that do not reflect the change made at the class level.* Since remodularization is mainly related to software elements (classes, packages, methods etc.). Therefore, it is a good idea to leave out all those commits that do not involve making changes to software elements. This helps in measures that are more accurate.
3. *Filtering out the commits that show transactions which affect a large number of software elements at the same time.* During our manual analysis of commits associated with different software, it is found out that such commits do not represent the actual evolutionary dependency in the system. This is because such commits generally represent software quality improvement tasks such as dead code elimination or refactoring related task such as renaming on a software system. So, it is better to leave out all such commits.
4. *Splitting commits that involve changes related to multiple maintenance issues into individual issues.* Sometimes more than one maintenance issues are simultaneously handled by developers and all changes to such issues are stored in single commits. So, it is better to separate such issues into different independent commits in order to have a better view of an evolutionary coupling.
5. *Merging commits that are related to the same maintenance issue.* By manual inspection of the change history, it was observed that sometimes the changes related to a single issue are made in parts. So, it is better to merge all such commits which represent a change to a single maintenance issue in order to have a better view of an evolutionary coupling.
6. *Removing duplicate commits.* As due to splitting or merging, there are chances that there exists duplicity among commits. So, all such duplicity is removed.

Step 2: Co-Change Graph

After performing the step 1, the change history information is represented as a set C = $\{C_1, C_2 \ldots C_n\}$, where each C_i represents a commit denoting the number of elements that undergo modification simultaneously. In this step, an apriori association rule mining algorithm is applied to C. These algorithms extract sets of items that represent the frequently co-changed elements that are likely to be linked together. All such co-changed sets may not be evolutionary coupled. So, to evaluate them, *Support* and *Confidence* measures are used based on *min_support* and *min_confidence* values. Support for an association rule $X = Y$ is the measure of the probability that both the items X and Y appears together in an event and is calculated as:

$$Support(X = Y) = Support(XY) = C_{XY} \longrightarrow (7)$$

Where C_{XY} is the total number of commits in the change history in which both X and Y appeared together. Similarly, Confidence for an association rule $X = Y$ denotes the

probability that item Y will appear in a commit operation provided item X appears in that commit operation. It is calculated by using the following formula:

$$Confidence(X = Y) = \frac{Support(X = Y)}{Support(X)} = \frac{C_{XY}}{C_X} \longrightarrow (8)$$

Where C_X is the total number of commits in which item X appeared independently or in association with other items and C_{XY} is the total number of commits in the change history in which both X and Y appeared together. For each pair of the software elements, we calculate both support and confidence measure and later using min_support and min_confidence threshold values, we created the co-change graph G = (V, E) where V is the set of all software elements and E is the set of edges between them. Finally, using the co-change graph we construct the evolutionary dependency matrix EvDM.

4.4 Clustering Analyser

The Clustering Analyser performs clustering by considering various dependency relations in the form of SDM and clustering criteria as inputs. Here, total 42 test cases (7 dependency X 6 clustering schemes) are considered and their clustering output is recorded. The various SDM measures the ability of different dependency schemes in predicting the actual dependency among software elements. Since different schemes have different measuring capabilities, so, clustering output is different in each case.

4.5 Clustering Evaluator

The clustering evaluator evaluates the clustering results obtained after considering each of 42 test cases. In this paper, the evaluation is done using expert criteria approach in which we compare obtained clustering result with the well-known gold standard of the corresponding software. Gold Standard/ Authoritative Partition for every software system is obtained by rigorous analysis by a team consisting of software developers having an industry background with varied experience in the software development and postgraduate students who are well *familiar with Java-based software development. Together they analyzed the software* systems rigorously & calculates the gold standard by using the following rules:

1. The original software system's architecture as available with the Java Archive (JAR) file of an associated software system is taken into account in determining the gold standard.
2. If a sub package in the original structure has software elements less than or equal to five are merged with its parent package.
3. If, the original structure is not available, then the underlying package structure is used for determining the gold standard.
4. After following step 1- 3, the same structure of the software system is presented to various software developers and their comments are obtained.
5. Finally, after considering comments and opinion of software developers regarding software system structure, the expected gold standard is decided.

4.6 Determining Optimal Weights for Combining Dependency Relations

This subsection of the paper explains the hit and trial method used to determine the optimal weights used for combining different dependency relations together in order to model a new dependency scheme. In hit and trial method, different weighing schemes are decided randomly and are used to model different dependency schemes. Finally, the obtained dependency schemes are evaluated & the one that gives the overall best result is selected as an optimal weight value. In this paper the randomly decided weights are (0.4, 0.6); (0.6, 0.4); (0.5, 0.5); (0.2, 0.8) and (0.8, 0.2) for pairwise dependency schemes and (0.3, 0.3, 0.3); (0.25, 0.25, 0.50); (0.6, 0.2, 0.2); (0.2, 0.6, 0.2) and (0.2, 0.2, 0.6) for a scheme which combines all three dependency relations together. All these random weights constitute the considered weighing scheme, which is empirically evaluated for modeling the actual dependency relations. The obtained results are shown in Figure 3. Here, the y-axis denotes an F-measure value and x-axis represents the weighing scheme used under different dependency schemes. Here for independent dependency relations, no weighing scheme is used. For other dependency schemes, different weighing schemes are used and the corresponding results are obtained.

From the results depicted in bar chart diagrams for different software systems, it is clear that the change history of a software plays an important role in remodularization & when it is combined with other forms of dependency relations viz SEM, ST and/ or SEM+ST then a higher weight assigned to change history relations is capable of accurately model the overall dependency relations. The weights mainly (0.4, 0.6) and (0.2, 0.8) used in dependency schemes, namely SEM+CH & ST+CH gives better results as compared to weights (0.6, 0.4) and (0.8, 0.2). These results motivate us to use the change history in a combination with conceptual & structural dependency relations and to propose a new dependency measurement scheme called SEM+ST+CH and to conclude that a weight factor of 60% or more assigned to evolutionary relations always helps in modeling the dependency relations with higher accuracy.

The above conclusion regarding 60% or higher weight factor to evolutionary relation holds in our proposed dependency scheme also. Here, the weight (0.2, 0.2, 0.6) gives more accurate results as compared to (0.2, 0.6, 0.2) and (0.6, 0.2, 0.2).

5 Case Study

This section describes the experimentation steps carried out to evaluate the effect of different dependency and clustering criteria on remodularization. It describes the definition and the planning done before actual experimentation is done [Yin, 03].

5.1 Definition and Context

Based on the Goal Question Metric (GQM) formulation [Mashiko, 97], the goal of the study is to evaluate which combination of dependency and clustering criteria used affects most during the remodularization of the software system. The GQM formulation helps us in deciding the important aspects of our study well in advance before the actual experimentation [Wohlin, 00].

The experimentation is carried out on four open source standard Java software systems belonging to different domains. The idea behind the selection of these software systems is that they are widely being used among the research community [Rathee, 17; Amarjeet, 17]. Moreover, these are standard software systems whose change history has been properly maintained by the developers. This gives us an opportunity to mine more accurate underlying data. Table-3 summarizes the subject systems together with basic metrics about them. The first column is the ID, the second and third column describes the name and the version of the subject system. Column fourth and fifth gives a total number of classes and KLOC. The sixth and seventh column denotes the total number of commits present in change history and the time slab considered for change history. The eighth column gives a short description of the subject system.

ID	Soft. Name	Version	# Classes	KLOC	# Commits	Period	Description
1	JFlex	1.5	61	6.9	1254	2006 to 2017	A Java lexical generator
2	Junit	4.10	164	15.0	5643	2003 to 2017	A Java testing framework
3	Commons-io	2.5	62	6.13	3211	2007 to 2017	Apache library
4	JPF	2.5.1	89	8.7	1156	2002 to 2017	Plug-in framework

Table 3: Subject Systems Considered for Experimentation

5.2 Experimentation Planning

In order to evaluate the effectiveness of dependency and clustering criteria in software remodularization on the different subject systems, the steps shown in Figure 2 are carried out one by one. For each of the subject system, the system is considered as a Java Archive (JAR) file of the specified version and both the source and binary version file are taken for the study. The structural information is extracted using Structure 101 tool and corresponding dependency matrix is determined using the custom-made software in Java. Similarly, the source code version of the JAR file is used to determine the conceptual dependency matrix by extracting the elementary tokens present in the source code. This step is also carried out with the help of a custom-made software in Java. In order to determine the evolutionary information, the change history information from beginning till the version taken under the study of different subject systems is extracted from the GitHub repository. A custom-made tool is designed in Java to process and analyze the extracted change logs. It creates a co-change graph by analyzing the change logs and finally the co-change graph is converted into an evolutionary dependency matrix.

As our empirical evaluation procedure is based on the expert criteria approach & is widely used by researchers [Risi, 12; Scanniello, 10; Wu, 05; Bittencourt, 09]. So, first, it is necessary to identify an authoritative partition of each subject system. The approach used for this is already discussed in Section IV of this paper. The rationale behind this approach is simply that if a given approach is capable of repartitioning the

considered bunch of classes into its authoritative structure, then that approach is considered sufficient and it also performs for any other software system [Wu, 05].

5.3 Research Questions and Variables

As the case study in the current paper focuses on evaluating the effect of different dependency schemes and clustering criteria on software remodularization, so, the following hypothesis and research questions are formulated:

HP 0: there is no significant statistical difference between the data of various SDM (Null Hypothesis).
HP 1: there is a significant statistical difference between the data of various SDM, which is further used for clustering, and evaluation under different schemes (Alternate Hypothesis).
RQ 1: how the choice of dependency measurement criteria viz structural, conceptual, and evolutionary or their combination in a pair of two and three affects the software remodularization results?
RQ 2: how the choice of different clustering criteria affects the results of software remodularization based on the choice of dependency considered in RQ1?

The final clustering produced during the evaluation should closely resemble with the given authoritative partition [Corazza, 11]. The closeness of the obtained cluster P_C and the authoritative partition (Gold Standard) P_A are judged based on Precision, Recall and F-Measure metrics [Powers, 11]. These metrics are evaluated at the micro level by calculating these values at the package level. The overall value is obtained by averaging micro values for each package. The values of these metrics vary between 0 and 1. Higher the value of the f-measure metric better the approach used for remodularization.

5.4 Statistical Analysis

To test the validity of the considered hypothesis, Mann-Whitney-Wilcoxon Test is performed [Conover, 98]. Using this test, we can decide whether or not the population distributions are identical without assuming them to follow the normal distribution. It helps to determine if the variations seen in the obtained results are random or are statistically significant. Further, if statistically significant, then they are due to the fact that the considered dependency schemes capture different dimensions of the actual dependence among different software elements. To apply this test, we randomly select samples from the population and then corresponding p-value obtained for the considered sample after applying the Wilcoxon Test are compared. Here, the obtained F-Measures values of a considered software system under different clustering schemes constitute the considered population. During the test, the obtained average p-value is 0.00185, which is significantly less than the considered minimum threshold of 5% (0.05). Therefore, we rejected the null hypothesis (HP0) and supports the alternate hypothesis (HP1) for the obtained dependency data. The hypothesis, HP1, signify that the variation in the results is in fact due to the fact that different dependency schemes capture a different aspect of the actual dependency relations in a software system.

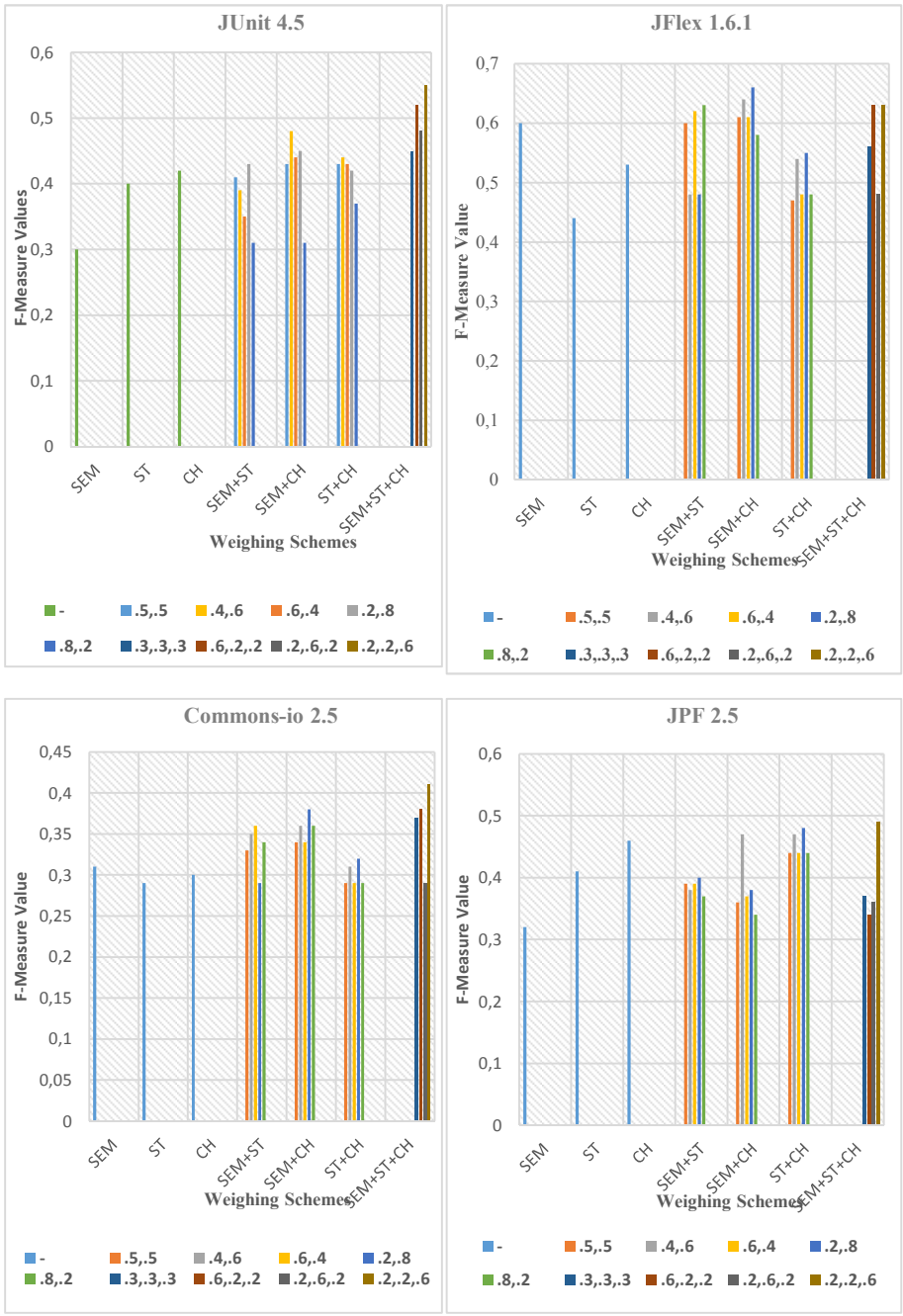

Figure 3: Representation of F-Measure Value of different software systems corresponding to different weighing schemes considered.

6 Results And Discussion

In this section of the paper, the results obtained after empirically evaluating the software remodularization problem from different aspects (evaluating them from dependency and clustering point of view) is presented. Here, results for both the proposed dependency measurement metric as well as other formulated dependency schemes are presented for the sake of comparison. Finally, a discussion regarding the implications of the obtained results is presented. Table- 4 show the obtained values of precision, recall, and f-measure for different subject systems studied under different software remodularization scenarios (total 42 schemes designed by considering dependency and clustering criteria).

RQ 1: how the choice of dependency measurement criteria viz structural, conceptual, and evolutionary or their combination in a pair of two and three affects the software remodularization results?

Figures 4-7 show the boxplots for different F-Measure values obtained by considering different dependency measurement schemes (including the proposed dependency measurement metric). It is clear from the plots that the proposed approach that makes combined use of weighted structural, conceptual and evolutionary dependency relations performs far better than any other dependency measurement schemes formulated by taking dependency relations as an individual or in among different software elements. In the case of Figure 5, the accuracy (measured as an F-Measure score) in the case of the proposed dependency scheme (ST+CT+EV) is 76% and is much higher than the corresponding score in other dependency schemes. Moreover, whenever the evolutionary relations are combined with structural and conceptual relations, it shows an increase in F-Measure score by 15% to 8% respectively. In the case of JFLEX software (Figure 4), the accuracy of the proposed dependency scheme is highest and is 72%. Similarly, the contribution of evolutionary relations in combination with structural and conceptual relations separately improves the result by 20% and 10% respectively. Similarly, with Commons-IO and JPF software system (Figures 6 & 7 respectively), the accuracy in the case of the proposed dependency scheme is highest as compared to other dependency schemes. This justifies the feasibility of our proposed dependency measurement scheme.

Also, it is clear from the boxplots that the change history plays an important role in representing actual dependency relations among software elements. It is also clear that whenever the change history relation (evolutionary relations among different software elements) is paired with any of the other dependency relation, namely structural or conceptual, then the accuracy of the software remodularization process increases (indicated by higher values of the F-Measures in pairs as compared to individual scores). This justifies our finding that the change history plays an important role in accurately determining the dependency relations among software elements and also the proposed dependency measurement metric which is defined as a weighted combination of structural, conceptual and evolutionary relations, as it outperforms the other dependency measurement schemes earlier formulated in this research paper.

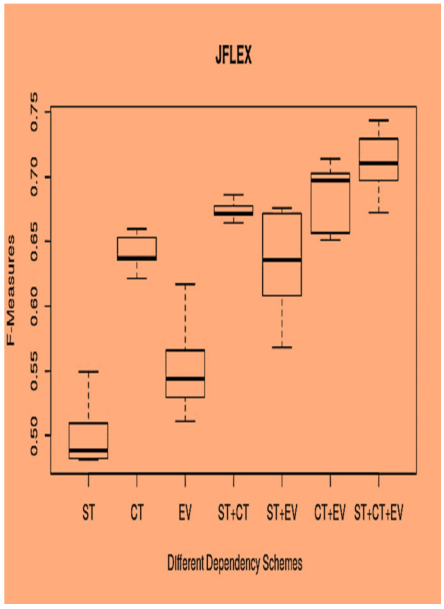

Figure 4: Boxplots under different Dependency Schemes for JFlex.

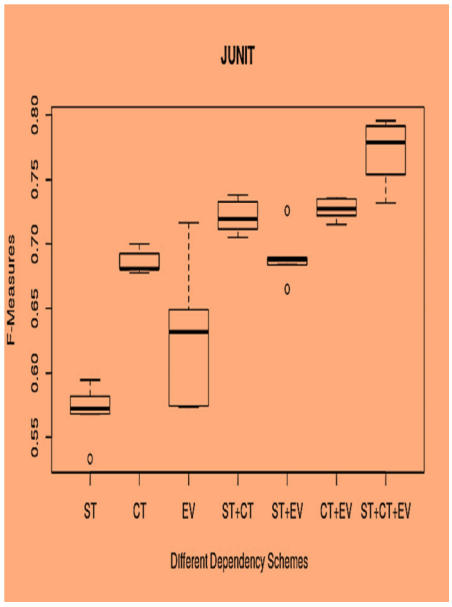

Figure 5: Boxplots under different Dependency Schemes for Junit.

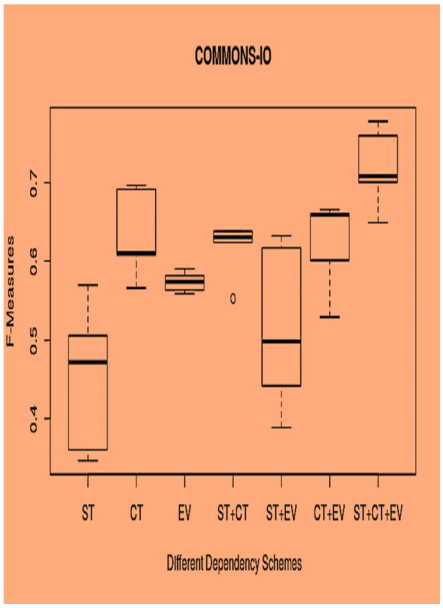

Figure 6: Boxplots for Commons-io.

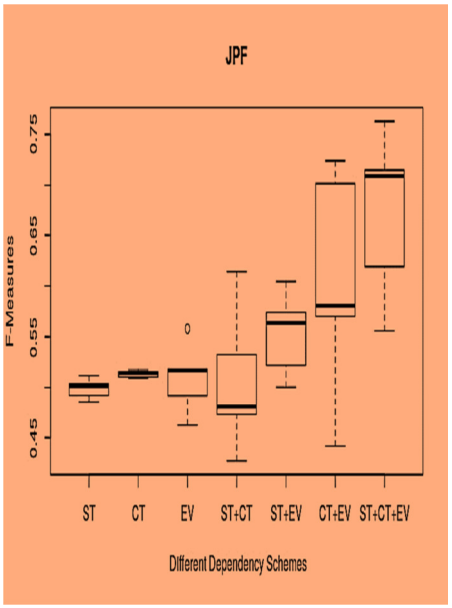

Figure 7: Boxplots under different Dependency Schemes for JPF.

RQ 2: how the choice of different clustering criteria affects the results of software remodularization based on the choice of dependency considered in RQ1?

Figure- 8 shows the boxplot between F-Measure value and different clustering schemes. This helps us visualize the role of different clustering algorithms.

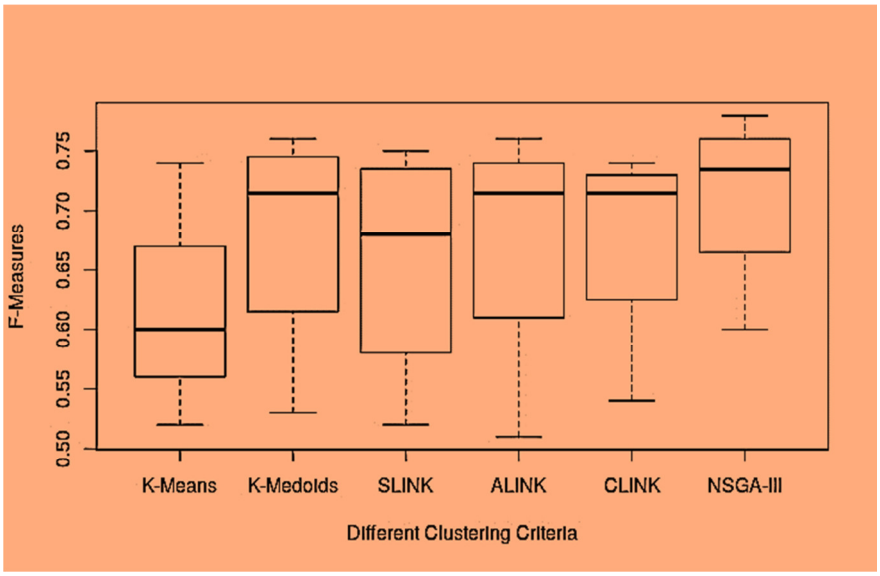

Figure 8: Boxplot showing performance of different Clustering Schemes.

In Figure 8, our proposed dependency scheme is considered during evaluation. Here, the F-Measure value against different clustering scheme denotes their ability in representing how close the obtained clusters matches to the clusters identified manually using the expert criteria approach. From the boxplot, it is clear that evolutionary-based clustering (NSGA-III) outperforms other clustering algorithms given a dependency measurement scheme. Further, the partition-based clustering algorithm (K-Medoids) and hierarchical-based clustering algorithms (ALINK & CLINK) are nearly equal in clustering different software elements given a dependency measurement scheme. The obtained results in these cases are comparable to that of NSGA-III. Whereas, the SLINK clustering algorithm performs average in clustering while the K-Means clustering algorithm is least in performance. Here, it is observed that the NSGA-III clustering algorithm outperforms the rest of the clustering algorithm under the proposed dependency scheme (that makes combined use of structural, conceptual and evolutionary relations). This is because the NSGA-III is a non-dominated sorting genetic algorithm and it has a slight edge as compared to K-Medoids, ALINK & CLINK when applied to large sized software systems.

Soft. Remodul. Criteria		JFlex			JPF			JUnit			Commons-io		
Clust.	**Dep.**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
K-Means	S-1	.80	.35	.48	.54	.38	.45	.93	.72	.76	.79	.16	.34
	S-2	.69	.62	.66	.79	.37	.50	.90	.67	.69	.63	.10	.70
	S-3	.54	.64	.59	.59	.37	.46	.94	.61	.72	.10	.44	.10
	S-4	.85	.65	.72	.49	.42	.45	.92	.76	.74	.75	.15	.53
	S-5	.80	.35	.48	.51	.46	.48	.92	.68	.73	.82	.04	.35
	S-6	.90	.55	.69	.40	.39	.39	.91	.65	.71	.34	.15	.52
	S-7	.74	.57	.64	.82	.38	.52	.93	.75	.74	.57	.05	.60
K-Medoids	S-1	.83	.36	.50	.63	.40	.49	.94	.76	.75	.85	.16	.58
	S-2	.95	.59	.73	.82	.37	.51	.91	.64	.75	.45	.21	.56
	S-3	.91	.39	.55	.59	.36	.45	.92	.56	.59	.86	.15	.64
	S-4	.92	.68	.73	.51	.35	.42	.93	.63	.74	.83	.14	.56
	S-5	.83	.36	.50	.59	.39	.47	.91	.65	.64	.39	.14	.56
	S-6	.94	.56	.70	.50	.40	.44	.93	.67	.71	.37	.12	.50
	S-7	.66	.56	.60	.82	.39	.53	.93	.70	.76	.89	.13	.73
SLINK	S-1	.83	.36	.50	.62	.35	.45	.86	.38	.53	.80	.10	.65
	S-2	.95	.59	.73	.84	.37	.51	.96	.59	.73	.84	.09	.62
	S-3	.91	.39	.55	.59	.36	.45	.92	.40	.56	.86	.14	.61
	S-4	.88	.56	.69	.85	.36	.51	.92	.58	.71	.80	.13	.55
	S-5	.80	.35	.48	.84	.37	.51	.91	.62	.72	.90	.10	.52
	S-6	.94	.56	.70	.59	.39	.47	.93	.63	.72	.72	.13	.55
	S-7	.90	.58	.71	.88	.37	.52	.94	.67	.75	.89	.11	.72
ALINK	S-1	.83	.36	.50	.49	.38	.43	.83	.35	.49	.89	.18	.65
	S-2	.95	.59	.73	.82	.37	.51	.84	.61	.70	.70	.20	.70
	S-3	.91	.39	.55	.59	.37	.46	.92	.40	.56	.69	.04	.64
	S-4	.88	.56	.69	.84	.37	.51	.85	.65	.72	.72	.09	.57
	S-5	.80	.35	.48	.84	.37	.51	.86	.53	.68	.77	.11	.55
	S-6	.94	.56	.70	.72	.41	.52	.90	.67	.73	.80	.15	.56
	S-7	.87	.59	.70	.82	.37	.51	.93	.72	.76	.87	.17	.72
CLINK	S-1	.86	.38	.52	.61	.38	.47	.93	.72	.76	.81	.14	.57
	S-2	.90	.64	.75	.82	.37	.51	.90	.67	.69	.63	.11	.70
	S-3	.91	.39	.55	.59	.36	.45	.94	.61	.72	.87	.38	.60
	S-4	.87	.58	.70	.66	.38	.48	.92	.76	.74	.66	.10	.56
	S-5	.91	.64	.75	.82	.37	.51	.92	.68	.73	.91	.45	.58
	S-6	.83	.36	.50	.47	.40	.43	.91	.65	.71	.65	.13	.56
	S-7	.87	.58	.70	.78	.41	.54	.93	.75	.74	.89	.15	.72
NSGA-III	S-1	.90	.67	.69	.86	.38	.53	.88	.54	.62	.86	.38	.53
	S-2	.94	.61	.72	.96	.59	.73	.93	.65	.73	.96	.59	.73
	S-3	.92	.64	.70	.92	.40	.56	.92	.64	.70	.92	.40	.56
	S-4	.91	.66	.70	.92	.58	.71	.92	.68	.71	.92	.58	.71
	S-5	.91	.66	.70	.91	.62	.72	.93	.66	.70	.91	.62	.72
	S-6	.92	.69	.72	.93	.63	.72	.92	.64	.70	.93	.63	.72
	S-7	.94	.72	.75	.94	.67	.75	.94	.75	.78	.94	.67	.75

The legends used in this table are:- S-1: Structural; S-2: Conceptual; S-3: Evolutionary; S-4: Structural + Conceptual; S-5: Structural + Evolutionary; S-6: Conceptual + Evolutionary; S-7: Structural + Conceptual + Evolutionary; P: Precision; R: Recall; F: F-Measures

Table 4: Obtained Results under different dependencies and clustering schemes for different software systems under study.

Based on the obtained values of precision, recall and f-measure for different studied software systems, it is concluded that the proposed weighted dependency measurement scheme, to measure the dependency relations by combined use of structural, conceptual and evolutionary relations among software elements, significantly improves the software remodularization results. The obtained results are much better as compared to other dependency measurement schemes formulated by considering structural, conceptual and evolutionary as an individual or as pairs. The proposed dependency measurement scheme performs well with different clustering-schemes taken into consideration. In addition, based on the results, the evolutionary search-based clustering algorithm (NSGA-III) outperforms the other clustering techniques, whereas the results of hierarchical clustering techniques (ALINK & CLINK) and K-Medoids are comparable with it. The least performer clustering algorithm is K-Means. The reasons behind getting a high value of precision and low value of recall that results in slightly improved values of the F-Measure is that the proposed weighted dependency scheme is capable of better capturing the actual dependency relations among different software elements.

7 Threats to Validity

Three threats to validity [Maxwell 2014] to our proposed research methodology are identified. First, the obtained results are based on the authoritative partition of a software system, which is a subjective decision and may differ from professionals, resulting in some deviations in the obtained authoritative partition [Arisholm, 04]. The second threat is about the generalization of the obtained results. The proposed research methodology is applied to four open source software systems, and this risk has been minimized by taking software systems belonging to different domains and of variable sizes. The third threat to validity is regarding the determination of weights used while combining various dependency relations. Ideally, all possible combinations of weights should be considered, but it is practically impossible. Thus, the weights are determined using hit and trial method and all major possibilities have been considered to minimize this threat.

8 Conclusion And Future Work

Having modular design is always a desirable characteristic of software engineering & remodularization being one of the approaches to enhance the modular structure of a software. The accuracy of any remodularization approach is affected by two factors: dependency measurement and clustering criteria. In this paper, the dependency measurement is studied from three viewpoints viz structural, conceptual and evolutionary relations existing in a software. Based on these different relations, a total of six dependency schemes is designed by considering them as individual and pairwise. Further, a weighing scheme is designed to measure the contribution that each relation plays while combining them in pairwise. Based on the study and experimentation performed, it is concluded that a weight factor >= 60% for evolutionary relations always enhance the quality of a remodularization. This motivates us to give more importance to evolutionary relations during dependency

measurement and to propose a new dependency measurement scheme that combines three dependency relations together with suitable weights. Furthermore, remodularization is also studied by considering six different clustering algorithms. Finally, different combinations of dependency and clustering schemes are empirically evaluated using expert criteria approach. The obtained results show that the proposed approach, significantly increase modularity by enhancing accuracy in dependency measurement.

In the future, the proposed dependency approach can be utilized for the recovery of the underlying architecture in legacy systems. There is also a scope to apply our proposed approach for software refactoring by applying the approach at the more granularity levels. Other future works involve the evaluation of the proposed approach against its capability in the identification / extraction of reusable software components. Moreover, as the proposed weighted dependency approach considers assigning arbitrary weights, so another possible direction is to calculate absolute weights by performing the empirical evaluation.

References

[Abdeen, 09] Abdeen, H., Ducasse, S., Sahraoui, H. A., Alloui, I.: Automatic package coupling and cycle minimization, In Proceedings of the 16th Working Conference on Reverse Engineering. IEEE, 103–112, 2009.

[Abdeen, 13] Abdeen, H., Sahraoui, H., Shata, O., Anquetil, N., Ducasse, S.: Towards automatically improving package structure while respecting original design decisions, In proc. of the 20th Working Conference on Reverse Engineering (WCRE). pp. 212–221, 2013.

[Agrawal, 93] Agrawal, R., Imieliski, T., Swami, A.: Mining association rules between sets of items in large databases, In Proc. ACM SIGMOD, 22(2): 207 – 216, 1993.

[Amarjeet, 17] Amarjeet, Chhabra, J.K.: Harmony search based remodularization for object-oriented software systems, Computer Languages, Systems & Structures, 47 (2), 153-169, 2017.

[Anquetil, 99] Anquetil, N., Lethbridge, T.: Experiments with clustering as a software remodularization method, In Proceedings of the sixth Working Conference on Reverse Engineering. IEEE, 235–255, 1999.

[Arisholm, 04] Arisholm, E., Sjoberg, D.: Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software, IEEE Transactions on Software Engineering 30 (8), 521–534, 2004.

[Arnaoudova, 10] Arnaoudova V., Eshkevari L., Oliveto R., Guéhéneuc Y.-G. And Antoniol G.: Physical and conceptual identifier dispersion: Measures and relation to fault proneness, In 26th IEEE Inter. Conf. on Software Maintenance (ICSM'10), Timisoara, Romania, 1-5, 2010.

[Baldwin, 03] Baldwin, C., Clark, B.: Design Rules: Power of Modularity. MIT Press, 2003.

[Barros, 14] Barros, M., Farzat, F.A., Travassos, G.H.: Learning from optimization: a case study with Apache Ant, Inf. Softw. Technol. 57, 684–704, 2014.

[Bavota, 10] Bavota, G., Lucia, A. D., Marcus, A., Oliveto, R.: Software remodularization based on structural and semantic metrics, In Proceedings of the Working Conference on Reverse Engineering. 195–204, 2010.

[Bavota, 12] Bavota, G., Carnevale, F., Lucia, A. De, Penta, M. Di, and Oliveto, R.: Putting the developer in the loop: An interactive GA for software re-modularization, In Proceedings of the 4th International Symposium on Search Based Software Engineering (SSBSE'12). 75–89, 2012.

[Bavota, 13] Bavota, G., Lucia, A. De, Marcus, A., and Oliveto, R.: Using structural and semantic measures to improve software modularization, J. Empirical Softw. Eng. 18, 5, 901–932, 2013.

[Beck, 13] Beck, F., Diehl, S.: On the impact of software evolution on software clustering, Empirical Software Engineering, 18(5), 970– 1004, 2013.

[Beck, 16] Beck, F., Melcher, J., and Weiskop, D.: Identifying Modularization Patterns by Visual Comparison of Multiple Hierarchies, ICPC 2016, Austin, Texas.

[Bishnoi, 16] Bishnoi, M., Singh, P.: Modularizing Software Systems using PSO optimized Hierarchical Clustering, International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT), IEEE, 2016.

[Bittencourt, 09] Bittencourt, R.A., Guerrero, D.D.S.: Comparison of graph clustering algorithms for recovering software architecture modular views. In: Proceedings of the European conference on software maintenance and reengineering. IEEE Computer Society, pp 251–254, 2009.

[Briand, 99] Briand, L. C., Daly, J. and W¨ust, J.: A unified framework for coupling measurement in object-oriented systems, IEEE Trans. on Soft. Eng., 25 (1), 91–121, 1999.

[Caprile, 99] Caprile, C. and Tonella, P.: Nomen EST Omen: Analysing the Language of Function Identifiers, 6th IEEE Working Conference on Reverse Engineering (WCRE'99), Atlanta, Georgia, USA, 112-122, 1999.

[Chapin, 01] Chapin, N., Hale, J. E., Khan, K. Md.: Types of software evolution and software maintenance, Jour. of Soft. Mainten. And Evo. Research and Practice, 13:3–30, 2001.

[Chhabra, 17] Amarjeet, Chhabra, J.K.: Improving modular structure of software system using structural and lexical dependency, Information and Software Technology, 82, 96–120, 2017.

[Chikofsky, 90] Chikofsky, E. J., and Cross II, J. H.: Reverse engineering and design recovery: A taxonomy, IEEE Software, 7(1):13–17, 1990.

[Conover, 98] Conover, W.J.: Practical nonparametric statistics, 3rd. Wiley, 1998.

[Corazza, 10] Corazza, A., Di Martino, S., Scanniello, G.: A probabilistic based approach towards software system clustering, In: 14th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 88–96, 2010.

[Corazza, 11] Corazza, A., Di Martino, S., Maggio, V., Scanniello, G.: Investigating the use of lexical information for software system clustering, In: Proceedings of European conference on software maintenance and reengineering. IEEE Computer Society, 35–44, 2011.

[D'Ambros, 09] D'Ambros, M., Lanza, M., and Robbes, R.: On the Relationship Between Change Coupling and Software Defects, *In 16th Working Conference on Reverse Engineering*, Lille, 135-144, 2009.

[Deissen, 06] Deissenboeck, F., and Pizka, M.: Concise and Consistent Naming, Software Quality Journal 14(3): 261-282, 2006.

[Erdemir, 14] Erdemir, U., Buzluca, F.: A learning-based module extraction method for object-oriented systems, J. Syst. Software 97, 156–177, 2014.

[Ferrante, 87] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, ACM Trans. on Programming Lang. and Systems 9, 319–349, 1987.

[Ferrucci, 13] Ferrucci, F., Harman, M., Ren, J., and Sarro, F.: Not going to take this anymore: multi-objective overtime planning for software engineering projects. In Proceedings of the International Conference on Software Engineering. IEEE, 462–471, 2013.

[Fowler, 99] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code, Addison Wesley, Boston, 1999.

[Gunasekara, 14] Gunasekara, R.P.T.H., Wijegunasekara, M.C.: Comparison of major clustering algorithms Using Weka Tool, Inter. Conf. On Adv. in ICT for Emerging Regions, IEEE, 2014.

[Hwa, 17] Hwa, J., Yoo, S., Seo, Y.S., Bae, D.H.: Search-Based Approaches for Software Module Clustering Based on Multiple Relationship Factors. International Journal of Software Engineering and Knowledge Engineering, vol. 27 (07), pp. 1033 – 1062, 2017.

[Lehman, 96] Lehman, M.: Laws of software evolution revisited, In European Workshop on Software Process Technology, pages 108–124, Berlin, Springer, 1996.

[Maffort, 15] Maffort, M., Valente, M.T., Terra, R., Bigonha, M., Anquetil, N., Hora, A.: Mining architectural violations from the version history. In: Emp. Soft. Eng. Springer, 1–42, 2015.

[Mancoridis, 98] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y. F., and Gansner, E. R.: Using automatic clustering to produce high-level system organizations of source code. In Proceedings of the International Workshop on Program Comprehension. 45–55, 1998.

[Maqbool, 07] Maqbool, O., and Babri, H. A.: Hierarchical clustering for software architecture recovery. IEEE Trans. Soft. Eng. 33(11), 759–780, 2007.

[Mashiko, 97] Mashiko, Y., Basili, V.: Using the GQM paradigm to investigate influential factors for software process improvement. J Syst Softw 36(1):17–32, 1997.

[Maxwell, 04] Maxwell, J.: Qualitative research design: An interactive approach. Sage Publications Inc., 2004.

[Mishra, 09] Mishra, D., Mishra, A.: Simplified software inspection process in compliance with international standards, Computer Standards & Interfaces, 31 (4), 763-771, 2009.

[Mitchell, 06] Mitchell, B. S., and Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Trans. Softw. Eng. 32, 3, 193–208, 2006.

[Mitchell, 08] Mitchell, B. S., and Mancoridis, S.: On the evaluation of the bunch search-based software modularization algorithm. Softw. Computing 12, 1, 77–93, 2008.

[Mkaouer, 15] Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., and Ouni, A.: Many-objective software remodularization using NSGA-III. ACM Transactions on Software Engineering Methodology 24, 3, 17:1–17:45, May 2015.

[Nierstrasz, 03] Nierstrasz, O., Ducasse, S., and Demeyer, S.: Object-Oriented Reengineering Patterns, Morgan Kaufmann Publishers, 2003.

[Powers, 11] Powers, David M. W.: Evaluation: From Precision, Recall and F-Measure to ROC. Journal of Machine Learning Technologies. 2 (1): 37–63, 2011.

[Praditwong, 13] Praditwong, K., Harman, M., and Yao, X.: Software module clustering as a multi-objective search problem, IEEE Trans. on Soft. Eng., vol. 37, no. 2, 264–282, Mar. 2011.

[Rathee, 17] Rathee, A., Chhabra, J.K.: Software Remodularization by Estimating Structural and Conceptual Relations among Classes and Using Hierarchical Clustering. In: Advanced Informatics for Computing Research. Communications in Computer and Information Science, vol 712. Springer, Singapore, 2017.

[Risi, 12] Risi, M., Scanniello, G., Tortora, G.: Using fold-in and fold-out in the architecture recovery of software systems, Formal Asp Comput 24(3):307–330, 2012.

[Sanner, 16] Sanner, J. M., Hadjadj-Aoul, Y., Ouzzif, M., Rubino, G.: Hierarchical clustering for an efficient controllers' placement in software defined networks, Global Information Infrastructure and Networking Symposium (GIIS), IEEE, 2016.

[Scanniello, 10] Scanniello, G., D'Amico, A., D'Amico, C., D'Amico, T.: Using the Kleinberg algorithm and Vector Space Model for software system clustering, In: Proceedings of international conference on program comprehension. IEEE Computer Society, 180–189, 2010.

[Scitovski, 14] Scitovski, R., and Sabo, K.: Analysis of the k-means algorithm in the case of data points occurring on the border of two or more clusters,' Knowl. Based Syst., 57, 1–7, Feb. 2014.

[Shtern, 09] Shtern, M., and Tzerpos, V.: Methods for selecting and improving software clustering algorithms. In Proceedings of 17th IEEE International Conference on Program Comprehension. IEEE, 248–252, 2009.

[STAFFORD, 01] Stafford, J.A, Richardson, D.J, Wolf, A.L.: Architecture-Level Dependence Analysis for Software Systems. Int. Journal of Soft. Eng. and Knowledge Eng. 11, 431-451, 2001.

[Swanson, 76] Swanson, E. B.: The dimensions of maintenance. In Proceedings of the 2nd International Conference on Software Engineering. 492–497, 1976.

[Tzortzis, 14] Tzortzis, G., Likas, A.: The MinMax k-means clustering algorithm, Pattern Recognit. , vol. 47, pp. 2505–2516, Jul. 2014.

[Vieira, 01] Vieira, M., Dias, M., Richardson, D.J.: Describing Dependencies in Component Access Points. In: Proceedings of 4th ICSE Workshop on CBSE, 2001.

[Wiggerts, 97] Wiggerts, T. A.: Using clustering algorithms in legacy systems remodularization, In Proceedings of the 4th Working Conference on Reverse Engineering. IEEE, 33–43, 1997.

[William, 92] Frakes, William B.: Information Retrieval Data Structures & Algorithms. Prentice-Hall, Inc. ISBN 0-13-463837-9, 1992.

[Wohlin, 00] Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wessl´en, A.: Experimentation in software engineering - an introduction, Kluwer, 2000.

[Wu, 05] Wu, J., Hassan, A. E., Holt, R. C.: Comparison of clustering algorithms in the context of software evolution. In: Proceedings of international conference on software maintenance. IEEE Computer Society, 525– 535, 2005.

[Yassin, 13] Yassin, W., et al.: Anomaly-based Intrusion Detection Through K-means Clustering and Naives Bayes Classification, Proc. of ICOCI 2013, 298-303.

[Yin, 03] Yin, R. K.: Case Study Research: Design and Methods. SAGE Publications, 3rd edition, 2003.

[Yu, 12] Yu, L., Mishra, A.: Experience in predicting fault-prone software modules using complexity metrics, Quality Technology & Quantitative Management 9 (4), 421-434, 2012.

[Zhong, 16] Zhong, L., He, J., Zhang, N., Zhang, P., Xia, J.: Software Evolution Information Driven Service-Oriented Software Clustering, IEEE International Congress on Big Data, 2016.