Evaluating Software Clustering Algorithms in the Context of Program Comprehension

Anas Mahmoud  
Computer Science and Engineering  
Mississippi State University  
Mississippi State, USA  
amm560@msstate.edu

Nan Niu  
Computer Science and Engineering  
Mississippi State University  
Mississippi State, USA  
niu@cse.msstate.edu

Abstract—We propose a novel approach for evaluating software clustering algorithms in the context of program comprehension. Based on the assumption that program comprehension is a task-driven activity, our approach utilizes interaction logs from previous maintenance sessions to automatically devise multiple comprehension-aware and task-sensitive decompositions of software systems. These decompositions are then used as authoritative figures to evaluate the effectiveness of various clustering algorithms. Our approach addresses several challenges associated with evaluating clustering algorithms externally using expert-driven authoritative decompositions. Such limitations include the subjectivity of human experts, the availability of such authoritative figures, and the decaying structure of software systems. We conduct an experimental analysis using two datasets, including an open-source system and a proprietary system, to test the applicability of our approach and validate our research claims.

Index Terms—Program comprehension, maintenance, software clustering.

I. INTRODUCTION

Program comprehension is one of the most frequently performed activities in software maintenance [1]. Developers working on maintenance tasks spend around 60% of their time comprehending the system [2, 3]. To reduce this effort, several program comprehension techniques adopt clustering to divide the system into smaller, more focused and easier to comprehend subsystems [4, 5, 6]. A full comprehension can then be achieved by following a top-down, a bottom-up, or a combined strategy [7, 8].

Several clustering techniques for aiding program comprehension and software maintenance have been proposed in the literature [4, 5, 9, 10, 11]. A majority of these techniques rely on a well-defined fitness function that captures the twin objectives of high cohesion and low coupling to help to determine the boundaries between clusters. Cohesion measures how tightly the internal elements of a cluster are related to one another [12], and coupling measures the strength of interconnection between modules [13]. Subsystems with internal high cohesion and external low coupling are easier to comprehend and maintain [14]. The main assumption driving this kind of analysis is that the system is well-organized into cohesive subsystems that are loosely interconnected [15, 16]. However, there is no guarantee that original developers of the system have followed high-cohesion and low-coupling principles when writing initial code. This problem becomes obvious in evolving software systems as the modular structure of the system tends to degrade over time [17, 18]. In addition, there is no standard method for quantifying coupling and cohesion in a software system [16, 19, 20]. This raises several consistency concerns when evaluating clustering algorithms, and often leads to complications in implementing working prototypes driven by optimizing an objective function of coupling and cohesion [4, 21, 22, 23].

To overcome these limitations, an external evaluation approach using an authoritative decomposition, or a gold standard, of the system can be used [4, 15, 24]. Such a decomposition is often generated by domain experts. The effectiveness of a particular clustering algorithm is computed based on the resemblance between its output and the authoritative decomposition. The resemblance can be quantified using several metrics, such as MoJo [25] or KE [26]. This approach avoids several complications associated with internal measures by focusing on the final output of the clustering algorithm rather than internal structural attributes of the software system. However, such evaluation mechanism has several limitations, for example, an agreed upon architecture or an authoritative decomposition of a complex software system often does not exist. Instead, ad-hoc figures are typically produced by system experts to carry out the evaluation process [9, 26, 27]. This raises several subjectivity concerns as different clustering objectives create different decompositions [28], which might lead to biased results toward the expert’s objective. Another concern is the fact that these decompositions might not reflect current state of the system. Software systems evolve constantly over time and maintaining an up-to-date authoritative figure can become a tedious and error-prone task [21].

To address these concerns, in this paper we propose a novel approach, which utilizes transaction logs of previous maintenance sessions, for evaluating software clustering algorithms. Our main assumption is that program comprehension is a task-driven activity, thus the information embedded in maintenance transaction logs can be utilized to automatically devise multiple comprehension-aware decompositions of the system. State-of-the-art evaluation mechanisms are then used to evaluate the effectiveness of different clustering techniques in recovering these decompositions [25]. Our main contribution is an automatic and task-sensitive approach for evaluating software clustering algorithms in the context of program comprehension. Our
Objectives are to address several limitations associated with current evaluation mechanisms, enable other kinds of clustering analysis, such as stability analysis [29], and address several issues associated with maintaining long-term large software systems, such as the problems of partial information and software aging [17]. We conduct an experimental analysis, using an open-source and a proprietary systems, to test the usefulness and the scope of applicability of our approach and validate our research assumptions.

The rest of the paper is organized as follows. Section II describes our approach. Section III presents our experimental settings, including our datasets, experimental analysis and results. Section IV discusses potential threats to our study validity. Section V reviews seminal threats to our approach and validate our research assumptions. Finally, Section VI concludes the paper with a summary and some directions for future work.

II. ASSUMPTIONS AND METHODOLOGY

Program comprehension is a task-driven activity [30, 31]. The process starts from the change request. A change request can range from a simple bug fix, to a more complicated modification in the system behavior, such as adding a new feature or altering an existing one. Developers then use information from the request’s description to locate parts of the code related to the task, gradually building a mental representation of the problem. In other words, developers follow a partial, pragmatic, as-needed rather than a systematic comprehension strategy to comprehended the task [32, 33, 34]. A developer accommodates a change by performing a sequence of interactions with the system, including viewing and editing certain files. A log of these multiple interactions captures the mental model of the developer while comprehending that part of the system, or represents a self-contained comprehension unit [32, 2, 35]. Our main assumption in this paper is that a sample of these units over a certain codebase represents a comprehension-aware decomposition of that code. Based on that, we devise an algorithm that utilizes maintenance transaction logs of software systems to produce multiple authoritative decompositions for these systems.

To explain our approach, Figure 1 shows a number of maintenance transaction logs over a hypothetical system’s code base. The system has 12 source code files \{1, 2, …, 12\} and transaction logs of six successfully resolved issues including \{A, B, …, F\}. We define the following properties for each log:

- **Size ($Sz$):** Number of files the log contains.
- **Overlap index ($OI$):** Number of other transaction logs the log shares files (overlap) with.
- **Overlap ratio ($OR$):** Percentage of files the log shares with other transaction logs.

For example, transaction log $E$ in Figure 1 contains five files ($Sz = 5$). It overlaps with two other transaction logs, $F$ and $D$ ($OI = 2$) and share three files with these two logs ($OR = 6$).

The problem now is to choose combinations of these transaction logs that can later be used as authoritative decompositions over the system’s code base.

To simplify our analysis, at the current stage of our research we enforce a non-overlapping constraint on the generated decompositions. Non-overlapping means that each file in the authoritative decomposition can belong to at most one cluster. Formally, using set theory, assuming our code base represents a universal set ($U$), and each log ($l$) represents a subset of $U$ that contains a certain number of files. The main objective is to identify combinations of mutually exclusive subsets over $U$, such that $\forall l_i \cap l_j = \emptyset, i \neq j$.

This problem can be treated as an instance of the constraint satisfaction problem (CSP) [36]. A constraint satisfaction problem is defined by a set of variables $\langle V_1, V_2, …, V_n \rangle$ and a set of constraints, $\langle C_1, C_2, …, C_m \rangle$. Each variable $V_i$ has a nonempty domain $D_i$ of possible values. Each constraint $C_i$ involves some subset of the variables, and specifies the allowable combinations of values for that subset. In our case the variables are the transaction logs $\langle A, B, C, D, E, F \rangle$, the domain of each variable is the set $\langle Sz, OI, OR \rangle$. The formulation of the problem as a CSP gives us the flexibility to enforce or relax various constraints to fit different settings. For example, instead of enforcing a non-overlapping constraint on the allowable combinations ($OR = 0$), different overlapping ratios can be considered ($OR < \alpha$). Similarly, a different overlap index ($OI < \beta$), or a size ($Sz > \gamma$) constraint can also be enforced.

![Fig. 1. Sample system with 12 source code files and 6 transaction logs](image)

To find a consistent assignment that does not violate any constraints, we represent the structure in Figure 1 as a constraint graph $G$. Each node in the graph ($u$) represents a transaction log. Currently, only a binary overlap constraint is shown in the graph (logs either overlap or do not). An edge indicates overlapping between the two nodes (i.e. two transaction logs share some files). The graph is shown in Figure 2.

![Fig. 2. Constraint graph for system in Fig. 1](image)

To solve our problem, a breadth-first search can be applied as follows: For each node $u$ in the graph $G$, the procedure tries to find all the paths through the graph that starts with $u$ and goes through the largest possible number of non-overlapping nodes.
TABLE I. GENERATED PATHS FOR GRAPH G

<table>
<thead>
<tr>
<th>Figure ID</th>
<th>Logs</th>
<th>Files</th>
<th>Coverage Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A, C, F</td>
<td>1,3,5,4,8,11</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>A, E, C</td>
<td>1,3,5,8,9,10,11,12,4</td>
<td>75%</td>
</tr>
<tr>
<td>3</td>
<td>A, D, F</td>
<td>1,3,5,4,8,11,12</td>
<td>58%</td>
</tr>
<tr>
<td>4</td>
<td>B, E, C</td>
<td>1,2,6,7,8,9,10,11,12,4</td>
<td>83%</td>
</tr>
<tr>
<td>5</td>
<td>B, C, F</td>
<td>1,2,6,7,4,11,12</td>
<td>58%</td>
</tr>
<tr>
<td>6</td>
<td>B, D, F</td>
<td>1,2,6,7,4,8,11,12</td>
<td>75%</td>
</tr>
</tbody>
</table>

For example, starting from A, there are four valid nodes to add to the path \{C, E, F, D\}. B cannot be chosen because it overlaps with A \(i.e., \text{violates the} \ R \text{O} = 0 \text{constraint}\). Assuming C was chosen, the next step would be to find all the nodes that do not overlap with A and C. In our example this set includes \{E, F\}. The procedure continues recursively until no more non-overlapping points can be found. In other words, a complete assignment is achieved and each log has appeared in at least one set. Since the order of the nodes in each path does not matter, redundant paths are removed. The implementation of this algorithm is shown in Figure 3. This algorithm has a \(O(n \log n)\) complexity. However, since all transaction logs are locally available on disc, processing time is negligible.

```plaintext
PROCEDURE Generate Decompositions(G)
    List Decompositions = ∅.
    Decomposition Figure;
    FOREACH u ∈ G
        G = G - u
        Figure.Add(u)
        AddNode(Figure, G)
    RETURN Decompositions

PROCEDURE AddNode(Figure, G)
    IF(G = ∅)
        IF NOT (Decompositions.Contains(Figure))
            Decompositions.Add(Figure);
    ELSE
        FOREACH u ∈ G
            IF NOT Figure.Overlaps(u)
                Figure.Add(u)
                G = G - u;
                AddNode(Figure, G)
        ELSE
            G = G - u

Fig. 3. Authoritative decompositions generation procedure
```

For our graph example \(G\) in Figure 2, the solution set includes the following sets (allowable combinations)\{\{A, C, F\},\{A, E, C\},\{A, D, F\},\{B, E, C\},\{B, C, F\},\{B, D, F\}\}. These sets are shown in Table I. For each set, the associate partial code base is also shown. Separately, each set is considered a task-sensitive authoritative decomposition of certain parts of the system. The coverage percentage shows the percentage of the system files included in each of the partial decompositions. Next we show how these partial figures can be used to evaluate different clustering algorithms.

III. EXPERIMENTAL EVALUATION

In this section we conduct an experimental analysis, using two datasets generated from an open-source system and a proprietary system, to validate our research assumptions and claims and test the applicability of our approach in evaluating several clustering algorithms.

A. Subject Systems

We analyze the maintenance transaction logs of two software systems. These transactions are available through the each system’s issue tracking database. These systems include:

**Mylyn:** is an open-source, task-focused interface for software developers using the Eclipse platform. Mylyn has been chosen because it fits the description of a large-scale, not-so-easy to comprehend complex system. In this paper we use the latest stable release of Mylyn (version 3.7.1) which contains 2405 Java files. Among Mylyn’s programming logs, 893 sessions modify the files from the latest version. These sessions are contributed by 31 developers on 751 software change tasks. Every programming session is linked with its corresponding task \(e.g., \text{a bug report, an enhancement request, etc.}\) in the issue tracking system. These files are available for download through the project’s issue tracking system as XML files at (http://wiki.eclipse.org/Mylyn_Bugzilla_Connector).

**WDS:** is a software-intensive platform that provides technological solutions for service delivery and workforce development in a specific region of the United States. In order to honor confidentiality agreements, we use the pseudonym “WDS” to refer to the system. WDS has been deployed for almost a decade and is developed using Java technologies. WDS employs a commercial state-of-the-practice issue tracking system to manage system evolution information. Every time a change is requested, a maintenance ticket is opened for that request. The system keeps track of all the developers’ actions while working on the change \(e.g., \text{edit, view, .. etc.}\). Current version of the system has 521 source code files and 157 maintenance sessions contributed by 18 developers on 139 tasks. These transaction logs are also available for download as XML files through the project’s issue tracking system.

A filtering process is applied to select valid transaction logs in both subject systems. We define a valid transaction log as the log of an issue that has been successfully resolved \(\text{a fixed bug, or a successfully added or modified feature}\), and all of its code files \(\text{filename and version}\) are available on disk. An XML reader is developed to deal with the large number of issues available for our subject systems. Results of the filtering process, in addition to a description of both of our datasets, are shown in Table II.

<table>
<thead>
<tr>
<th># of files</th>
<th>Mylyn</th>
<th>WDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>213.9K</td>
<td>44.6K</td>
</tr>
<tr>
<td>Lines of comments</td>
<td>69.6K</td>
<td>10.7K</td>
</tr>
<tr>
<td># of transaction logs</td>
<td>893</td>
<td>157</td>
</tr>
<tr>
<td># of tasks</td>
<td>751</td>
<td>139</td>
</tr>
<tr>
<td># of developers</td>
<td>31</td>
<td>18</td>
</tr>
<tr>
<td># of valid logs</td>
<td>351</td>
<td>141</td>
</tr>
</tbody>
</table>
B. Task-Sensitivity of Directory Structure

We start our analysis by conducting a task-sensitivity analysis over our systems’ directory structures. Several studies adopt the directory structure of the system as an authoritative decomposition. The main assumption is that original developers (domain experts) organized the system’s files in folders and subfolders in such a way that facilitates future maintenance tasks [37, 38]. Therefore, such a structure can be considered as an authoritative decomposition to compare the performance of different clustering algorithms [4]. In our analysis, we claim that authoritative decompositions devised based on such information are not necessarily task-sensitive, thus they are not sufficient to evaluate clustering algorithms in the context of program comprehension. To validate this claim, we compare the transaction logs in both of our systems to their directory structures. Our objective is to measure how well such structures overlap with the actual maintenance data.

To carry out this analysis, we use the Precision measure proposed by Anquetil and Lethbridge [24]. Assuming \( A \) and \( B \) were two decompositions of a software system, precision calculates the percentage of intra-pairs in decomposition \( A \) that are also intra-pairs in decomposition \( B \). In particular, for each maintenance transaction log, we measure the percentage pairs of files that also appear in the same folder in the directory structure of the software system. It is important to note that when working with a directory structure, precision increases with higher levels at the system hierarchy, until eventually the whole system is treated as one folder, in that case, precision = 1. However, that decomposition is practically useless. Therefore, when adopting the directory structure of the system as its authoritative decomposition, it is important to decide on the cut-off point in the system’s directory tree.

The results of precision analysis for both systems are shown in Table III. Results are reported at three levels (cut-off points), where level 1 is the finest granularity level at the system directory structure. At that level folders contain only source code files with no subfolders. The 2\(^{nd}\) level is where system folders contain at least one subfolder from level 1. The highest level, the whole system is treated as one folder. The longest branch in Mylyn’s directory structure tree has a length of 6, while in WDS it has a length of 5.

Table III shows that, even at higher directory level, the precision values are still relatively low, indicating that the majority of tasks are actually spread over multiple folders, covering a wide range of the system’s directory structure. Therefore, authoritative decompositions produced based on the system's directory structure are not necessarily task-sensitive. This might lead to misleading results when evaluating clustering algorithms in the context of software maintenance or program comprehension.

C. Clustering Algorithms

We test the applicability of our approach over multiple clustering algorithms. These algorithms cover a wide spectrum of clustering methods available in the software clustering literature including:

- **Partitional clustering:**
  We use a basic, randomized k-means algorithm as a representative of this category of centroid-based partitional clustering algorithms. Centroid-based refers to the family of algorithms that use an initial number of centers to represent and group input data.

- **Hierarchical Agglomerative Clustering (HAC):**
  Anquetil and Lethbridge [24], proposed a hierarchical clustering algorithm suite, which offers a selection of association and distance coefficients as well as update rules to cluster software systems. We use three variants of HAC in our analysis. These well-known algorithms, underlying many software architecture recovery techniques, can be distinguished based on their update rules as follows:
  - Single Linkage (SL): \( M(A, B) = \min \{d(a, b) : a \in A, b \in B \} \)
  - Complete Linkage (CL): \( M(A, B) = \max \{d(a, b) : a \in A, b \in B \} \)
  - Average Linkage (AL): \( M(A, B) = \frac{1}{\text{\#A} \times \text{\#B}} \sum_{a \in A} \sum_{b \in B} d(a, b) \)
  where \( d(a, b) \) is the distance between data objects \( a \) and \( b \), and \( M(A, B) \) defines the linkage (merging) criteria or clusters \( A \) and \( B \). For example, SL merges the two clusters with the smallest minimum pairwise distance.

- **Comprehension-based clustering:**
  We use ACDC as a representative of this category of algorithms. ACDC is a comprehension-driven clustering algorithm based on the patterns frequently appeared in manual decompositions of large-scale software systems [4]. A key contribution is bounded cardinality that ensures a reasonable cluster size to ease comprehension; thus each resulting cluster of ACDC contains 5 to 20 entities. An implementation of ACDC along with other related tools are available at (http://www.cse.yorku.ca/~bil/downloads/).

A main question to answer when clustering software systems is how to define the distance among software artifacts. In the literature, either formal or nonformal information can be utilized for calculating distance in source code [24, 27]. Formal information refers to the structural information of the source code, or features that have direct impact on the system’s behavior. For example the functions’ calls, parameters definitions, or inheritance relations in object oriented code. In contrast, nonformal information refers to the knowledge embedded in the taxonomy of the code, such as the parameters and functions names and the code comments, or features that do not have influence on the code operation. Investigation of both sources of information revealed that nonformal information can be as good as formal ones [10, 24, 27]. Nonformal features are less redundant and more abstract and easier to extract, interpret and understand by humans [24]. Extracting formal information, however, can be more
complicated as parsing rules need to be applied. This process gets even more complicated when only partial information is available. Therefore, we adopt nonformal features in our analysis.

The next question to answer is what similarity measures should be used to quantify the distance. Several similarity measures have been proposed and investigated in the software clustering literature [39]. Anquetil and Lethbridge [24] performed a comprehensive investigation of software clustering algorithms. Their experiment revealed that Jaccard and Sorenson-Dice similarity measures give the best results, both in internal and external evaluation. A comparison between similarity coefficients by Davey and Burd [15] also revealed that Jaccard and Sorenson-Dice coefficients yield to best results. These results were also confirmed by Maqbool and Babri [27], who found that overall clustering results are expected to be better for members of the Jaccard family. In general, there seems to be a consensus on Jaccard-like metrics to compute pairwise similarity between documents. Therefore, in this paper we adopt basic Jaccard as our similarity measure. Mathematically, using Jaccard coefficient, similarity between two artifacts $X$ and $Y$, given that both $X$ and $Y$ are represented as two sets of keywords, can be described as follows:

$$
Sim(X, Y) = \frac{a}{a + b + c}
$$

(1)

where $a = |X \cap Y|$, $b = |X / Y|$, and $c = |Y / X|$ [24].

D. Execution

We applied the procedure in Figure 3 over the selected transaction logs in our datasets. In addition to the non-overlapping constraint, we enforced the following size ($Sz$) constraint over each generated decomposition $S$ over the source code base $U$:

$$
\forall S \in U, \left( \sum_{i=1}^{k} S_{zi} \right) > \alpha
$$

(2)

where $k$ is the number of transaction logs in each decomposition. We set $\alpha = 100$ for WDS and $\alpha = 400$ for Mylyn. This constraint is enforced to ensure that the generated decompositions are of acceptable size, also to ensure a high ratio of the systems’ files appears in each of the generated decompositions. In particular, The size constraint threshold value ($\alpha$) in both systems was gradually increased, using a hill climbing strategy, until a maximum average coverage ratio was produced in both systems (Mylyn =83%, WDS = 89%). In addition, a size constraint is enforced so that each log contains at least $[5-45]$ files. The decision to consider only this size range is made to avoid cluster extremity. In particular, we want to avoid a situation where the majority of files are grouped into one or few large clusters (black hole) or the majority of clusters are singletons (dust cloud) [15, 40]. The effect of enforcing these constraints on both datasets is shown in Table IV. For each dataset, the table shows, the number of produced authoritative decompositions, the average size of logs, and the average percentage of the system’s files that are included in each decomposition.

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Mylyn</th>
<th>WDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum S_{zi} &gt; 400$</td>
<td>$\alpha = 0, OI = 0$</td>
<td>$\alpha = 0, OI = 0$</td>
</tr>
<tr>
<td>No. decompositions</td>
<td>42</td>
<td>27</td>
</tr>
<tr>
<td>Average Size</td>
<td>37</td>
<td>17</td>
</tr>
<tr>
<td>Average Coverage</td>
<td>83%</td>
<td>89%</td>
</tr>
</tbody>
</table>

The set of source code files included in each decomposition are clustered using the different clustering algorithms proposed earlier. Since we are enforcing a non-overlapping constraint, we restrict our discussion to mutually exclusive clustering, which allows each file to be assigned to one and only one cluster. MoJoFM is then used to evaluate the average effectiveness of the different clustering algorithms in recovering the associated decompositions [25]. MoJoFM estimates the distance between two decompositions of a software system based on the number of Move and Join operations to transform one to the other [41]. MoJoFM value of 1 means that the algorithm successfully recovered the exact authoritative decomposition. A value of 0 means that the algorithm produced a decomposition that is completely different from the authoritative decomposition. Since MoJoFM is non-symmetric, similarity between clustering algorithms $A$ and $B$ is defined as:

$$
MoJoFM_{A,B} = \max(MoJoFM(A,B), MoJoFM(B,A))
$$

(3)

In our approach, since only non-overlapping clusters are produced, a partition of the system instead of a hierarchy is needed. In hierarchical clustering algorithm, finding the appropriate height at which to stop clustering is not a trivial task [24]. Such a cut-off point highly depends on the nature of the problem and the objective of clustering [15]. In some cases, a static cut-off point is used [37, 28]. In some other cases, different cut-off points are taken, and either the performance is averaged over all the cut-off points or the best value is kept [24, 27, 42, 43]. In this paper we adopt the following strategy: since we know the size range of our transaction logs $Sz = [5, 46]$, a transaction log with $n$ number of files can have a maximum of $\frac{n}{5}$ number of clusters, assuming all transaction logs in the decomposition have the minimum size ($\forall l_i \in S, Sz_i = 5$), and a minimum of $\frac{n}{45}$ clusters where ($\forall l_i \in S, Sz_i = 45$). For each source code base to cluster, we start from the smallest possible number of clusters, increasing 5 cluster at the time until the largest possible number of clusters is hit. At each cut-off value, we calculate MoJoFM, eventually the best MoJoFM is kept. These values are then averaged over all the decompositions. Same approach is used to select $k$ at the beginning of $k$-means.

MoJoFM results of comparing the outcome of different clustering algorithms to the generated authoritative decompositions of Mylyn and WDS are shown in Figures 4, 5 respectively. Pairwise analysis-of-variance results are shown in Table V. We used the 0.05 alpha level ($\alpha=0.05$) to test the statistical significance of the results. Significant results in the table are shown in italic font.
TABLE V. ANALYSIS OF VARIANCE

<table>
<thead>
<tr>
<th></th>
<th>Mylyn</th>
<th>WDS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$F$</td>
<td>$p$</td>
</tr>
<tr>
<td></td>
<td>$p$</td>
<td></td>
</tr>
<tr>
<td>SL x CL</td>
<td>805.314</td>
<td>.00</td>
</tr>
<tr>
<td>SL x WA</td>
<td>416.464</td>
<td>.00</td>
</tr>
<tr>
<td>SL x K-mean</td>
<td>1.956</td>
<td>.196</td>
</tr>
<tr>
<td>SL x ACDC</td>
<td>1.107</td>
<td>.299</td>
</tr>
<tr>
<td>CL x WA</td>
<td>20.916</td>
<td>.00</td>
</tr>
<tr>
<td>CL x K-mean</td>
<td>461.105</td>
<td>.00</td>
</tr>
<tr>
<td>CL x ACDC</td>
<td>483.384</td>
<td>.00</td>
</tr>
<tr>
<td>WA x K-mean</td>
<td>607.66</td>
<td>.00</td>
</tr>
<tr>
<td>WA x ACDC</td>
<td>542.732</td>
<td>.00</td>
</tr>
<tr>
<td>k-means x ACDC</td>
<td>9.09</td>
<td>.05</td>
</tr>
</tbody>
</table>

Fig. 4. SL, CL, AL, k-means and ACDC performance in Mylyn

Fig. 5. SL, CL, AL, k-means and ACDC performance in WDS

E. Clustering Results Analysis

We start our analysis by looking at k-means algorithm results. To mitigate the effect of randomization on k-means performance, for each authoritative decomposition in both datasets, k-means is executed 100 times for each cut-off value and best values for MoJoFM are kept [37]. Overall, the results show that, even with the optimization step, in both datasets, k-means fails to produce any meaningful results in recovering authoritative decompositions. We proceed our analysis by looking the performance of the HAC family. The results show that in both datasets Complete Linkage (CL) achieves the best results in recovering the generated authoritative decompositions, significantly outperforming all other algorithms. Average Linkage (AL) comes in second with a relatively good performance in both systems. In fact, in Mylyn dataset, AL is only slightly out performed by CL, however, the difference in the performance is statically insignificant. The third algorithm in the HAC family, Single Linkage (SL), could not match the other two algorithms performance, failing to beat k-means algorithm in Mylyn. Finally, the results show that the comprehension-based algorithm ACDC surprisingly fails in recovering authoritative decompositions, achieving a comparable weak performance to k-means. ACDC performs slightly better in Mylyn, however, the difference in the performance is statistically insignificant.

In an attempt to explain the difference in performance between these different algorithms, we take a look at their internal operation. Because of its update rule, where the smallest distance is used as the new distance, whenever SL is used, bigger clusters tend to be grouped together rather than incorporating singletons. Therefore, SL tends to create a small number of large, isolated clusters. In constant, CL, uses the largest distance as a representative for the newly formed clusters, thus pushes clusters apart, creating lots of small but highly cohesive clusters, hence it tends to do better in systems that exhibit high cohesion. To relate these observations to our datasets we perform a cohesion analysis over both of our systems. The average cohesion of each authoritative decomposition is calculated as follows [44]:

$$CH = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{m_i} \sum_{a,b \in c_i} \text{sim}(a, b)$$  \hspace{1cm} (4)$$

where $k$ is number of clusters in the decomposition, $a$ and $b$ are files in cluster $i$, $\text{sim}(a, b)$ is the Jaccard similarity between the two files, and $m_i$ is the normalization factor for cluster $i$ calculated as $\binom{a}{2}$, where $n$ is the cluster size. Results are shown in Table VI. The table shows the average cohesion of decompositions produced by the various algorithms and the average cohesion of the authoritative decompositions.

It is important to point out here that coupling and cohesion are inherently ordinal scale metrics, so there is no clear definition of which absolute value represents a high or a low cohesion; however, it is obvious that CL in both datasets produces the most cohesive clusters, achieving relatively comparable characteristics to the authoritative decompositions. Table VI also shows that WDS authoritative decompositions exhibit higher cohesion than Mylyn decompositions, which explains the better results CL achieves in this dataset. The better performance of SL in WDS can be explained based on the observation that SL tends to work better for smaller size systems [45]. In addition, the table shows the difference in cohesion between both k-means and ACDC output and the authoritative decompositions; explaining the poor performance of these two algorithms in both systems.
Overall, our results seem consistent with previous work in related domain which showed that, in relatively large systems, CL gives the best results in terms of recovering authoritative decompositions [15, 24]. In addition, Average Linkage is always closer in its performance to Complete Linkage than Single Linkage [15]. For k-means, it seems that the random decisions taken by the algorithm during clustering affect the quality of its performance negatively. This is clearly shown in the low cohesion values achieved by k-means. This kind of behavior is actually expected due to the inherent hierarchical nature of software systems which makes hierarchical clustering algorithms more suitable for source code clustering than partitional algorithms.

The weak performance of ACDC in the context of program comprehension can be attributed to the Bounded Cardinality design decision adopted in its internal operation. As mentioned earlier, ACDC limits the size of the cluster to contain no more than 20 files. This constraint is based on the assumption that smaller clusters are more manageable and easier to understand. However, even though a size range of [5-45] is used in filtering transaction logs in both of our systems, there is no guarantee that cardinality will be bounded around 20 files per cluster. To further investigate this we calculate the average cardinality (cluster-size) for the authoritative decompositions in both of our datasets. These values are compared with average cluster size produced by ACDC. Results show that ACDC relatively kept its bounded cardinality in both systems, producing an average cluster size of <21, 17.6> for Mylyn and WDS respectively, while the average cardinality in the generated authoritative decompositions is <37, 26> for Mylyn and WDS respectively. Obviously, the up-to-20 files cardinality does not serve comprehension very well. ACDC produces too fine-grained clusters (containing large number of clusters) than their corresponding authoritative decompositions.

### F. Stability Analysis

Stability is viewed as an indication whether the model proposed by some clustering algorithm fits the data or not [29]. Bun-Hur et. al. described the notion of stability as "repeatedly sample data points and apply the clustering algorithm, then a good algorithm should produce clusterings that do not vary much from one sample to another" [46]. In other words, a clustering algorithm is considered stable if its output is not grossly effected by slight modification in its input. Tzerpos and Holt stated that to be practically stable, a clustering algorithm has to be consistent over several versions of the system that are several days apart [46]. However, keeping track of a system structure on a daily basis is impractical, which limits carrying out stability analysis in practice [46, 47, 48]. Our approach overcomes this problem by enforcing a time constraint on the procedure in Figure 3. This provides an efficient and more practical way to measure stability as several time-sensitive figures can be produced at different time intervals set by the time constraint. In the literature, stability has been formally defined in various ways depending on the context [37, 49]. In our analysis we assess practical stability using the following formula:

\[
\text{Stab}(C) = \Delta(C(A_{d1}), C(A_{d2}))
\]

where \(C\) is the clustering algorithm under investigation, \(A_{d1}\) and \(A_{d2}\) are the MoJoFM values of applying \(C\) to the two authoritative decompositions samples produced at times \(d_1\) and \(d_2\) respectively, and \(\Delta\) is the statistical similarity between the two samples. \(C\) is considered stable if it does not produce significantly different results over authoritative decompositions from different time periods.

To assess the stability of the different clustering algorithms, we devise multiple time-sensitive groups of authoritative decompositions for each of our systems over three time intervals. These intervals are selected based on the date of committing each transaction log. We then execute our clustering algorithms over the different groups of generated decompositions and monitor the changes in MoJoFM values. ANOVA (analysis of variance) is then used to detect significant changes in the performance. Results are shown in Figures 6 and 7 and Table VII. Analysis of variance shows that both CL, and AL achieve a stable performance across decompositions from different time intervals, SL is not stable in Mylyn. These results seem to contradict previous theoretical predictions which suggest that SL is more stable [47, 50]. Results also show that both ACDC and k-means are unstable in both systems, producing significantly different results over the different decompositions groups sampled over different time intervals.

In general, our evaluation approach revealed that, with the exception of SL, hierarchical clustering algorithms seem to do better than comprehension-based and partitional clustering algorithms in the context of program comprehension. This can be explained based on the characteristics of the comprehension-aware decompositions derived from the characteristics of transaction logs produced by developers. These logs are rarely bounded in their cardinality: some maintenance tasks require changing less than 5 files in the system, while other tasks might require changing hundreds of files. Another attribute is the high cohesion of these logs. High cohesion within an entity means that there is a strong inter-relationship between its constituent parts. In addition, in terms of stability, the HAC algorithms are found to be the most stable, thus they are potentially more practical to be integrated in tools that are targeted toward program comprehension.
In summary, and based on our analysis results, the potential benefits of our approach can be described as follows:

- Often only a single decomposition of the system is available to evaluate clustering algorithms. Such decomposition is usually devised by a domain expert. Using only a single data point is not sufficient to conduct a sound statistical analysis or draw valid conclusions [28]. As shown in our analysis, by enforcing and relaxing various constrains, our approach can automatically devise a large number of authoritative decompositions for the system, which enables a more rigorous analysis when externally evaluating clustering algorithms as more data points are generated [28].

- Our approach adopts a unique definition of what makes a cluster. Our definition gets its authoritativeness from the ground-truth experience of developers rather than the subjectivity of human experts or the directory structure of the system. This eliminates any possible experimental bias. In addition, it produces task-sensitive decompositions that are suitable to evaluating software clustering algorithms in the context of software maintenance and program comprehension.

- Since it deals with partial information, our approach does not rely on the internal structure of the system. The system does not have to be complete or to fully compile for our approach to work.

- Our approach deals with the problem of decaying software structure [17]. As software systems evolve, more up-to-date authoritative decompositions, using more recent maintenance transaction logs can be devised. Clustering algorithms can then be recalibrated using the newly generated decompositions. Enforcing a time constraint over our approach also enables other kinds of analysis such as stability analysis [47].

IV. Threats to Validity

Several factors can affect the validity of our study. Construct validity is the degree to which the variables accurately measure the concepts they purport to measure [51]. In our experiment, a threat might stem from the fact that only MoJoFM was used to measure the effectiveness of the various clustering algorithms. Other related measures such as Precision and Recall [24] and KE [26] could have also been used. However, MoJoFM has been used extensively in software clustering research, and we believe it provides a sufficient indication of the various clustering algorithms’ effectiveness.

Threats to external validity impact the generalizability of results. In particular, the results of this study might not generalize beyond the underlying experimental settings [51]. A major threat to the external validity comes from the fact that only two subject systems were used in our experiment. However, both of our subject systems exhibit industrial strength properties, such as the large size and the availability of maintenance transaction logs. Another factor that might affect the generalizability of our results is specific design decisions made and measures and heuristics used during the experimental analysis. For example, we used nonformal information and Jaccard coefficient to implement our clustering algorithms. Also different filtering and optimization criteria were used in selecting our transaction logs and building our authoritative decompositions. However, identifying the best settings for effectively integrating clustering in a working application is viewed as an NP-complete optimization problem [52]. For this reason, we were compelled to make these decisions about the general clustering features in order to achieve an acceptable approximation and keep analysis complexity under control. Also, proper justification from the literature was provided to support our decisions.

V. Related Work

Our approach tries to address several issues related to current state of practice in evaluating software clustering algorithms. A variety of evaluation frameworks have been proposed in the literature to evaluate the quality of clustering techniques [26, 53]. Closer to our work comes the work by Shtern and Tzerpos [28] who introduced a novel approach that facilitates a more accurate evaluation of the effectiveness of different clustering algorithms. Their approach, known as LimSim, produces large number of MojoFM values by comparing clustering results with a large number of simulated authoritative decompositions. LimSim works by applying a series of well-defined modifications on the system’s original Module Dependency Graph (MDG) and its corresponding authoritative decomposition to obtain new MDGs and authoritative decompositions that are significantly different from the originals. These figures are then used to externally
evaluate various clustering algorithms. Our work can be distinguished from this work in the fact that no initial MDG or authoritative decomposition is needed to generate other decompositions.

In the approach presented in this paper, historical transaction logs of previous maintenance sessions are utilized to produce authoritative decompositions of the system. The value of mining historical development information to aid program comprehension and software maintenance tasks has been outlined clearly in related literature. Tilley [54] categorized leveraging corporate knowledge and experience as a main source for supporting comprehension. Tjoortjis and Layzell [32] reported that knowledge derived from past maintenance as the most useful pieces of information to facilitate code comprehension when maintaining software. Beyer and Noack [9] introduced an approach for identifying subsystem candidate clusters by computing a layout that reveals clusters of frequently co-changed artifacts. These layouts are expected to facilitate a more effective comprehension process.

Historical development knowledge has also been utilized in designing several tools to aid program comprehension [7, 55]. For example, Cubranic and Murphy proposed Hipikat [56], a tool that uses information stored in a project’s historical archives to recommends artifacts that are relevant to a task that a newcomer is trying to perform. Robillard and Dagenais [57] proposed a technique that scans the change history of a software system to determine if there exists any change clusters that overlap with a set of elements of interest to a developer. The general assumption is that a developer working on a task related to a change cluster can potentially benefit from knowledge about the set of elements in the cluster. Zimmermann et al. [35] introduced ROSE, a tool that operates on the version history to guide the programmer along related changes. By mining association rules from change transactions, ROSE suggests further changes to be made and warns about missing ones. The approach proposed in this paper shares similar assumptions with related work about the value of mining historical change data, as it utilizes historical change information to guide the process of evaluating other well-defined clustering algorithms.

VI. CONCLUSIONS

Many software clustering algorithms have been proposed in the literature. These algorithms are designed to address the needs of several software engineering tasks such as program comprehension, software maintenance, and reverse engineering activities. However, less attention has been paid to developing effective methods to evaluate such algorithms. In most cases, such algorithms are evaluated through case studies and based on the researchers’ objectives. As part of the effort to bridge this gap, in this paper, we proposed a novel approach for evaluating clustering algorithms in the context of program comprehension. Our approach uses transaction logs of previous maintenance sessions to produce multiple task-sensitive and comprehension-aware decompositions of the system. The approach is described as an instant of the Constraint Satisfaction Problem (CSP), where certain constraints can be enforced or relaxed to produce a large number of authoritative decompositions. Formulating the problem as a CSP gives us the flexibility to generate decompositions that fit different contexts for software cluster evaluation. For instance, a date constraint can be enforced to generate more up to date figures of the system and enable other kinds of analysis such as stability analysis, a size constraint helps to tune clustering algorithms by generating acceptable size clusters.

Two datasets were used to evaluate and examine the applicability of our approach. Clustering algorithms from different families including partitional, hierarchical agglomerative and comprehension-based clustering were evaluated in our experimental analysis. The results showed that, with the exception of Single Linkage (SL), hierarchical algorithms are the most successful in recovering comprehension decompositions. The results also showed that the comprehension-based algorithm ACDC was not as successful as the hierarchical ones. Stability analysis showed that hierarchical algorithms are more stable than other clustering algorithms. Our approach not only facilitates a more robust way for evaluating software clustering algorithms, but also provides insights into these algorithms operation and addresses several issues related to software aging and partial information. Future work will be focused on conducting more experiments using industrial-size datasets to evaluate the practicality of our approach and the feasibility of integrating our findings in modern development environments.

ACKNOWLEDGMENT

We would like to thank the partner company for the generous support of our research. This work is supported by the U.S. NSF (National Science Foundation) Grant CCF1238336.

REFERENCES


