Factbase and Decomposition Generation

Mark Shtern and Vassilios Tzerpos
York University
Toronto, Ontario, Canada
{mark,bil}@cse.yorku.ca

Abstract

The software maintenance research community has developed a large number of approaches that can help maintainers understand large software systems accurately and efficiently. However, tools that can facilitate research in program comprehension are rarely publicly available.

In this paper, we introduce two approaches that generate artifacts, such as factbases and decompositions, that can be used to study the behaviour of existing software clustering approaches for the comprehension of large software systems. We also present three distinct applications of these approaches: the development of a simple evaluation method for clustering algorithms, the study of the behaviour of the objective function of the Bunch tool, and the calculation of a congruity measure for clustering evaluation measures. Implementations of the two approaches are available online.

I. Introduction

Several of the approaches that have been developed to help maintainers understand large software systems accurately and efficiently are available online [1], [2], [3], [4], [5]. However, tools that can facilitate research in program comprehension are rarely publicly available. This means that researchers have to recreate tools already developed by other community members, a process that requires significant amounts of time and energy.

While it is true that many research studies require dedicated tools, there are many general purpose approaches that can be reused in different contexts. In this paper, we introduce two such approaches that generate artifacts that could be useful to software maintenance researchers:

- A factbase generator that produces simulated factbases that resemble factbases extracted from real software systems
- A decomposition generator that produces simulated decompositions of software systems into subsystems that resemble decompositions produced by system experts or existing clustering algorithms

Both presented approaches support a number of configuration parameters that allow the user to produce output that is suited for their purposes. For example, the number of entities as well as an upper bound for the number of clusters in a software decomposition can be specified. Several experiments have been conducted to confirm that the two approaches produce artifacts that adhere to the provided configuration parameters, as well as uniformly span the whole spectrum of possible artifacts.

We also present three distinct applications of the introduced approaches: the development of a simple evaluation method for clustering algorithms, a study of the behaviour of the objective function of the Bunch tool [6], and the calculation of a congruity measure for clustering evaluation measures, such as MoJoFM [7] and KE [8]. The generator tools have also been used to study the comparability of software clustering algorithms [9].

All tools presented in this paper are available online as part of the Java Reverse Engineering Toolkit (JRET) [10].

The structure of the remainder of the paper is as follows: Section II presents necessary terminology background. The factbase generation approach is presented in Section III. Section IV introduces the decomposition generation approach. Experiments that demonstrate properties of the produced decompositions are presented in Section V. Section VI describes the application of the presented approaches to three distinct research tasks. Finally, Section VII concludes the paper.

II. Background

We begin by presenting some of the terminology used in this paper in Section II-A. The approaches presented in this paper have several applications in the software clustering context. For this reason, Section II-B briefly describes some of the most established software clustering algorithms: Hierarchical algorithms, Bunch, ACDC and LIMBO. In Section II-C we present existing methods for the evaluation of software clustering results.

A. Terminology

Several reverse engineering tasks start with the construction of a *factbase*. A factbase contains information about software entities, such as classes, source files etc., as well as their relations, such as inheritance, aggregation etc. Such information can be extracted from various sources, such as source code, makefiles, binary modules etc.

After a factbase has been constructed, a software clustering algorithm may be employed to group entities from the factbase in subsystems, thus creating a *decomposition* of the software system at hand.

A *flat* decomposition does not contain nested clusters. There is one level of clusters and one level of entities. An example of a flat decomposition is shown in Figure 1.

A *nested* decomposition is a decomposition that may contain nested clusters. An example of a nested decomposition is shown in Figure 2.

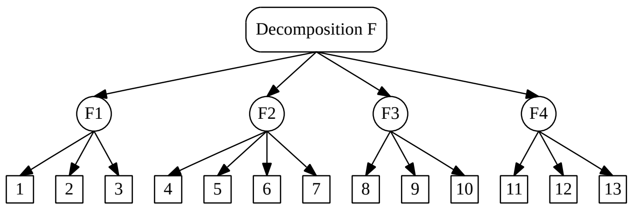


Figure 1. An example of a flat decomposition

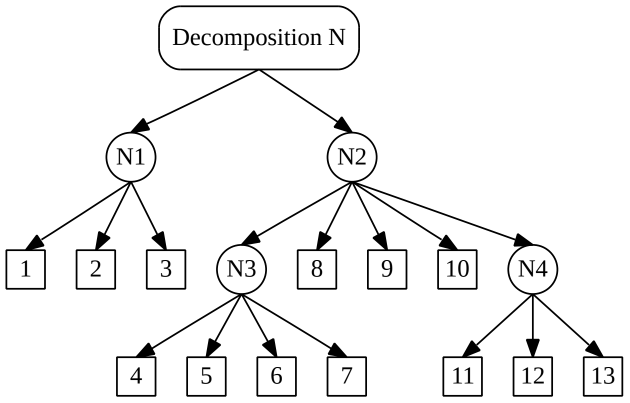


Figure 2. An example of a nested decomposition

B. Clustering algorithms

Hierarchical algorithms [11] utilize a similarity measure in order to compute pair-wise similarities for all entities to be clustered. The most similar entities are grouped together iteratively until some heuristic rule determines that a satisfactory set of clusters has been reached. The way the similarity between clusters is determined is called the update rule. In this paper, we experimented with two different similarity metrics (Jaccard and Sorensen–Dice) and four different update rules (single linkage, complete linkage, weighted average, and unweighted average).

Bunch is a clustering tool intended to aid the software developer and maintainer in understanding, verifying and maintaining a source code base [12]. Bunch views the clustering problem as an optimization problem, attempting to find a partition that maximizes an objective function that represents what a “good” partition is. A good partition is a partition where highly interdependent modules are grouped in the same cluster (representing subsystems) and independent modules are assigned to separate clusters. Finding a good graph partition involves systematically navigating through a very large search space of all possible partitions using techniques such as hill-climbing and genetic algorithms.

ACDC performs the task of clustering in two stages. In the first stage, it creates a skeleton of the final decomposition by identifying subsystems that resemble established subsystem patterns, such as the body-header pattern and the subgraph dominator pattern [13]. Depending on the pattern used the subsystems are given appropriate names. In the second stage, ACDC completes the decomposition by using an extended version of a technique known as Orphan Adoption. It attempts to place each unclustered software entity in the subsystem that depends most on it.

LIMBO is introduced in [14] as a scalable hierarchical categorical clustering algorithm that builds on the *Information Bottleneck* framework for quantifying the relevant information preserved when clustering. LIMBO has been successfully applied to the software clustering problem [15]. LIMBO’s goal is to create clusters whose features contain as much information as possible about the features of their contents.

C. Evaluation methods

Evaluating the usefulness of clustering results is a painstaking and often impractical task to perform manually. Manual evaluation can also be rather subjective. For this reason, researchers have developed automatic methods that can objectively assess the quality of a clustering algorithm’s output. Most of these methods compare the automatic decomposition created by the algorithm to an authoritative decomposition prepared by a system expert.

One of the early works on experimental evaluation of software clustering techniques was presented by Lakhotia and Gravely [16]. Their evaluation metric, however, can only be applied to hierarchical clustering algorithms that produce dendrograms, a feature that limits its applicability.

More recently, Anquetil and Lethbridge [17] used the well-known Information Retrieval measures of Precision and Recall in order to measure similarity between different clusterings of the same software system. Though their work produced many interesting results, the limitations of the two measures are well-documented [18], [19].

The MoJo distance measure [20] attempts to capture the distance between two decompositions as the minimum number of operations one has to perform in order to transform one into the other. The MoJoFM effectiveness measure [7] is based on MoJo distance but produces a number in the range 0-100 which is independent of the size of the decompositions.

Koschke and Eisenbarth [8] presented an elaborate framework that extends and removes the limitations of the approach taken by Lakhotia and Gravely [16]. They incorporate the fact that one may have to cope with approximate matches between clusters in real life. Their approach includes an overall recall rate that captures the quality of a particular technique. We refer to this recall rate as KE.

Mitchell and Mancoridis [18] were the first to present a similarity (*EdgeSim*) and a distance (*MeCl*) measurement that consider relations between components as the important factor for the comparison of software system decompositions.

The EdgeMoJo measure [21] attempts to combine several aspects of existing techniques. It is the only comparison method that considers both the placement of objects into clusters, as well as the relations between them.

Alternative attempts to evaluate software clustering algorithms have focused on an algorithm's stability [22], or the distribution of the cluster sizes [23]. While these methods can effectively weed out algorithms that are unstable or produce extreme clusters, they offer little insight when comparing stable algorithms that produce non-extreme results.

Finally, the CRAFT framework [24] presented an evaluation method that did not require an authoritative decomposition. It attempted to create one by combining the results of several clustering algorithms. Unfortunately, the framework was not adopted by the research community.

III. Simulated Factbase Generation

Many of the program comprehension approaches published in the literature involve the extraction of relevant information from the system's source code into what is typically called a *factbase*, i.e. a collection of facts about the software system. Once extracted, a factbase can be used as input for a wide variety of program comprehension tasks, such as visualizing the extracted facts, clustering the system's entities, or mining for patterns.

However, the creation of such a factbase can be a time-consuming task. Research activities that require a large number of such factbases, such as performance and robustness testing for clustering or visualization tools, would benefit from the ability to create simulated factbases that closely resemble ones from real software systems. In this section, we present a method for the creation of module dependency graphs, the most common type of factbase.

A module dependency graph (MDG) has vertices that are system modules, such as functions, classes, or source files, and edges that represent dependencies between these modules, such as procedure calls or variable references. As a result, an MDG is a directed graph.

In order to create realistic simulated MDGs, we need to determine the relevant properties of MDGs from actual software systems. Our approach identifies a number of such properties, and our implementation [25] provides options to enable or disable any or all of them. The design of the implementation allows for the easy addition of further properties.

The graph properties identified by our approach range from simple ones, such as having no loops, to quite involved ones, such as the power law distribution for the nodes' degrees. The kind of dependency to be simulated by the produced MDG should determine which of the following properties should be enabled:

1) No multiple edges: Only one edge is allowed between two modules.
2) No loops: A module may not depend on itself.
3) No cycles: Certain types of dependencies, such as class inheritance, may not appear in a cycle, while others, like object reference, frequently contain cycles.
4) Connectivity: The produced graph needs to be connected. If the generation process described below produces a disconnected graph, our algorithm identifies the connected components and modifies as many edges as required in order to connect them (without violating other constraints).
5) Low Clustering Coefficient: The clustering coefficient of an undirected graph is a measure of the number of triangles in the graph [26]. The software graphs of several large open-source systems, such as Linux and Mozilla, have clustering coefficients that belong to the interval [0.2, 0.4], so we implemented a similar restriction for the simulated graphs. Since

it is not practical to change the clustering coefficient of a graph by reshuffling edges, a violation of this constraint requires that the generation process has to be repeated.

6) Degree Power Law Distribution: Several studies argue that the degrees of the nodes in a typical MDG follow a power law distribution [27], [28], [29]. More precisely [30], the probability $P(k)$ of a given node having an in- or out- degree k, is expressed as:

$$P(k) \sim k^{-\gamma},$$

where γ is a value larger than 1, called the power law exponent. The value of γ in large software systems is consistently found to be around 2.5 with the average in-degree exponent being slightly less than that and the average out-degree exponent slightly more [29]. Our implementation uses 2.5 as the default value for the power law exponent.

The approach presented in this section works in a manner similar to the following simplified process (adapted so it applies to a directed graph) from a typical algorithm for the generation of random graphs with a power law distribution for its degrees [31]:

- Every node in the graph starts with a current in- and out- degree of 0.
- Every node in the graph is assigned a target in- and out-degree, so that the sequence of degrees follows a power law distribution. The sum of all target in-degrees is equal to the sum of all target out-degrees.
- Let I be the set of nodes whose current in-degree is less than their target in-degree. Let O be the set of nodes whose current out-degree is less than their target out-degree. While I and O are not empty, randomly select a node from each set. Create an edge between the two nodes.

In order to achieve the graph properties listed earlier, we modify the above process in order to apply additional constraints. We allow for two types of constraints:

1) Edge constraints. These are applied after each edge creation. If the constraint is violated, the edge creation is cancelled, and a new pair of nodes is selected. These ensure that the first three graph properties are satisfied: No multiple edges, no loops, and no cycles.
2) Graph constraints. These are applied after all edges have been created. If the constraint is violated, an effort will be made to satisfy it if possible by reshuffling some of the edges. If that is not possible, the graph generation process needs to be repeated.

Our implementation of this approach [25] produces MDGs that contain a single kind of dependency. If multiple dependencies are required, the process may be run several times with different settings. Since the output format of our implementation is RSF (Rigi Standard Format), combining the various dependencies is as simple as concatenating files.

Experiments using this factbase generation approach to study the comparability of software clustering algorithms [9] have demonstrated that the produced MDGs exhibit similar behaviour to graphs obtained by real software systems.

IV. Decomposition Generation

The second approach introduced in this paper allows a researcher to create simulated decompositions of large software systems. These can be used for a variety of tasks, such as studying the behaviour of software clustering evaluation measures [7], [8], [18], or testing the performance and robustness of visualization tools [32].

Section IV-A presents two versions of an algorithm for the generation of simulated flat software decompositions, while Section IV-B presents such an algorithm for nested decompositions.

A. Flat Decomposition Generation

The flat decomposition generation algorithm requires two inputs:

1) E: The number of entities to be clustered. Alternatively, a list of entity names can be provided. This way the produced decomposition will contain entities whose names are meaningful to the researcher.
2) U: An upper bound for the number of clusters in the produced decomposition. The exact number of clusters A will be a random number from the interval [2,U].

Once the number of clusters is determined, the generation algorithm proceeds in one of two ways:

- Unrestricted cardinality: In this version, each entity is randomly assigned to one of the clusters. Any clusters that remain empty after all entities have been assigned are deleted from the produced decomposition.
- Predetermined cardinality: In this version, each cluster is randomly assigned a cardinality before any entity distribution takes place. When entities are assigned to clusters, a cluster is selected only if its current size is less than the assigned cardinality.
  The sum of all cardinalities is, of course, equal to the number of entities to be clustered. Also, in order to avoid creating unrealistic software decompositions, 70% of the cardinalities are chosen from a normal distribution around the expected cardinality $\frac{E}{A}$, while the rest are random numbers from the interval [1,E-1].

As will be shown in Section V, both versions of the generation algorithm can produce all possible software decompositions for a given E and U. However, the two versions produce decompositions that differ in structure in the average case. Depending on the relation between E and A, they may produce balanced or unbalanced decompositions. Table I presents all possibilities.

Algorithm	Inputs	Decomposition
Unrestricted Cardinality	$A \ll E$	Balanced
Unrestricted Cardinality	$A \sim E$	Unbalanced
Predetermined Cardinality	$A \ll E$	Unbalanced
Predetermined Cardinality	$A \sim E$	Balanced

Table I. The relation between algorithm version, inputs, and output decomposition type.

A researcher should use the information in Table I as a general guideline when selecting input values as well as algorithm version for their research task.

B. Nested Decomposition Generation

The generation algorithm for simulated nested decompositions is in fact a simulation of a divisive hierarchical clustering algorithm, i.e. it begins by placing all entities in one cluster, which is then divided into successively smaller clusters. In order to achieve this, our algorithm requires three inputs:

1) E: The number of entities to be clustered, or a list of entity names as in the case of the flat generator.
2) C: An upper bound for the number of subclusters for a given cluster.
3) D: An upper bound for the height of the containment tree.

The generation algorithm starts by creating a flat decomposition using one of the algorithms presented in the previous section (U is equal to C in this case). This becomes the first level of the nested decomposition. An upper bound L for the height of the containment subtree rooted at each first level cluster is determined as a random number from the interval [1,D].

For each one of the clusters of the first level, the following process is repeated iteratively:

Create a flat decomposition for the entities in the cluster. Update the value of L as a random number from [0,L-1]. Stop if L=0, otherwise iterate to the next level.

The algorithm is presented in pseudocode in Figure 3.

In the next section, we present experiments that attempt to determine the randomness of the decompositions produced by the algorithms presented above.

V. Decomposition Generator Properties

For the approaches presented in this paper to be useful, the generated factbases and decompositions would need to span the entire space of possible outputs. Our factbase generator is based on an algorithm that has been shown to have this property [31]. As a result, in this section, we focus on investigating whether the decomposition generators presented in Section IV produce decompositions with the following two properties:

- **Uniqueness**: The probability that two generated decompositions are equal must be very small. Our experimental results are presented in Section V-A.
- **Randomness**: To our knowledge, no test for the randomness of decompositions exists currently. We adapt methods developed in the context of cryptography [33] to validate random number generators for our purposes. Our experimental results are presented in Section V-B.

A. Uniqueness

In this section, we investigate the uniqueness of the decompositions created by the generation algorithms presented earlier. We present experiments using the nested decomposition generator, since a flat decomposition is a special case of a nested decomposition. Experiments using the flat generators have yielded similar results.

An important issue to consider is that of isomorphic decompositions. Figure 4 presents two different representations for the same nested decomposition. To address this problem, we applied a method developed for comparing unsorted trees [34]. We define a string representation of a nested decomposition in such a way, so that two nested decompositions are equal iff their string representation is identical.

Definition V.1. The **string representation** $R(T)$ of nested decomposition T is a string constructed as follows:

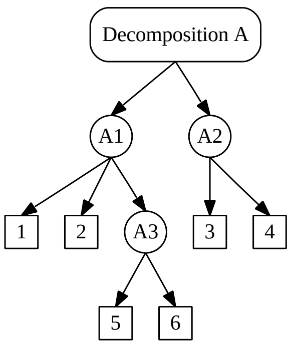
1) If $|T| = 1$, then $R(T)$ equals `"{root}"`, where `root` is the label of the single entity in the decomposition.
2) If $|T| > 1$, then let $S_1, ..., S_m$ be the immediate subtrees of T ordered so that $S_1 \succeq S_2 \succeq ... \succeq S_m$, where $\succeq$ is any partial order. Then, $R(T)$ is equal to "$\{R(S_1)R(S_2)...R(S_m)\}$".

For example, `"{{{1}{2}{{5}{6}}}{{3}{4}}}"` is the string representation for the two decompositions shown in Figure 4, when the partial order used is lexicographical sorting.

In order to assess the uniqueness of the decompositions produced by our tool, we created a large number of nested decompositions, constructed the string representation for

```
GenerateNestedDecomposition(E,C,D:int)
  begin
    F := GenerateFlatDecomposition(E,C)
    L := Random number in [1,D]
    Assign L as subtree height to all clusters in F
    Create empty nested decomposition N
    Assign F as the first level of N
    while there exists cluster x in N whose subtree height is
           larger than 0 do
      F := GenerateFlatDecomposition(|x|,C)
      L := Random number in [0,L-1]
      Assign L as subtree height to all clusters in F
      Replace cluster x with F
    end while
  end
```

Figure 3. Pseudocode for the simulated nested decomposition generation process.



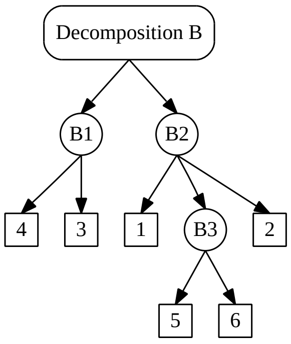


Figure 4. An example of two isomorphic nested decompositions.

each one, and computed the number of duplicate string representations.

We performed 100 experiments where the input parameters were: E = 1000, C = 10, D = 9. In every experiment, 1000 decompositions were generated. There was only one pair of duplicate decompositions found in all these experiments. This confirms that our algorithm produces unique decompositions.

B. Randomness

Next, we investigate whether the decomposition generators produce well randomized sets of decompositions. We concentrated on the following variables and identified a null hypothesis for the frequency distribution of each variable:

- Number of clusters in a flat decomposition. Null hypothesis: Uniform distribution in the range of requested values.
- The probability that two entities belong in the same cluster in a flat decomposition. Null hypothesis: This probability is the same for each pair of entities.
- Number of children for a non-leaf node of a nested decomposition. Null hypothesis: Uniform distribution in the range of requested values.
- The probability that two entities belong in the same cluster in a nested decomposition. Null hypothesis: This probability is the same for each pair of entities.
- Height of leaf nodes in a nested decomposition. Null hypothesis: Uniform distribution in the range of requested values.

We used the chi-square (χ^2) test statistic [35] to measure how well the observed frequency distribution for the above variables matches the expected one. For an experiment with k possible outcomes, the value of the chi-square test statistic is given by the formula:

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

where E_i is an expected frequency asserted by the null hypothesis, and O_i is the observed frequency.

Based on the value of the chi-square statistic, one can calculate the p-value, i.e. the probability that the differences between the observed and expected frequencies occurred by chance. If that probability is small (typically less than 0.05) then the null hypothesis is rejected.

In our experiments, the p-values we observed were always in the interval [0.9, 1], even though we repeated each experiment 1000 times. While this is not equivalent to proving the null hypotheses, it provides a strong indication that the decomposition generators presented in Section IV are well randomized.

VI. Applications

We now present three distinct situations where the application of the generator tools presented in this paper can help provide significant insight in various open questions related to software clustering.

A. A simple clustering evaluation method

Software clustering algorithms decompose a software system into subsystems. To evaluate the quality of a software clustering algorithm, one typically compares the decomposition produced by a software clustering algorithm with a decomposition produced by an expert. However, due to the complexity of most real industrial systems, experts often disagree about what constitutes a good decomposition. This situation makes evaluation of software clustering algorithms a rather difficult task.

For the clustering evaluation method presented in this section, we make the assumption that when the structure of a software system is not complex, then experts and clustering algorithms alike should agree on how to decompose the system into subsystems. For instance, if the MDG of a software system is a rooted tree, such as the one presented in Figure 5, one would expect that a decomposition such as the one shown in Figure 6 would be universally accepted[1].

By utilizing the factbase generator tool presented in Section III, we can devise a simple way to evaluate how well clustering algorithms, such as ACDC and Bunch, perform in such a situation. Our evaluation method proceeds as follows:

[1]The root entity can be assigned to any cluster from the universally accepted decomposition. In our experiments, we consider the placement of the root to be correct regardless of the cluster it belongs to.

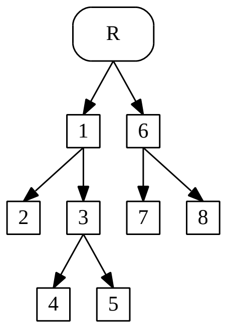


Figure 5. A rooted tree MDG.

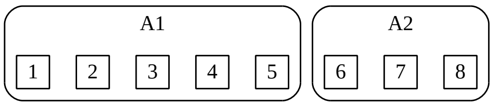


Figure 6. The decomposition that should correspond to the MDG in Figure 5.

- Construct an MDG that is a rooted tree by requiring no loops, no cycles, and a connected graph. To ensure that the output is a rooted tree, the direction of certain edges was reversed.
- Construct a decomposition from the MDG using the clustering algorithm being evaluated.
- Calculate the MoJoFM distance between the expected decomposition and the decomposition constructed by the clustering algorithm.
- Repeat from the beginning a large number of times.

We applied this evaluation measure to ACDC and Bunch by repeating the above process 100 times for three different sets of input values. The obtained results are shown in Table II.

The results show that even established software clustering algorithms, such as ACDC and Bunch, may fail to discover good software decompositions from software systems with trivial structure (if they always did this correctly, the MoJoFM values should have always been 100). The MDGs created during these experiments may provide interesting insight as to why the algorithms deviated from the expected result.

It is also important to note that the evaluation method just presented does not require the existence of an authoritative decomposition. Very few clustering evaluation methods share this property [22], [23].

B. Properties of Bunch's MQ function

For our next application, we will use both presented generator tools in order to study the properties of the

E	C	D	Algorithm	MoJoFM values	Mean	Standard Deviation
1000	20	10	ACDC	60.73 - 100	88.91	9.4
			Bunch	33.88 - 100	77.49	18.06
800	20	10	ACDC	76.29 - 100	90.87	6.62
			Bunch	34.25 - 100	79.77	17.16
800	10	10	ACDC	91.3 - 100	95.57	4.77
			Bunch	51.25 - 100	83.77	14.16

Table II. Results for the simple clustering evaluation method.

objective function used by Bunch. It is well-known that, due to the fact that it uses randomization, Bunch may create different results for subsequent runs using the same input parameters. We wish to investigate whether this can be attributed to the fact that the objective function MQ has a large number of local optima.

The way Bunch clusters software systems is as follows: Start with an initial decomposition that may be created randomly or provided by the user. Modify the decomposition by moving entities between clusters. If these changes improve the value of the objective function MQ, then accept the changes, otherwise discard them. Repeat this process until no further improvement can be made to the value of MQ. It has been shown that Bunch will always find a decomposition whose MQ values is close to the global maximum [36].

We run the following experiment:

1) Create a simulated factbase F using the generator tool presented in Section III. For the experiments presented, the following constraints were activated: no multiple edges and no loops. The number of entities (E) was 2000.
2) Create decomposition B_0 by running Bunch with F as input.
3) Construct 100 simulated decompositions S_i ($1 \leq i \leq 100$) that refer to the same entities as F by using the decomposition generator tools in Section IV-A. The input values used were: E = 2000, U = $|B_0|$ + 10. The value of parameter U was selected so that the simulated decompositions have a cardinality that is similar to B_0.
4) Construct decompositions B_i ($1 \leq i \leq 100$) by running Bunch with F as input and S_i as the initial decomposition.
5) Compute the MQ value for all B_i, and compare to MQ(B_0).

We repeated this experiment 10 times. Between 20% to 30% of the B_i decompositions had an MQ value higher than that of B_0. This shows that there are several distinct decompositions that have a high MQ value, which implies the existence of a large number of local optima. It is interesting to note that the B_i decompositions that had higher MQ values than B_0 were distinctly different from B_0. The MoJoFM similarity between B_i, for which MQ(B_i) $>$ MQ(B_0), and B_0 ranged in the interval [35.9, 45.65].

C. Congruity of Clustering Evaluation Measures

The last application of the generator tools introduced in this paper has to do with investigating the behaviour of clustering evaluation measures, such as MoJoFM and KE. While the goal of such measures is the same – to determine the similarity between two different decompositions of the same software system – they often lead to contradictory results [37], [9].

Suppose we want to compare three decompositions A, B, and C of the same software system using evaluation measures M and U (the higher the value of M or U, the more similar the decompositions). By applying both measures to all pairs of decompositions, we obtain the results in Table III.

Decomposition pair	(A,B)	(B,C)	(A,C)
M values	3	2	1
U values	5	8	2

Table III. Evaluation measure values for all decomposition pairs.

According to evaluation measure M, A and B are the two most similar decompositions. However, according to U, this is incorrect (B and C are the two most similar). This is a significant difference because it affects the way the two algorithms rank the three decompositions. In other words, we would not consider it a noteworthy issue if U produced larger similarity values than M, as long as the two algorithms ranked all decompositions in the same order of similarity.

A generalization of this idea leads to a congruity metric that determines correlation between two evaluation methods of software clustering algorithms. Assume we have N pairs of decompositions of the same software system. We can apply both M and U to each pair and obtain two different values m_i and u_i. We arrange the pairs of decompositions in such a way so that for $1 \leq i \leq N-1$, we have $m_i \leq m_{i+1}$. The value of the congruity metric will be the number of distinct values of i for which $u_i > u_{i+1}$. Values for this metric range from 0 to $N-1$. A value of 0 means that both comparison methods rank all pairs in exactly the same order, while a value of $N-1$ probably indicates that we are comparing a distance measure to a similarity measure. Values significantly removed from both 0 and $N-1$ indicate important differences between the two comparison methods.

The calculation of the congruity metric requires a large number of pairs of software decompositions. Using the tools introduced in this paper, we can devise two distinct ways of generating these decompositions:

The first generation method utilizes any of the decomposition generator algorithms presented in Section IV.

The second generation method starts by generating a random factbase using the process described in Section III. Next, two different clustering algorithms construct two distinct software decompositions based on the same factbase.

Both generation methods have advantages and disadvantages. The main drawback of the first method is that the decompositions may be significantly different from each other. This situation is unlikely when the two decompositions are actual clustering results. The main drawback of the second method is its dependency on the two software clustering algorithms used. Experiments with several pairs of algorithms may have to be conducted in order to remove possible bias in the obtained results.

VII. Conclusions

This paper presented two approaches for the generation of factbases and decompositions that can be useful to researchers requiring large numbers of such artifacts.

Three example applications of these generation approaches were also presented to demonstrate their usefulness for different research tasks.

Implementation of these approaches are available online [10]. We hope that researchers will find them useful for their purposes.

References

[1] http://serg.cs.drexel.edu/redmine/projects/list_files/bunch.

[2] http://www.cse.yorku.ca/~bil/downloads/.

[3] http://www.swat.uwaterloo.ca/~swagkit.

[4] H. A. Müller and K. Klashinsky, "Rigi - a system for programming-in-the-large," in *Proceedings of the 10th International Conference on Software Engineering*, 1988, pp. 80–86.

[5] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer, *Moose: a Collaborative and Extensible Reengineering Environment*, ser. RCOST / Software Technology Series. Franco Angeli, 2005, pp. 55–71. [Online]. Available: http://www.moosetechnology.org/

[6] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IEEE proceedings of the 1998 International Workshop on Program Comprehension*. IEEE Computer Society Press, 1998.

[7] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings of the Twelfth International Workshop on Program Comprehension*, 2004, pp. 194–203.

[8] R. Koschke and T. Eisenbarth, "A framework for experimental evaluation of clustering techniques," in *Proceedings of the Eighth International Workshop on Program Comprehension*, Jun. 2000, pp. 201–210.

[9] M. Shtern and V. Tzerpos, "On the comparability of software clustering algorithms," in *International Conference on Program Comprehension*, 2010.

[10] https://wiki.cse.yorku.ca/project/cluster/tools.

[11] O. Maqbool and H. Babri, "The weighted combined algorithm: a linkage algorithm for software clustering," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, 2004, pp. 15–24.

[12] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A clustering tool for the recovery and maintenanceof software system structures," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[13] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proceedings of the Seventh Working Conference on Reverse Engineering*, Nov. 2000, pp. 258–267.

[14] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "Limbo: Scalable clustering of categorical data." in *Proceedings of the Ninth International Conference on Extending DataBase Technology*, ser. Lecture Notes in Computer Science, vol. 2992. Springer, 2004, pp. 123–146. [Online]. Available: http://dblp.uni-trier.de/db/conf/edbt/edbt2004.html#AndritsosTMS04

[15] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 150–165, 2005.

[16] A. Lakhotia and J. M. Gravley, "Toward experimental evaluation of subsystem classification recovery techniques," in *Proceedings of the Second Working Conference on Reverse Engineering*, Jul. 1995, pp. 262–269.

[17] N. Anquetil and T. Lethbridge, "Experiments with clustering as a software remodularization method," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, Oct. 1999, pp. 235–255.

[18] B. S. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *Proceedings of the International Conference on Software Maintenance*, Nov. 2001, pp. 744–753.

[19] V. Tzerpos, "Comprehension-driven software clustering," Ph.D. dissertation, University of Toronto, July 2001.

[20] Z. Wen and V. Tzerpos, "An optimal algorithm for MoJo distance," in *Proceedings of the Eleventh International Workshop on Program Comprehension*, May 2003, pp. 227–235.

[21] ——, "Evaluating similarity measures for software decompositions," in *Proceedings of the International Conference on Software Maintenance*, 2004.

[22] V. Tzerpos and R. C. Holt, "On the stability of software clustering algorithms," in *Proceedings of the Eighth International Workshop on Program Comprehension*, Jun. 2000, pp. 211–218.

[23] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 525–535.

[24] B. S. Mitchell and S. Mancoridis, "Craft: A framework for evaluating software clustering results inthe absence of benchmark decompositions," in *Proceedings of the Eighth Working Conference on Reverse Engineering*, Oct. 2001, pp. 93–103.

[25] M. Shtern, "The factbase generator tool," Available online: https://wiki.cse.yorku.ca/project/cluster/mdg_generator.

[26] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks." *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998. [Online]. Available: http://dx.doi.org/10.1038/30918

[27] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Physical Review E*, vol. 68, no. 4, p. 46116, 2003.

[28] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 687–708, 2007.

[29] S. Valverde and R. V. Sole, "Hierarchical small worlds in software architecture," Available online: http://arxiv.org/abs/cond-mat/0307278.

[30] M. L. Goldstein, S. A. Morris, and G. G. Yen, "Problems with fitting to the power-law distribution," August 2004. [Online]. Available: http://arxiv.org/abs/cond-mat/0402322

[31] W. Aiello, F. Chung, and L. Lu, "A random graph model for power law graphs," *Experimental Math*, vol. 10, pp. 53–66, 2000.

[32] Software Architecture Group (SWAG) at University of Waterloo, "LSEdit: The graphical landscape editor," Available online: http://www.swag.uwaterloo.ca/lsedit.

[33] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, A. Rukhin, J. Soto, M. Smid, S. Leigh, M. Vangel, A. Heckert, J. Dray, and L. E. B. Iii, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, 2001.

[34] S. R. Buss, "Alogtime algorithms for tree isomorphism, comparison, and canonization," in *Proceedings of the 5th Kurt Godel Colloquium on Computational Logic and Proof Theory*. Springer-Verlag, 1997, pp. 18–33.

[35] A. M. Mood, F. A. Graybill, and D. C. Boes, *Introduction to the Theory of Statistics*, 3rd ed. McGraw-Hill Companies, April 1974. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0070428646

[36] S. Xanthos, "Clustering object-oriented software systems using spectral graph partitioning," 2006. [Online]. Available: http://www.acm.org/src/subpages/papers/GrandFinals2005/SpirosXanthosACMSRC.pdf

[37] M. Shtern and V. Tzerpos, "Refining clustering evaluation using structure indicators," in *ICSM '09: Proceedings of the 25st IEEE International Conference on Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 297–305.