DOI 10.1007/s00165-011-0199-y
BCS © 2011
Formal Aspects of Computing (2012) 24: 307–330

Formal Aspects of Computing

Using fold-in and fold-out in the architecture recovery of software systems

Michele Risi[1], Giuseppe Scanniello[2] and Genoveffa Tortora[1]

[1] University of Salerno, Via Ponte Don Melillo, 84084 Fisciano (SA), Italy. E-mail: mrisi@unisa.it; tortora@unisa.it
[2] Dipartimento di Matematica e Informatica, University of Basilicata, Viale Dell'Ateneo, 85100 Potenza, Italy.
E-mail: giuseppe.scanniello@unibas.it

Abstract. In this paper we present an approach to automate the architecture recovery process of software systems. The approach is built on information retrieval and clustering techniques, and, in particular, uses Latent Semantic Indexing (LSI) to get similarities among software entities (e.g., programs or classes) and the k-means clustering algorithm to form groups of software entities that implement similar functionality. In order to improve computational time in the context of the software evolution and then reduce energy waste, the architecture recovery process can be also applied by using fold-in and fold-out mechanisms that, respectively, add and remove software entities to the LSI representation of the understudy software system. The approach has been implemented in a prototype of a supporting software system as an Eclipse plug-in. Finally, to assess the approach and the plug-in, we have conducted an empirical investigation on five open source software systems implemented using the programming languages Java and C/C++. In the investigation special emphasis has been also given to the effect of using the fold-in and fold-out mechanisms.

Keywords: Software partitioning, Latent semantic indexing, Clustering, Program comprehension, Software maintenance

Correspondence and offprint requests to: M. Risi, E-mail: mrisi@unisa.it

1. Introduction

Software engineering is the application of engineering to software. The life cycle paradigm for software includes: requirements, design, construction, testing, and maintenance [BD09]. Although the software maintenance phase is an integral part of the software life cycle, it has not historically received the same degree of attention as the other phases of the software life cycle [Pig97]. However, the interest in the software community has been changing with respect to maintenance since software companies try to keep software operating as long as possible to obtain the most out of the development investments.

In contrast with the development that typically can last for 1-2 years, the maintenance phase typically lasts for many years [ZSG79]. This phase starts after the delivery of the first version of a software and lasts much longer than the initial development phase [IEE99]. During the maintenance phase a software system will be continuously changed and enhanced due to execution of maintenance operations that are carried out for several reasons (e.g., to correct faults or to improve performances) [Leh84]. Maintenance operations might modify the architectural structure of the software system [BHB99, EGK+01], thus making the documentation (if present) outdated [TP04].

The recovery of the architectural documentation of an existing software system represents a longstanding and relevant research topic. In fact, a number of approaches, techniques, and tools have been designed and developed to support the architecture recovery and the modularization of legacy software systems [vDHK+04, Kos00, MOTU93]. They are generally based on clustering algorithms to partition software systems into meaningful subsystems. Furthermore, they may be difficult to use in practice as they are semiautomatic and are designed for software systems implemented using a specific programming language [AFL99, Ton01, WH05].

This paper presents an automatic approach for the architecture recovery of existing software systems. To partition existing software systems into subsystems the approach first performs an analysis phase to compute the dissimilarity between software entities (e.g., programs or classes) using Latent Semantic Indexing (LSI) [DDL+90] and then uses the k-means clustering algorithm [Mac67] to group the entities. Indeed, to identify the best partition of software entities into clusters k-means is iteratively executed using different configurations. In the context of software evolution, computational time can be improved and then energy waste can be reduced using *fold-in* and *fold-out* mechanisms. Fold-in is employed to incrementally add entities to the LSI representation of the under-study software system. On the other hand, entities can be *folded out* to the LSI representation removing software entities that have not to be used in the architecture recovery process since they are not any more available in the analyzed system. Another remarkable advantage of our approach is that it is general and can be applied on software systems implemented using any kind of programming language.

To automate the clustering process we have also implemented a prototype of a supporting software system as an Eclipse plug-in. The approach and the plug-in have been successively applied on a number of versions of five open source software systems implemented using the Java and C/C++ programming languages. These systems have been used to empirically assess whether our approach outperform other clustering based approaches for software remodularization (i.e., the ones presented in [BG09, CDMS10, WH05]). Depending on the approach, the comparison has been performed according to the following three criteria:

1. *authoritativeness*: An automatically produced partition should approximate the one produced by a software architect (e.g., an authority).
2. *stability*: When a system changes modestly in subsequent versions, the produced partitions should also change modestly.
3. *non-extremity cluster distribution*: Automatically produced clusters within software partitions should generally not be either large (i.e., containing hundreds of programs or classes) or small (i.e., containing very few programs or classes).

The investigation has also been designed to verify the effect of using fold-in and fold-out mechanisms on the clustering. To this end, the analysis has been performed using the three criteria above and the computation time and the efficiency as well. The efficiency has been computed dividing the quantitative evaluation of the authoritativeness of the clustering by the time needed to perform the clustering.

The paper is an extension of the work presented in [SRT10]. With respect to that paper, we provide here the following main new contributions:

- The approach has been extended enabling the fold-in and fold-out of software entities from the LSI representation of the understudy software system;
- An extended version of the case study, which now involves five open source software systems implemented in Java and C++. The case study also analyzes the benefit deriving from the use of the fold-in and fold-out mechanisms.

The remainder of the paper is organized as follows: related work is discussed in Sect. 2. In Sect. 3 we describe the proposed approach, while the supporting environment is presented in Sect. 4. The design of the empirical investigation and the discussion of the obtained results are presented in Sects. 5 and 6, respectively. The paper is concluded discussing final remarks and drawing possible future directions for our research.

2. Related work

Clustering based methods and approaches have been proposed [AFL99, KDG07, Wig97] to retrieve the architectural documentation of a software system. To avoid that such a kind of approaches produce unsuitable results, several non trivial issues have to be considered [Kos00]. In particular, a software engineer should [40]: (i) choose the level of granularity for the software entities to consider in the clustering (e.g., methods or classes); (ii) identify a suitable measure to compare these entities; (iii) select the clustering algorithm to group the most similar entities according to the identified measure.

Based on the experience we gained in surveying the research work presented in this section, we considered the following characteristics to group and discuss the related literature summarized in Table 1:

1. The programming language used to implement software systems on which the clustering based approaches and tools can be applied,
2. The level of granularity of the software entities to cluster,
3. The used information to compare software entities, namely structural, lexical, and their combination;
4. The clustering algorithm/s used or assessed to group the software entities,
5. If the approach is automatic or semiautomatic,
6. The kind of software system/s used for assessing the approach and tool,
7. The criteria used to assess the validity of the results.

Anquetil and Lethbridge [AFL99] extend the work by Wiggerts [Wig97] by presenting a comparative study of different hierarchical clustering algorithms and by analyzing their properties with regard to software remodularization. Also the considered algorithms need human decisions (e.g., cutting points of the dendrograms) to get the best partition of software entities into clusters.

Maqbool and Babri [MB07] highlight hierarchical clustering research in the context of software architecture recovery and remodularization. Special emphasis is posed on the analysis of various similarity and distance measures that could be used in the software clustering and in the software remodularization, in particular. The main contribution of the paper is, however, the analysis of two clustering based approaches and their experimental assessment. The discussed approaches try to reduce the number of decisions to be taken during the clustering. Differently from us, good quality clusters are only identified with the support of a software engineer. They also conducted an empirical evaluation of the clustering based approaches on four large software systems. The stability of the clustering algorithms has not been considered in their empirical evaluation.

In [MM06] a clustering system (i.e., Bunch) is presented and analyzed. Differently from our tool, to produce a decomposition of a system in subsystems Bunch uses search techniques to partition the graph representing software entities and their relations. Indeed, the tool is based on several heuristics to navigate through the search space of all possible graph partitions. To evaluate the quality of graph partitions and to find a satisfactory solution the tool uses fitness functions and search algorithms. The tool effectiveness has been assessed using qualitative and quantitative empirical investigations. Also, in [DMM99] a structural approach based on genetic algorithms is proposed to group software entities in clusters. The effectiveness of the approach has been assessed on a small software system. Subsequent versions have not been considered.

Table 1. Overview of architecture recovery approaches

Approach	Supported programming language	Granularity	Used information	Clustering algorithm	Automatic or semiautomatic	Type of systems used for the assessment	Assessment Criteria
Anquetil and Lethbridge [AFL99]	C/Pascal	Files	Structural	Bunch hierarchical	Semi-automatic	Open-source	Coupling cohesion
Kuhn et al. [KDG07]	Java/Smalltalk	Classes/methods	Lexical	Hierarchical	Semi-automatic	Open-source	Qualitative Analysis
Maqbool and Babri [MB07]	C	Procedures	Lexical/structural	Hierarchical	Semi-automatic	Open-source	Mojo precision recall koschke-eisenbarth
Mitchell and Mancoridis [MM06]	C/C++/Java	Classes	Structural	Hill climbing	Automatic	Industrial open-source	Authoritativeness stability ned
Doval et al. [DMM99]	C/C++/Turing	Classes	Structural	Genetic algorithm	Automatic	Industrial	Qualitative analysis
Bittencourt and Guerrero [BG09]	C/C++/Java	Classes	Structural	Edge betweenness k-means modularization quality design structure matrix	Semi-automatic	Open-source	Authoritativeness stability ned
Wu et al. [WH05]	C/C++	Classes	Structural	Agglomerative program comprehension patterns Bunch	Semi-automatic	Open-source	Authoritativeness stability ned
Tzerpos and Holt [TH00]	C	Files/procedures	Structural	Hierarchical	Semi-automatic	Industrial/ open-source	Analytical stability practical stability black-box stability
Corazza et al. [CDMS10]	Java	Classes	Lexical	k-Medoids	Semi-automatic	Open-source	Authoritativeness ned
Corazza et al. [CDMMS11]	Java	Classes	Lexical	k-Medoids	Semi-automatic	Open-source	Authoritativeness ned
Maletic and Marcus [MM01]	C	Files	Lexical/structural	Minimum spanning tree		Open-source	Coupling cohesion
Scanniello et al. [SDDD10b]	Java	Classes	Lexical/structural	k-means	Automatic	Open-source	Authoritativeness stability ned
Proposed approach	C/C++/Java	Classes	Lexical	k-means	Automatic	Open-source	Authoritativeness stability ned efficiency time

Clustering algorithms based on structural information have been also used in the analysis of the software architecture evolution [BG09, WH05]. For example, Wu et al. in [WH05] present a comparative study of a number of clustering algorithms: (a) an agglomerative clustering algorithm (based on the Jaccard coefficient and the complete linkage update rule) using 0.75 and 0.90 as cutting points; (b) an agglomerative clustering algorithm (based on the Jaccard coefficient and the single linkage update rule) using 0.75 and 0.90 as cutting points; (c) an algorithm based on program comprehension patterns that tries to recover subsystems that are commonly found in manually-created decompositions of large software systems; and (d) a customized configuration of an algorithm implemented in Bunch [MM06]. To partition a software system into meaningful subsystems the greater part of these algorithms needs to be manually configured (e.g., the specification of cutting points and fitness functions). The selected algorithms have been applied on the subsequent versions of five large C/C++ open source systems. Stability, authoritativeness, and non-extremity of cluster distribution are used to compare these algorithms. Similarly, in [BG09] an empirical study is presented to evaluate four widely known clustering algorithms on a number of software systems implemented in Java and C/C++. The analyzed algorithms are: Edge betweenness clustering *(eb)*, k-means clustering *(km)*, modularization quality clustering *(mq)*, and design structure matrix clustering *(dsm)*. The clustering results are similarly assessed in both [BG09] and [WH05]. In particular, the authors use the measure proposed by Tzerpos and Holt in [TH99] and [WT03] to quantitatively assess the decompositions produced by the clustering algorithms in terms of authoritativeness and stability. This measure is called MoJo and is defined as the minimum number of move and joint operations to turn a source partition into a target one. The computation of the MoJo distance proposed in [TH99] is based on a heuristic that approximates the exact measure values, while in [WT03] is suggested a new algorithm to calculate the exact MoJo distance in polynomial time.

MoJo has been also used to assess the results of clustering based approaches and tools differently from the ones presented in [BG09, WH05]. For example, Tzerpos and Holt use the MoJo distance in [TH00] to study the stability and the quality of a number of software clustering algorithms. Even in this case, the selected algorithms need a tuning phase to get good software partitions. The comparison among clustering algorithms is conducted generating randomly "perturbed" versions of an example system. Successively, differences between the partition identified by the clustering algorithms and the original partition of the system are measured. Random perturbation of a fixed size system could be however considered inadequate to simulate the behavior of a clustering algorithm on actual software systems both commercial and open source.

Generally, reverse engineering approaches are focused on structural information to recover software architectures [DMM99, MM06]. Nevertheless, the domain knowledge of the developers is generally embedded in the code comments. For such a reason, Kuhn et al. in [KDG07] describe an approach to group software entities based on LSI [DDL+90]. The approach is language independent and tries to group source code containing similar terms in the comments. The authors consider different levels of abstraction to understand the source code (i.e., methods and classes). The approach also enables software engineers to identify topics in the source code by means of labeling of the identified clusters. The approach is implemented within a tool named Hapax, which is built on top of the Moose reengineering environment [DGLD05]. Case studies are used to assess the approach and the tool support. Differently from us, a hierarchical agglomerative clustering algorithm is employed and the authors do not assess the approach on different distribution versions of a given software system. Another difference with respect to our approach is that human decisions are needed to identify the best partition of software entities into clusters.

Corazza et al. [CDMS10] propose a clustering based approach that uses lexical information extracted from four zones in Java classes, which are weighed using a probabilistic model. The Expectation-Maximization (EM) algorithm is applied. Classes are then grouped using a customization of the k-medoids clustering algorithm. The approach is assessed on a case study in terms of authoritativeness and non-extremity of the cluster distribution. The case study is also focused on the assessment of the benefit deriving from the use of EM. This further investigation reveals that the use of the probabilistic model improves the results. Several are the differences with respect to the approach presented here: the human intervention is needed and the source code needs to be commented. More recently in [CDMMS11] an investigation on the effectiveness of exploiting lexical information coming from six different zones in source code is proposed. In particular, the authors analyze 13 open source Java software systems to explore the contribution of the combined use of six different dictionaries, corresponding to the six parts of the source code where programmers introduce lexical information, namely: class, attribute, method and parameter names, comments, and source code statements. The authors use a frequentistic based EM to automatically weight the relevance of the six zones. Software entities are then grouped using a hierarchical clustering algorithm.

Maletic and Marcus [MM01] propose an approach based on the combination of lexical and structural information to support comprehension tasks within the maintenance and reengineering of software systems. From the lexical point of view they consider problem and development domains. On the other hand, the structural dimension refers to the actual syntactic structure of the program along with the control and dataflow that it represents. Software entities are compared using LSI, while file organization is used to get structural information. To group programs in clusters a simple graph theoretic algorithm is used. The algorithm takes as input an undirected graph (the graph obtained computing the cosine similarity of the two vector representations of all the source code documents) and then constructs a Minimal Spanning Tree (MST). Clusters are identified pruning the edges of the MST with a weight larger than a given threshold. To assess the effectiveness of the approach some case studies on a version of Mosaic are presented and discussed. Differently from us, human decisions are needed to identify a good partition. Another difference concerns the used clustering algorithm and the fact that subsequent versions of Mosaic have been not considered to analyze the evolution of software architectures.

More recently, Scanniello et al. [SDDD10b] present a two phase approach for recovering hierarchical software architectures of object oriented software systems. The first phase uses structural information to identify software layers [SDDD10a]. To this end, a customization of the Kleinberg algorithm [Kle99] is used. The second phase uses lexical information extracted from the source code to identify similarity among pairs of classes and then partitions each identified layer into software modules. The main difference with respect to the approach presented here is that it is only suitable for software systems exhibiting a classical tiered architecture. This makes useless the comparison between the clustering results achieved by applying that approach and the one presented here.

None of the highlighted approaches is based on an incremental strategy (i.e., fold-in and fold-out) to group software entities that implement related functionality. Further, our approach can be automatically used to recovery the architectural view of a software system implemented using any programming language. Finally, in this paper is the first time that the efficiency has been considered to assess the validity of a clustering based approach for architecture recovery and modularization.

3. The approach

The approach uses the LSI technique [DDL+90, Har92] to compute a similarity measure between software entities (i.e., classes or programs), based on their textual content and then iteratively uses a k-means[1] clustering algorithm [JMF99, Mac67] to identify groups of entities implementing similar functionality. In particular, the approach presented here is based on the following two steps (or phases):

- *Entity analysis and computing dissimilarity.* A vector representation is constructed for each software entities of the system understudy and the LSI is then applied to compute the concept space of these entities. Fold-in and fold-out mechanisms are possibly used to reduce the time needed to update the concept space previously computed. The concept space is then employed to compute a matrix (dissimilarity matrix in the following), which will contain the dissimilarity between each pair of entities of the software system to study.
- *Clustering software entities.* A k-means clustering algorithm is applied more than once on the dissimilarity matrix computed in the previous phase. In particular, the clustering algorithm is applied choosing as priori fixed number of clusters numbers ranging from 2 to $n/2$, where n is the number of entities of the analyzed software system. The configuration that enables to maximize the following function is the partition that the approach produces as output:

$$\text{NED} = \frac{\sum_{i=1 \wedge i\ not\ extreme}^{m} n_i}{n} \tag{1}$$

where m is the number of clusters identified by the tool, while n_i represents the size of the i$^{\text{th}}$ identified cluster. Similarly to [ZSG79], clusters with less than 5 entities and more than 100 entities are considered as extreme lower and upper limits, respectively. We used the NED function to get neither large nor small clusters.

In the following subsections further details on these two phases are provided.

[1] K-means is a supervised partitional clustering algorithm that classifies entities through a priori fixed number of disjoint clusters.

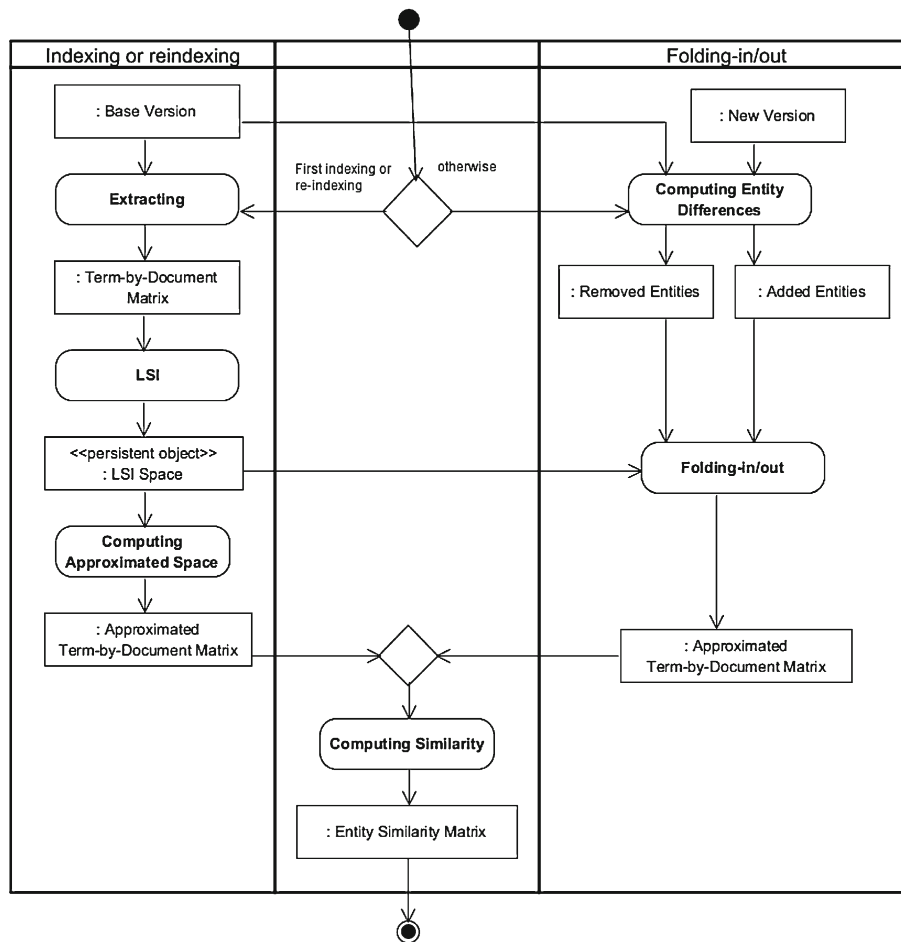

Fig. 1. The process of the phase entity analysis and computing dissimilarity

3.1. Entity analysis and computing dissimilarity

This phase is refined by the process of Fig. 1, which is modeled in terms of an activity diagram with object flow. Ovals represent phases, while rectangles represent intermediate artifacts produced or consumed by the phases of the process.

The *Extracting* phase is in charge of extracting the text (i.e., comment and source code) of the software entities. Software systems implemented in C/C++ and Java are differently treated because of their different nature (e.g., C and C++ support the static source code file inclusion). In particular, in each C/C++ source file all the header files, except the ones of the programming language library, are added. Once the preprocessing phase has been accomplished the Java and C/C++ source files are treated as plain text. This makes the subsequent phases of the approach independent form the programming language used to implement the software system to be analyzed. Therefore, the approach can be easily extended to support different programming languages simply defining a preprocessor to produce a plain text representation of the software entities to be grouped in clusters.

The attained plain text representations have to undergo a normalization phase in which non-textual tokens are eliminated (i.e., operators, special symbols, numbers, etc.), terms composed of two or more words are split (e.g., last_name is turned into last and name) and terms within a stopword list and with a length less than three characters are not considered. The keywords of the Java and C/C++ programming languages are included in the stopword list used in the preprocessing of the source code. Similarly to [CDMS10, CDMMS11, MM01, SDDD10b], we adopt a Porter stemmer [MRS08] on both the comments and the source code in order to reduce inflected (or

sometimes derived) terms to their stem. For example, both the words *designing* and *designer* lead to the common radix *design*.

The preprocessed text is then used to get the term-by-document matrix *A*. An entry $a_{i,j}$ of this matrix denotes the number of times that the *i*th term appears in the *j*th document (software entity in our case). We used the *term frequency-inverse document frequency*, also known as *tf-idf*, to assign to the term t_i a weight that is:

1. highest when t_i occurs many times within a small software number of entities, this lends a high discriminating power to those entities,
2. lower when t_i occurs fewer times in a software entity, or occurs in many software entities,
3. lowest when t_i occurs in virtually all entities.

The term-by-document matrix achieved by applying tf-idf (i.e., the *Term-by-Document Matrix* object) is the output of the Extracting phase and is the input object of the *LSI* phase, which computes the concept space by adopting the LSI technique [DDL+90]. LSI is an extension of the vector space model (VSM) assuming that there is some latent structure in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. In particular, LSI computes this latent structure applying a singular value decomposition (SVD) to a term-by-document matrix. The computed latent structure is composed by: a term vector matrix *T* (constituting left singular vectors), a diagonal matrix *S* (constituting singular values), and an entity vector matrix *E* (constituting right singular vectors). Therefore, the term-by-document matrix can be obtained as follows: $\mathrm{T} \bullet S \bullet E^T$. SVD can be geometrically interpreted: terms and documents could be represented as vectors in the d space (i.e., the singular values of the dimensionality reduction of the latent structure of the software entities) of the underlying concepts. Indeed, LSI allows a simple strategy for the optimal approximate fit using smaller matrices and using only a subset of d concepts. The choice of an appropriate value of d is an open issue (e.g., cut point, cut percentage, constant threshold, or variable threshold). In this paper we calculate the number of singular values according to the Guttman–Kaiser criterion [Gut54, Kai60]. This criterion considers the diagonal matrix S of the singular values, which are a kind of eigenvalue, of the term-by-document matrix in descending order. The value of d is the number of singular values in S greater than 1. Therefore, d is used to compute the truncated matrices T_d, S_d and E_d (i.e., the persistent *LSI Space* object). The *Computing Approximated Space* phase multiplies these matrices to obtain the *Approximated Term-by-Document Matrix* object.

Since the approach aims at recovering the architectures of evolving software systems we could have that in a version some entities are removed and/or added from the previous version. Furthermore, it can also happen that a given software entity is modified through subsequent versions. Currently, our approach only manages the addition and removal of software entities. Future work will be devoted to handle modifications of software entities through subsequent versions of a given software system. The phase is in charge of incrementally adding and removing entities from the preexisting latent semantic space is *folding-in/out*. This phase takes as input the LSI Space, *added entities*, and *removed entities* objects. The latter two objects are produced by the *Computing Entity Differences* phase that takes as input the entities of the base version of the software system and the entities of a subsequent version of the same system.

The folding-in/out phase uses: (i) fold-out to remove the columns from the E_d matrix corresponding to the entities that are deleted from the base version of the software system to its new version; (ii) fold-in to incrementally add new software entities and the corresponding lexical content to preexisting latent semantic space considering all the terms that are both in the newly added entities and in the base version of the software system. The folding-in/out phase uses the fold-in method proposed in [Wil07] to avoid re-computing SVD each time a change is made to the term-by-document matrix. If on one side the folding-in method can be computed very fast, on the other side, its accuracy may degrade very quickly. In the latter case the term-by-document matrix has to be calculated considering all the documents to index (see the left and side of Fig. 1, namely the *Indexing or reindexing* swimlane). The output the phase is the Approximated Term-by-Document Matrix object.

Computing Similarity is in charge of computing the similarity between each pair of entities using a measure based on the cosine similarity between their corresponding vectors in the approximated latent semantic space (i.e., approximated term-by-document matrix object). In particular, we compute the dissimilarity between the entities e_1 and e_2 as follows:

$$d(e_1, e_2) = 1 - \frac{V e_1 \bullet V e_2}{|V e_1| \bullet |V e_2|} \tag{2}$$

```
     input:
       D := {d1, ..., dn, d1, ..., dn} //the dissimilarity matrix
     output:
       s //partition of the documents maximizing the NED function

     procedure clustering-software-entities
1      set S := {s2, ..., sn/2}
2      for each k in 2:n/2
3          set I := {d1, ..., dn}
4          set C := {c1, ..., ck} to initial value //random selection of d1,..., dn
5          set m : I → C //cluster membership function
6          for each dj∈ I
7              m(ij) := arg min D(dj, cs)
                        s ∈ {1..k}
9          end
10         while m has changed
11             for each j ∈ {1..k}
12                 cj := average of di whose m(i) = j
13             end
14             for each dj ∈ I
15                 m(ij) := arg min D(ij, cs)
                            s ∈ {1..k}
16             end
17         end
18         sk := m
19         if NED(sk) = 1 then return sk
20     end
21     set index := arg max NED(st)
                    t ∈ {2..n/2}
22     return sindex
23   end
```

Fig. 2. The pseudocode of the phase clustering software entities

where Ve_1 and Ve_2 are vectors corresponding to the entities e_1 and e_2 in the latent semantic space. The adopted measure is 0 in case the cosine between the vectors of the software entities is 1 (their latent structure is the same). The output of computing similarity is a similarity matrix (i.e., the *Entity Similarity Matrix* object). Both *Indexing or reindexing* and *folding-in/out* have the same asymptotic complexity (cubic time in the worst case). However, the hidden constant of the asymptotic complexity of the folding-in/out sub-process is smaller than the one of sub-process Indexing or reindexing.

3.2. Clustering software entities

Figure 2 shows the pseudo-code of this phase. In particular, it includes the k-means clustering algorithm, which defines a centroid for each cluster to identify and iteratively refines the centroids minimizing the average distances/dissimilarity of the software entities to their closest centroids. At each iteration, the clusters are built by assigning each cluster to the closest centroid. The k-means variant used here minimizes the sum of the dissimilarity and performs two subsequent phases, namely *building* and *swap*. In the building phase, the algorithm looks for a good initial set of centroids, which have to be placed as much as possible far away from each other. In the swap phase the algorithm finds a local minimum for the objective function, that is, a solution such that there is no single switch of a centroid that minimizes the sum of the dissimilarity.

The k-means algorithm is applied more than once (see instruction 2) on the same dissimilarity matrix. In particular, the clustering algorithm is applied choosing as priori fixed number of clusters ranging from 2 to $n/2$, where n is the number of the software entities to be analyzed. The partition provided as output is the one that optimizes the NED function (see the instructions from 19 to 22).

4. The eclipse plug-in

The clustering based approach has been implemented within a software system prototype as an Eclipse plug-in. Figure 3 shows a UML Package Diagram representing the logical architecture of the plug-in. The *GUI* component (see Fig. 4a) enables the software engineer to select the system to be partitioned in subsystems and to visualize the identified clusters (see Fig. 4b). The component also provides information on the different iterations of the clustering process. For example, the plug-in enables the software engineer to get the value of the dimensionality reduction of the latent structure of the software system and the value of the clustering algorithm configuration maximizing the NED function. The plug-in also allows the software engineer to manually add, move, and remove software entities from the clusters automatically identified, thus improving the overall quality of the identified partition. Further on, the software engineer can decide to incrementally perform the indexing (i.e., the folding-in/out feature). Conversely, he/she can re-perform the indexing when required.

The dissimilarity between all the pairs of software entities is computed by computing dissimilarity. In case the software system to analyze is implemented in C or C++, the component *C/C++ Preprocessor* is used to include in each source file all the needed header files. This component has been implemented in C and has been integrated by JNI (Java Native Interface). The software component in charge of performing the normalization phase (see Sect. 3.1) is the *Text Normalizer* component. On the other hand, then software component *LSI Engine* computes the cosine similarity between the pairs of vector representations of all the normalized source code files. As this computation could be expensive, the prototype stores the dissimilarity matrix on the prototype file system before providing it as input to the software component implementing the k-means clustering algorithm (i.e., Grouping Software Entities). The LSI Engine component also implements the fold-in and fold-out mechanisms to update the term-by-document matrix. The components LSI Engine and Grouping Software Entities have been implemented in R.[2]

To integrate the software components implemented in R we have used the Rserve TCP/IP server. This server is available under GPL license and can be downloaded from http://rosuda.org/Rserve/down.shtml. Despite the availability of simpler methods to integrate R software components with Java code, we have decided to use Rserve since it offers the possibility of distributing the computation on different nodes on the Web.

It is worth mentioning that the plug-in is not available for free downloading. However, the interested reader may contact one of the authors to get and use it for research purposes.

5. Experimental setting

In this section, we describe the design underlying the presented empirical investigation.

5.1. Experimental hypotheses

The assessment of the overall quality of the clustering based process is generally a critical issue. We consider here three criteria that are widely adopted in the software maintenance field:

a) *authoritativeness*: It regards the resemblance between the software clusters identified by the tool and an authoritative partition (i.e., the decomposition performed by the developers or by the original software architect). The clusters produced by the approach should resemble as much as possible the groups of entities within the authoritative partition.
b) *stability*: It concerns the persistence of the partition structure of two consecutive versions of an evolving software system. A clustering based approach should produce similar partitions in case of small and incremental changes between successive versions.
c) *non-extremity cluster distribution*: It aims to investigate whether the approach identifies a partition whose clusters have a size distribution exhibiting no extreme values. A good approach should avoid clusters with too many or too few software entities.

[2] R is a freely available language and environment for statistical computing and graphics available at http://cran.r-project.org.

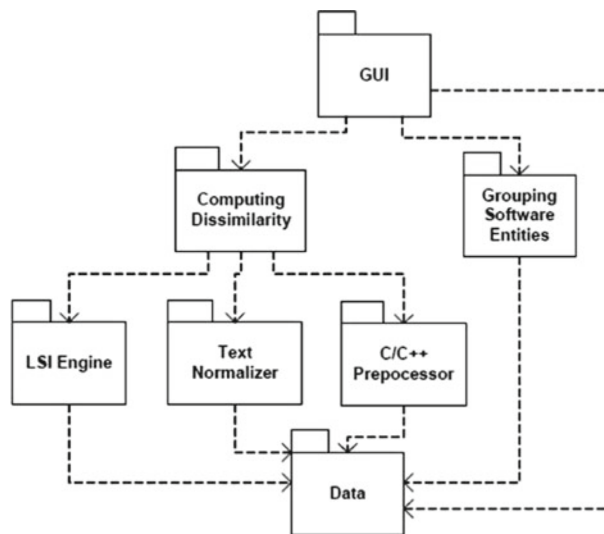

Fig. 3. Logical architecture of the prototype

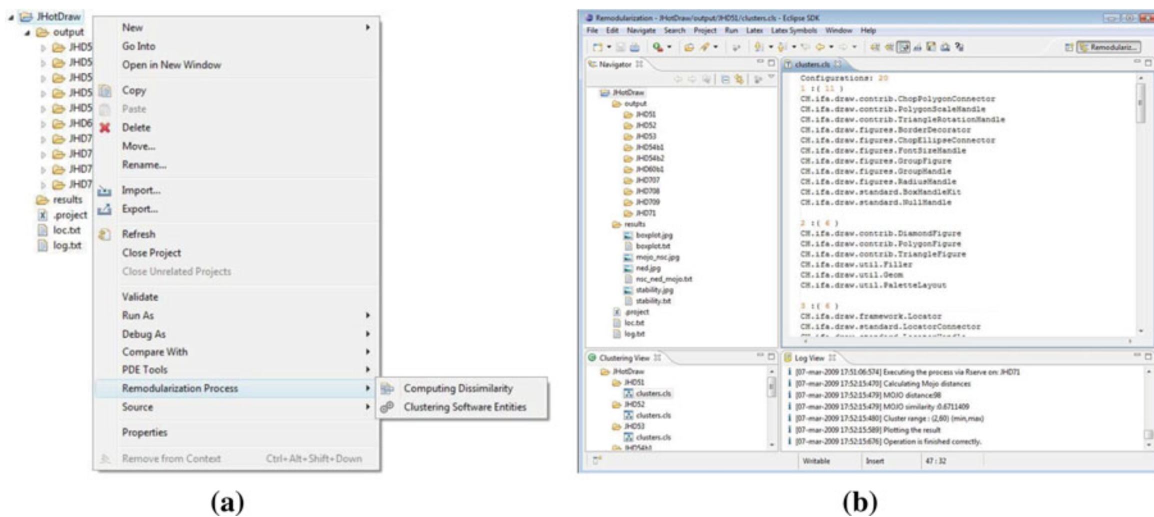

Fig. 4. The Eclipse plug-in

Authoritativeness aims at assessing the quality of a partition, while the remaining criteria are used to assess two desirable characteristics that a clustering based approach should have. In fact, a good approach should give consistent mappings for different consecutive versions of an understudy software system and should avoid clusters with too many or too few software entities [WH05]. According to the considered criteria we have formulated the following research question:

RQ1: Does our proposal outperform other clustering based approaches with respect to authoritativeness, stability, and non-extremity cluster distribution?

We also aimed at verifying whether the use of fold-in and fold-out influences clustering results. To this end, we compared the overall quality of the clustering results (in terms of Authoritativeness, Stability, and Non-extremity cluster distribution). We defined and investigated therefore the following research question:

Table 2. Selected systems

Systems	Language	Vers.	Files	KLOC	KLoCom.	Available at	Licence
JEdit	Java	18	394–561	94–116	26–34	www.jedit.org	GPL 2.0
OpenSSL	C/C++	20	711–830	218–281	57–68	www.openssl.org	Apache-style
PostGreSQL	C	30	826–1128	197–387	45–79	www.postgresql.org	Similar to BSD and MIT
JUnit	Java	17	31–154	2–5	0.4–2	www.junit.org	Common Public License
JHotDraw	Java	10	144–441	9–55	2–16	www.jhotdraw.org	LGPL

RQ2: Are clustering results affected by the adoption of fold-in and fold-out?

The third research question is to assess whether the use of fold-in and fold-out practically reduces the *computation time* (i.e., improves the performances) in the context of software evolution.

RQ3: Is there a computational difference when our approach uses or not fold-in and fold-out?

To understand whether fold-in and fold-out represent a suitable trade-off between clustering results and computation time we also performed an analysis based on the computational efficiency. Therefore, the following research question has been defined and investigated:

RQ4: Is our proposal more efficient when fold-in and fold-out mechanisms are employed?

The investigation of RQ4 also provides an indication on whether the use of fold-in and fold-out may represent a viable alternative to reduce energy consumption in the software clustering field.

5.2. Application selection

We have selected five open source software systems (three Java and two C/C++) to conduct a preliminary comparison between our proposal and the clustering based approaches investigated in [BG09, CDMS10, WH05]. Table 2 reports some descriptive statistics and details on the studied software systems. In particular, the first and the second columns show the names of the analyzed software systems and the programming languages used to implement them, respectively. The third column shows the number of versions analyzed in the empirical investigation. The maximum and minimum numbers of source files within all the analyzed versions are shown in the fourth column, while the minimum and maximum number of line of code (KLOC) and line of comments (KLoCom.) are reported on the fifth and sixth columns, respectively. The seventh column reports the addresses where details on the selected software systems including their source code can be found. The distribution licenses of the software systems are shown in last column.

In the following, we briefly describe the considered software systems:

JEdit is a programmer's text editor with an extensible plug-in architecture.

OpenSSL is a toolkit implementing the network protocols SSL and TLS and a general-purpose cryptography library.

PostGreSQL is a powerful relational database management system.

JUnit is a programmer-oriented testing framework for Java.

JHotDraw is a Java GUI framework for technical and structured Graphics.

5.3. Metrics

To evaluate the authoritativeness of our clustering based approach we used the following measure:

$$Authoritativeness(ToolPart, AuthPart) = 1 - \frac{Mojo(ToolPart, AuthPart)}{n} \quad (3)$$

where *ToolPart* is the partition automatically identified by our tool, while *AuthPart* is the authoritative partition and n is the number of software entities to be clustered. The Authoritativeness measure is based on the monodirectional MoJo distance [WT03] that is defined as the minimum number of join and move operations to turn a partition into another one. In particular, we used here the polynomial version of MoJo that is available for downloading at http://www.cse.yorku.ca/~bil/downloads.

AuthPart is often hard to find because either the architectural documentation of the software system to analyze is lacking or the documentation is outdated due the execution of a number of maintenance operations [Ton01]. Therefore, we addressed these issues using the same approach as in [BG09, CDMS10, WH05], where the original source folder structure of the system is exploited to:

a) create the subsystem hierarchy based on the directory structure (each directory represents a subsystem);
b) merge a subsystem with its parent subsystem in case it contains a number of source files less than or equal five;
c) create a cluster with each subsystem in the subsystem hierarchy.

The reader may object to the fact that this method to identify the authoritative partition may affect the assessment of approach authoritativeness. In fact, whether the folder structure does not reflect the software system architecture, the adopted measure indicates how "bad" the partitions identified on this system are. However, this method has been widely adopted in the literature and represents the only way to compare the results achieved by applying our approaches with the ones presented in [BG09, CDMS10, WH05]. It is also worth mentioning that the more the Authoritativeness value is, the more is the resemblance between the software clusters identified by the tool and an authoritative partition.

To assess the stability, we have used a Stability measure that is similarly to the one employed to evaluate the authoritativeness of the clustering result. The only difference is that it takes as input parameters the partitions automatically identified by the tool on two subsequent versions. As it is easy to imagine, the more the Stability value between subsequent versions is, the more is the similarity between the partitions of these versions.

We planned to assess the non-extremity of cluster distribution through the NED measure, which is defined in the Eq. 1. The more the NED value is,γ the more non-extreme is the size distribution of the clusters.

Finally, *Time* is use to measure the computational time needed to perform the clustering process and is expressed in terms of seconds. In order to verify the research question RQ4, the Efficiency measure has been used:

$$Efficiency(ToolPart, AuthPart) = \frac{Authoritativeness(ToolPart, AuthPart)}{Time(ToolPart)}\% \tag{4}$$

where *Authoritativeness* is the computed as shown in the Eq. 3 and *Time* is the number of seconds needed to get a partition by using or not fold-in and fold-out. Regarding Efficiency, the more its value is, the more efficient is the computation of a partition.

5.4. Data collection and experimental planning

In order to collect the results, we executed the approach on each version of the selected software systems without applying fold-in and fold-out. In particular, for each version of a software system we executed the tool, collected the computation time, and then computed the values of authoritativeness, stability, NED, and efficiency.

As far as the execution of the approach using fold-in and fold-out is concerned, we computed the partition on the first version of each system. Successively, the subsequent versions of each software system have been analyzed by applying the fold-in and fold-out mechanisms with respect to the first own version. All the partitions automatically identified by the tool and the time needs to obtain them have been collected and used to compute the values of Authoritativeness, Stability, NED, and Efficiency.

For the experimentation we used a laptop equipped by a 2.53 GHz Intel Core 2 Duo with 4 GB of RAM and Windows MS Seven 64bit as operating system.

6. Experimental results and discussion

We first present the general findings achieved by applying or not fold-in and fold-out. Successively, we discuss the results in terms of the defined research questions. A discussion of the threats that may affect the validity of the achieved results concludes the section.

6.1. General findings

The values of Authoritativeness, Stability, NED, and Efficiency obtained on all the considered versions of the five selected software systems are summarized in the Tables 3–7, respectively. In particular, for each table the first column shows the identifiers of the analyzed versions, while the distribution versions are reported in the second column. The values of Authoritativeness (the third column), Stability (the fourth column), and NED (the fifth column) are reported too. The sixth and seventh columns of each table show the computational time for partition the software entities and the Efficiency values, respectively. The last column of each table reports the number of clusters identified by the tool. It is worth noting that for the first version of each system, the tables do not report Stability values. This is due to Stability that is computed between two subsequent versions and then its value cannot be computed on the first version.

Table 3. JEdit results

ID	Vers.	Auth.	Stab.	NED	Time	Effic.	Clust.
(a) Without fold-in and fold-out							
V1	4.2.14	0.868	–	1	153.9	0.56	7
V2	4.2.15	0.87	0.99	1	147.4	0.59	7
V3	4.3.1	0.866	0.99	1	156.5	0.55	7
V4	4.3.2	0.853	0.96	1	171.0	0.50	7
V5	4.3.3	0.858	0.78	1	177.7	0.48	6
V6	4.3.4	0.859	0.98	1	179.6	0.48	6
V7	4.3.5	0.83	0.72	1	214.4	0.39	9
V8	4.3.6	0.85	0.81	1	233.3	0.36	8
V9	4.3.7	0.83	0.75	1	238.6	0.35	9
V10	4.3.8	0.836	0.93	1	249.0	0.34	9
V11	4.3.9	0.833	0.93	1	249.3	0.33	10
V12	4.3.10	0.834	0.85	1	267.1	0.31	10
V13	4.3.11	0.83	0.9	1	272.0	0.31	11
V14	4.3.12	0.828	0.96	1	275.7	0.30	11
V15	4.3.13	0.832	0.83	1	261.4	0.32	10
V16	4.3.14	0.82	0.92	1	276.1	0.30	10
V17	4.3.15	0.834	0.89	1	285.2	0.29	8
V18	4.3.16	0.83	0.94	1	286.9	0.29	9
(b) With fold-in and fold-out							
V1	4.2.14	0.868	–	1	153.9	0.56	7
V2	4.2.15	0.87	1	1	70.17	1.24	6
V3	4.3.1	0.81	0.79	1	80.11	1.01	7
V4	4.3.2	0.81	0.81	1	92.13	0.88	8
V5	4.3.3	0.815	0.74	1	82.15	0.99	7
V6	4.3.4	0.814	0.95	1	97.74	0.83	7
V7	4.3.5	0.818	0.74	1	119.7	0.68	8
V8	4.3.6	0.81	0.71	1	130.9	0.62	9
V9	4.3.7	0.8	0.75	1	137.4	0.58	9
V10	4.3.8	0.79	0.86	1	140.6	0.56	9
V11	4.3.9	0.79	0.86	1	141.6	0.56	9
V12	4.3.10	0.786	0.82	1	150.5	0.52	10
V13	4.3.11	0.786	0.8	1	152.9	0.51	10
V14	4.3.12	0.79	0.7	1	168.1	0.47	11
V15	4.3.13	0.77	0.75	1	163.2	0.47	11
V16	4.3.14	0.78	0.78	1	171.8	0.45	11
V17	4.3.15	0.8	0.88	1	177.6	0.45	9
V18	4.3.16	0.8	0.86	1	186.5	0.43	9

Table 4. OpenSSL results

ID	Vers.	Auth.	Stab.	NED	Time	Effic.	Clust.
(a) Without fold-in and fold-out							
V1	0.9.7d	0.84	–	0.97	875	0.096	36
V2	0.9.7e	0.86	0.99	0.96	947	0.091	40
V3	0.9.7f	0.857	0.99	0.97	926.8	0.092	41
V4	0.9.7g	0.82	1	0.97	929.7	0.088	42
V5	0.9.7h	0.82	0.98	0.97	989.4	0.083	36
V6	0.9.7i	0.82	1	0.97	1007.2	0.081	36
V7	0.9.7j	0.82	0.99	0.97	1001.3	0.082	36
V8	0.9.7k	0.82	1	0.97	1035.2	0.079	36
V9	0.9.7l	0.822	1	0.97	1026.3	0.080	36
V10	0.9.7m	0.825	0.99	0.98	1029.1	0.080	33
V11	0.9.8	0.83	0.97	0.97	1127.5	0.074	39
V12	0.9.8a	0.83	0.99	0.97	1142	0.073	39
V13	0.9.8b	0.83	0.99	0.97	1145.9	0.072	39
V14	0.9.8c	0.828	1	0.97	1163.3	0.071	40
V15	0.9.8d	0.828	1	0.98	1181.6	0.070	40
V16	0.9.8e	0.832	0.99	0.97	1182.3	0.070	41
V17	0.9.8f	0.833	0.99	0.97	1160.4	0.072	41
V18	0.9.8g	0.83	1	0.97	1151.8	0.072	41
V19	0.9.8h	0.83	0.98	0.97	1206.8	0.069	42
V20	0.9.8i	0.832	1	0.98	1214.4	0.069	42
(b) With fold-in and fold-out							
V1	0.9.7d	0.84	–	0.97	875	0.096	36
V2	0.9.7e	0.79	0.76	0.97	294.5	0.268	32
V3	0.9.7f	0.79	0.99	0.97	296.6	0.266	32
V4	0.9.7g	0.79	0.92	0.97	297.4	0.266	32
V5	0.9.7h	0.78	0.79	0.976	333.6	0.234	32
V6	0.9.7i	0.78	1	0.976	329.8	0.237	32
V7	0.9.7j	0.78	0.96	0.976	343.8	0.227	32
V8	0.9.7k	0.78	0.935	0.976	346.3	0.225	32
V9	0.9.7l	0.78	1	0.976	347.8	0.224	32
V10	0.9.7m	0.78	0.96	0.976	345	0.226	32
V11	0.9.8	0.79	0.73	0.977	420.2	0.188	35
V12	0.9.8a	0.79	0.82	0.977	418.6	0.189	35
V13	0.9.8b	0.79	1	0.977	419.6	0.188	35
V14	0.9.8c	0.785	1	0.98	410.2	0.191	34
V15	0.9.8d	0.785	1	0.98	408	0.192	34
V16	0.9.8e	0.79	0.95	0.98	414.3	0.191	34
V17	0.9.8f	0.788	0.84	0.98	413.2	0.191	34
V18	0.9.8g	0.788	1	0.98	415.3	0.190	34
V19	0.9.8h	0.79	0.72	0.983	445.8	0.177	35
V20	0.9.8i	0.79	0.98	0.983	462.9	0.171	35

In general, the partitions identified by the tool prototype match the partitions of the original developers. For example, the Authoritativeness values obtained on all the versions of the analyzed systems range between 0.77 (V30 of PostGreSQL) and 0.93 (V1 of JUnit). In case, the fold-in and fold-out mechanisms were used the Authoritativeness values range between 0.71 (V30 of PostGreSQL) and 0.92 (V2 of JHotDraw).

We also observed that on smaller software systems (e.g., JEdit and OpenSSL) the approach performs better in terms of authoritativeness. No difference was observed with respect to the programming language of the analyzed software systems. In fact, similar results have been obtained on JEdit and OpenSSL, the two software systems with comparable size.

Further, the difference in terms of Authoritativeness values, when using or not fold-in and fold-out, ranges from 6.7% (V3 of OpenSSL) to −3.3% (V8 of JUnit and V2 of JHotDraw), thus indicating that slightly better authoritativeness results are achieved without fold-in and fold-out. However, in some cases the authoritativeness is slightly better when fold-in and fold-out are used (e.g., this happens on V8 JUnit).

The results also indicate that the approach is not influenced by small and incremental changes (both using and not fold-in and fold-out) between consecutive versions of the studied software systems. Some considerations are however due on the programming language used to implement the analyzed software systems. In particular, we can observe that slightly Stability values have been achieved on the C/C++ systems (see Tables 4, 5).

Table 5. PostGreSQL results

ID	Vers.	Auth.	Stab.	NED	Time	Effic.	Clust.
(a) Without fold-in and fold-out							
V1	6.4.2	0.784	–	1	574.6	0.136	9
V2	6.5	0.779	0.97	1	474	0.164	9
V3	6.5.1	0.782	0.98	1	647.3	0.121	9
V4	6.5.2	0.778	0.94	1	519.98	0.150	9
V5	6.5.3	0.778	1	1	517.4	0.150	9
V6	7.0	0.773	0.86	1	799.69	0.097	14
V7	7.0.1	0.773	1	1	829.1	0.093	14
V8	7.0.2	0.773	1	1	807.1	0.096	14
V9	7.0.3	0.773	0.99	1	807.16	0.096	14
V10	7.1	0.783	0.86	1	834.6	0.094	19
V11	7.1.1	0.784	1	1	840.25	0.093	19
V12	7.1.2	0.784	1	1	841.8	0.093	19
V13	7.1.3	0.784	1	1	842.5	0.093	19
V14	7.2	0.78	0.90	0.995	895.8	0.087	22
V15	7.2.1	0.779	0.99	0.995	907.5	0.086	22
V16	7.2.2	0.78	0.99	0.995	894.3	0.087	22
V17	7.2.3	0.779	0.99	0.995	890.86	0.087	22
V18	7.2.4	0.779	1	0.995	892.16	0.087	22
V19	7.2.5	0.793	0.96	1	956.8	0.083	17
V20	7.2.6	0.793	1	1	896	0.089	17
V21	7.2.7	0.793	1	1	916.13	0.087	17
V22	7.2.8	0.793	1	1	934.3	0.085	17
V23	7.3	0.78	0.9	1	970.3	0.080	19
V24	7.3.1	0.78	0.96	1	913.4	0.085	18
V25	7.3.2	0.78	0.99	1	971.3	0.080	18
V26	7.3.3	0.78	0.99	1	972	0.080	18
V27	7.3.4	0.78	1	1	971.1	0.080	18
V28	7.3.5	0.78	1	1	954.7	0.082	18
V29	7.3.6	0.78	1	1	957.8	0.081	18
V30	7.4	0.77	0.89	1	828.72	0.093	19
(b) With fold-in and fold-out							
V1	6.4.2	0.784	–	1	574.6	0.132	9
V2	6.5	0.75	0.87	1	148.4	0.505	9
V3	6.5.1	0.77	0.95	1	161.3	0.477	9
V4	6.5.2	0.74	0.99	1	154	0.481	10
V5	6.5.3	0.74	1	1	169.7	0.436	10
V6	7.0	0.72	0.85	1	197.1	0.365	14
V7	7.0.1	0.72	0.99	1	195.7	0.368	14
V8	7.0.2	0.72	1	1	194	0.371	14
V9	7.0.3	0.72	1	1	193.6	0.372	14
V10	7.1	0.73	0.82	1	229.8	0.318	14
V11	7.1.1	0.73	1	1	232.5	0.314	14
V12	7.1.2	0.73	1	1	234.5	0.311	14
V13	7.1.3	0.73	1	1	238.7	0.306	14
V14	7.2	0.76	0.83	0.99	638.1	0.119	15
V15	7.2.1	0.76	1	0.99	633.7	0.120	15
V16	7.2.2	0.74	0.97	0.99	613.7	0.121	16
V17	7.2.3	0.74	1	0.99	605.9	0.122	16
V18	7.2.4	0.74	1	0.99	601.9	0.123	16
V19	7.2.5	0.74	1	0.99	603.6	0.123	16
V20	7.2.6	0.76	0.97	0.99	632	0.120	15
V21	7.2.7	0.76	1	0.99	635.2	0.120	15
V22	7.2.8	0.76	1	0.99	622.2	0.122	15
V23	7.3	0.74	0.76	0.978	807.5	0.092	28
V24	7.3.1	0.73	1	0.978	764.4	0.095	28
V25	7.3.2	0.74	0.99	0.976	764.7	0.097	29
V26	7.3.3	0.74	1	0.976	766.23	0.097	29
V27	7.3.4	0.74	1	0.976	761.3	0.097	29
V28	7.3.5	0.74	1	0.976	762	0.097	29
V29	7.3.6	0.74	1	0.976	765.1	0.097	29
V30	7.4	0.71	0.75	0.95	996.6	0.071	47

Table 6. JUnit results

ID	Vers.	Auth.	Stab.	NED	Time	Effic.	Clust.
(a) Without fold-in and fold-out							
V1	3.4	0.93	–	1	5.5	17.01	2
V2	3.5	0.89	0.95	1	6.53	13.61	2
V3	3.6	0.89	1	1	6.45	13.82	2
V4	3.7	0.89	0.95	1	6.185	14.37	2
V5	3.8	0.893	1	1	6.514	13.72	2
V6	3.8.1	0.87	0.915	1	6.42	13.59	2
V7	3.8.2	0.897	0.918	1	6.74	13.32	2
V8	4.0	0.877	0.92	1	8.95	9.80	2
V9	4.1	0.86	0.73	1	10.66	8.04	2
V10	4.2	0.86	0.97	1	10.91	7.89	2
V11	4.3.1	0.86	0.97	1	10.84	7.94	2
V12	4.4	0.85	0.97	1	16.2	5.27	2
V13	4.5	0.82	0.69	1	19.7	4.18	3
V14	4.6	0.81	1	1	21.1	3.84	3
V15	4.7	0.83	0.74	1	24.47	3.41	2
V16	4.8	0.84	1	1	23.56	3.56	2
V17	4.8.1	0.84	1	1	24.06	3.48	2
(b) With fold-in and fold-out							
V1	3.4	0.93	–	1	5.5	17.01	2
V2	3.5	0.87	0.89	1	3.31	26.18	2
V3	3.6	0.867	0.98	1	3.45	25.20	2
V4	3.7	0.87	0.977	1	3.27	26.50	2
V5	3.8	0.872	0.85	1	3.6	24.23	2
V6	3.8.1	0.872	1	1	3.63	24.03	2
V7	3.8.2	0.877	0.887	1	3.82	22.97	2
V8	4.0	0.91	0.95	1	7.43	12.22	2
V9	4.1	0.86	0.86	1	8.8	9.74	2
V10	4.2	0.87	0.83	1	8.84	9.88	2
V11	4.3.1	0.87	0.99	1	8.78	9.95	2
V12	4.4	0.85	0.98	1	14.26	5.99	2
V13	4.5	0.84	0.89	1	16.58	5.06	2
V14	4.6	0.83	0.98	1	17.53	4.75	2
V15	4.7	0.77	0.8	1	20.01	3.85	3
V16	4.8	0.78	0.96	1	20.24	3.82	3
V17	4.8.1	0.77	1	1	20.64	3.74	3

Table 7. JHotDraw results

ID	Vers.	Auth.	Stab.	NED	Time	Effic.	Clust.
(a) Without fold-in and fold-out							
V1	5.1	0.882	–	1	22.3	3.96	3
V2	5.2	0.86	0.76	1	25.8	3.34	4
V3	5.3	0.877	0.75	1	35.7	2.46	4
V4	5.4b1	0.88	0.78	1	64.03	1.37	5
V5	5.4b2	0.86	0.69	1	66.4	1.29	5
V6	6.0b1	0.87	0.95	1	67.01	1.31	4
V7	7.0.7	0.84	0.92	1	65.6	1.29	5
V8	7.0.8	0.87	0.87	1	80.7	1.08	4
V9	7.0.9	0.78	0.73	1	147.5	0.53	15
V10	7.1	0.8	0.81	1	136.6	0.59	8
(b) With fold-in and fold-out							
V1	5.1	0.882	–	1	22.3	3.96	3
V2	5.2	0.92	0.78	1	8.7	11.26	2
V3	5.3	0.91	0.77	1	15.5	5.85	3
V4	5.4b1	0.83	0.74	1	35.6	2.33	7
V5	5.4b2	0.827	0.64	1	35.63	2.32	7
V6	6.0b1	0.84	0.58	1	52.77	1.59	6
V7	7.0.7	0.87	0.94	1	51.7	1.68	4
V8	7.0.8	0.85	0.89	1	57.57	1.49	5
V9	7.0.9	0.76	0.83	1	91.4	0.84	11
V10	7.1	0.815	0.89	1	90.8	0.90	8

Table 8. Summary of the results on the Java systems

	Authoritativeness	Stability	NED
Bittencourt and Guerrero [BG09]			
eb	0.20–0.60	0.95–1.00	near zero
km	0.40–0.80	0.35–0.65	0.30–1.00
mq	0.30–0.70	0.35–0.95	0.10–0.50
dsm	0.35–0.75	0.45–0.80	0.10–0.70
Corazza et al. [CDMS10]			
K-Med.	0.47–0.83	–	0.95–1.00
K-Med.+EM	0.50–0.83	–	0.96–1.00
Corazza et al. [CDMMS11]			
6 Zones +Freq EM	0.58–0.79	–	0.59–0.78
Our approach			
Without fold-in and fold-out	0.78–0.93	0.69–1.00	1.00–1.00
With fold-in and fold-out[a]	0.76–0.92	0.58–1.00	1.00–1.00

To be fair as much as possible we did not consider the values of Authoritativeness, Stability, and NED achieved on the first version of the analyzed software systems. The motivation is that on the first version the fold-in and fold-out mechanisms are actually not used.

In fact, the Stability values that mostly occur in these tables are very close to 1. However, the Stability values obtained on JEdit and JUnit (Tables 3, 6, respectively) are not so far from the ones obtained on the analyzed versions of OpenSSL and PostGreSQL. A further analysis revealed that some of the classes of these versions underwent maintenance operations that sensibly modified their source code. The trend is nearly the same also applying fold-in and fold-out.

The obtained results also show that our approach enabled to get very high NED values (are mostly 1 or very close to 1 both using and not using fold-in and fold-out), thus indicating that all the identified partitions do not exhibit non-extreme values in terms of size. According to the obtained results, we can conclude that our approach generally produces appreciable results on all the Java and C/C++ selected software systems. Finally, the results show that generally the computation time is lower and the Efficiency values are greater when using fold-in and fold-out.

6.2. RQ1: Does our proposal outperform other clustering based approaches?

Table 8 summarizes the results presented in [BG09, CDMS10, CDMMS11] while Table 9 reports the results highlighted in [WH05]. The rationale for grouping the data in such a way is that in [BG09, CDMS10, CDMMS11] the authors analyzed both JEdit and JUnit, while in [WH05] the authors investigated the effect of the selected clustering algorithms on a set of software systems including OpenSSL and PostGreSQL. It is also worth recalling that structural based information was used in [BG09] and [WH05] to group software entities, while the approaches proposed in [CDMS10, CDMMS11] are lexical based (see Table 1).

The Authoritativeness values obtained on the Java systems is higher with respect to the ones of the other clustering algorithms/approaches both using and not using fold-in and fold-out. Another difference regards the variability of the results. In fact, as shown in Table 8 the interval of the Authoritativeness values is smaller in case of our approach (0.78–0.93 and 0.76–0.92 with and without fold-in and fold-out, respectively). On both the C/C++ software systems, SL75 and SL90[3] perform better than our approach. This is acceptable as the stability of the approaches is comparable and our approach outperforms these algorithms with respect to the non-extremity of the cluster distribution. In fact, the NED values are nearly 0 for SL 90 and range from 0 to 0.04 for SL 75, while in our case the worst obtained result is 0.95 (V30 of PostGreSQL).

Regarding the stability on the Java systems, we observed that our proposal outperforms all the other clustering algorithms with the exception of the edge betweenness algorithm (eb). In particular, our approach produced Stability values between 0.69 and 1.00, while (eb) from 0.95 to 1.00. This is an acceptable drawback since (eb) produced worse NED (near zero) and Authoritativeness values (ranging from 0.20 to 0.60). Similar conclusions can be drawn when fold-in and fold-out were applied. On OpenSSL and PostGreSQL, we observed that the clustering algorithms that outperform our approach in terms of Stability obtained worse NED values.

[3] An agglomerative clustering algorithm (based on the Jaccard coefficient and the single linkage rule) using 0.75 and 0.90 as cutting points.

Table 9. Summary of the results on OpenSSL and PostGreSQL

	Authoritativeness	Stability	NED
Wu *et al.* [WH05]			
Bunch	0.61–0.76	0.24–0.61	0.38–1.00
CL90	0.56–0.60	0.74–0.95	0.67–0.82
CL75	0.57–0.59	0.79–0.97	0.37–0.54
ACDC	0.70–0.72	0.96–0.99	0.20–0.36
SL75	0.89–0.90	0.94–1.00	0.00–0.04
SL90	0.92–0.93	0.99–1.00	Almost zero
Our approach			
Without fold-in and fold-out	0.77–0.86	0.86–1.00	0.96–1.00
With fold-in and fold-out[a]	0.71–0.79	0.72–1.00	0.95–1.00

To be fair as much as possible we did not consider the values of Authoritativeness, Stability, and NED achieved on the first version of the analyzed software systems. The motivation is that on the first version the fold-in and fold-out mechanisms are actually not used.

Our approach always performs better in terms of NED. In fact, on JEdit the approach enabled us to get 1 as NED values for each analyzed version. Only the approach proposed by Corazza et al. produced comparable results. However, we obtained higher Authoritative values. With respect to the C/C++ software systems our proposal outperforms other algorithms.

Concluding, we can positively answer RQ1 both using fold-in and fold-out. This is because our approach is a good trade-off in case we need to optimize all the considered criteria and we would avoid human decisions to identify the best partition of software entities into clusters. A possible reason for explaining the fact that our approach outperforms the structural based approaches on the selected software systems is that the contribution of lexical information is more relevant than structural information to group source files that implement related functionality. On the other hand, the better results with regard to the lexical based approach presented in [CDMS10] could be related to the used information retrieval technique and the applied clustering algorithm. To increase our awareness, special conceived investigations are, however, needed.

6.3. RQ2: Are clustering results affected by the adoption of fold-in and fold-out?

The experimentation indicates that the results are slightly affected in case fold-in and fold-out are used. In particular, for the larger systems (i.e., JEdit, OpenSSL, and PostGreSQL), we can observe that slightly better Authoritativeness values were achieved when fold-in and fold-out are not applied (see Tables 3, 4, and 5). A similar trend can be also observed on the Stability values. Regarding the non-extremity of the cluster distribution, the NED values are nearly the same on each version.

The results achieved on JUnit and JHotDraw (see results shown in Tables 6 and 7) indicate that in some cases the use of fold-in and fold-out produced more authoritative and stable partitions (see for example the version V8 of JUnit). Similarly to JEdit, OpenSSL, and PostGreSQL, the achieved NED values are the same both applying and not fold-in and fold-out.

Summarizing, the data analysis revealed that the use of fold-in and fold-out slightly affect the authoritativeness and the stability of the clustering results. This is probably due to the fact that the number of software entities added and removed to the first version to get the subsequent versions is lower with respect to the total number of software entities to be clustered. As far as the non-extremity of cluster distribution is concerned, the results are the same both using and not fold-in and fold-out.

6.4. RQ3: Is there a computational difference when our approach uses or not fold-in and fold-out?

Figure 5 shows the computational time (left hand side) and the efficiency (right hand side) of the approach both using and not using fold-in and fold-out. In this section we are interested in analyzing the computation time, while the assessment of the efficiency is presented in Sect. 6.5.

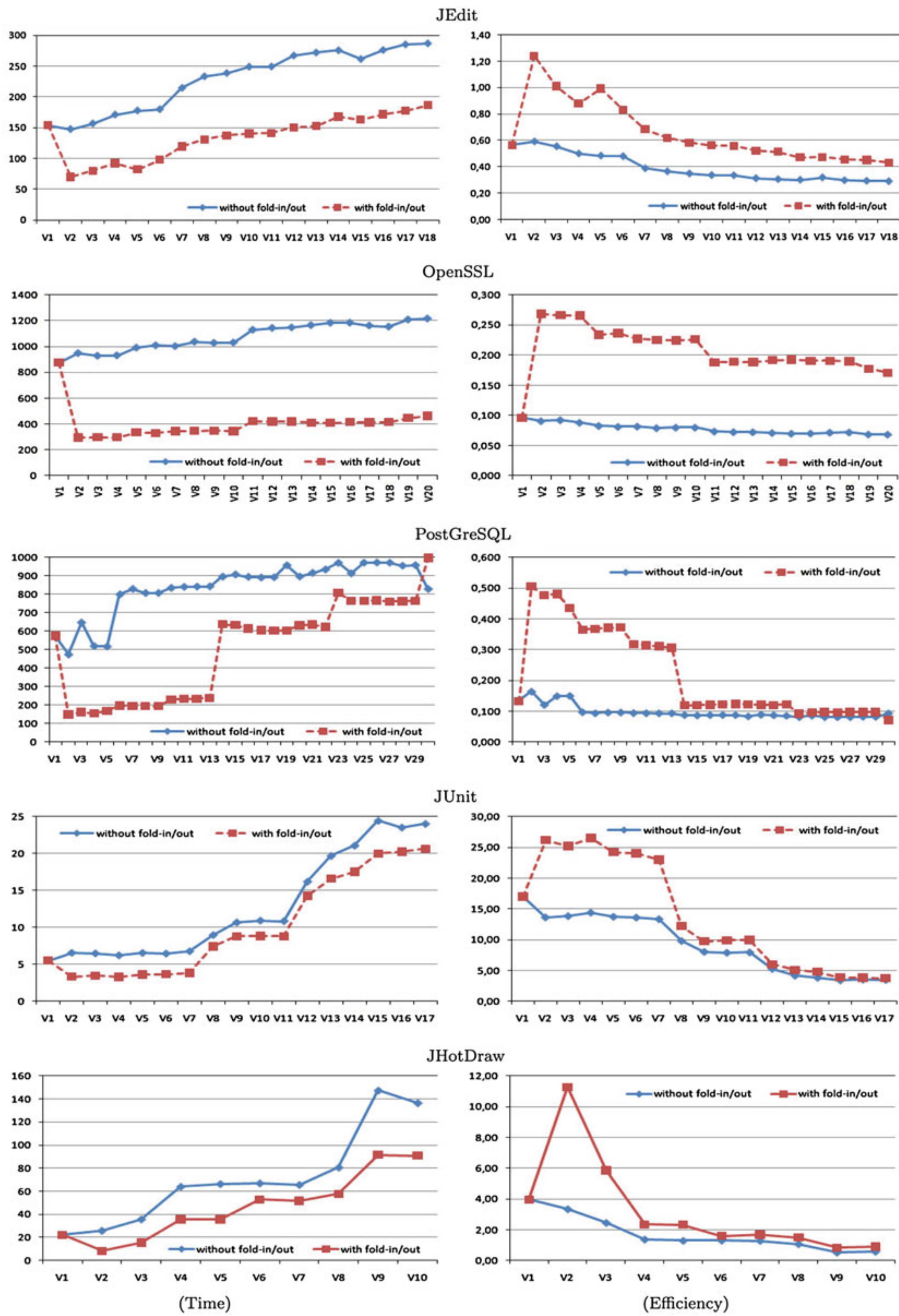

Fig. 5. Comparison of the results using or not the fold-in and fold-out mechanisms

The application of fold-in and fold-out mechanisms on all the version of the selected software systems always produced better performance results. In fact, for all the versions the time needed to partition the software entities is inferior when using fold-in and fold-out, while the time is the same on the first version of each system. The latter point is because the indexing of all the entities was needed on each first version of the selected software systems (see Sects. 3 and 5.4). A further consideration is due on the version V30 of PostGreSQL. In fact, as shown in Fig. 5 and in Table 5 the computation time is greater applying fold-in and fold-out (i.e., 996.6 with respect to 828.72). A further analysis on the versions V1 and V30 of PostGreSQL revealed that these versions are very different in terms of software entities, namely a large number of entities in the version V1 are not present in the version V30. In particular, 285 new software entities that are not present in V1 are added in V30, while 175 entities that are only present in V1 are then removed to get V30. This means that about 47% of the entities are added and removed from V1 to get V30 (460 entities out of 971). This represents the case in which the use of fold-in and fold-out could not be justified.

The data analysis therefore revealed that the performances of our approach are much better in case fold-in and fold-out are used provided that a suitable difference in terms of software entities can be observed between the analyzed version and the version on which the indexing has been performed the first time.

6.5. RQ4: Is our proposal more efficient when fold-in and fold-out mechanisms are employed?

The right hand side of Fig. 5 shows the efficiency results we achieved on all the versions of the selected system using or not fold-in and fold-out. We can observe that the approach is more efficient in case fold-in and fold-out are used, thus positive answering the research question RQ4. However, an interesting trend can be noticed: the more the Efficiency value is, the closer is the analyzed version to the version on which the indexing has been performed the first time. We can also note that on average the efficiency of the approach using fold-in and fold-out decrease after a given version. We can then conclude that the software engineer has to execute the re-indexing (see Sect. 3) of all the software entities when the analyzed version is far from the version on which the indexing has been performed the last time (i.e., the first version in our experimentation). The data analysis also shows that the Efficiency values are almost constant in case fold-in and fold-out are not used. This is always true except for JUnit and JHotDraw, where the Efficiency values decrease through the subsequent versions. A plausible motivation is that the system becomes larger and larger when passing through the versions, thus requiring a greater computational time.

6.6. Threats to validity

In our case the reliability of the used measures (i.e., Authoritativeness, Stability, NED, and Efficiency) may threaten the validity of the results. In particular, the Authoritativeness values as well as the efficiency could be affected by the used authoritative partition that has been identified by using the original decomposition of the classes/programs performed by the developers. Even, the lower and upper limits to compute the NED measure represent a further threat for the achieved results. Future work will be devoted to consider different measures (e.g., coupling and cohesion) [AFL99] to deeply investigate the effectiveness of the proposed approach. Objective criteria could be also employed to further and deeply evaluate the results in terms of correctness and completeness of each cluster within the partition identified by the tool prototype. To this end, measures from the information retrieval theory may be adopted, namely precision, recall, and F-measure. The main issue related to the computation of these measures is that software engineers with a suitable experience on the analyzed system are needed to get the authoritative partition. On an open source software system, this becomes impossible in practice since it is very difficult to find a developer to involve in studies like the one presented here. This issue could be possibly overcome conducting research industrial projects, where the software systems produced and marketed by the industrial partners could be analyzed to confirm or contradict the experimental results presented in the paper. This future direction for our work may also reduce the effect of evaluating the approach on open source software systems and could also allow reducing biases regarding the size of the analyzed software systems.

Regarding the stability assessment, the used measure may threaten the results as it assumes that a software entity appears in both the consecutive versions of a studied software system. This means that the first time a new class is added to a given version it does not contribute to the computation of the stability value. Future work will be devoted to adopt different approaches (e.g., [TH00]) to further investigate the stability. We did not use them because we were interested in this paper to assess whether our proposal outperforms other clustering based

approaches. Therefore, we did not have any alternative since the empirical assessments of these approaches were based on the measure used here.

7. Conclusion and future work

Clustering plays an important role in program understanding and software reengineering of traditional and web based systems [AFL99, DLST07, DLRST09, KDG07, RPTG08]. In this paper we have presented an Eclipse based environment implementing a novel and automatic clustering based approach to identify partitions of software systems into meaningful subsystems. The approach first analyzes software entities and then computes the dissimilarity between them using an LSI based measure. Successively, a partition of the software entities is identified using the k-means clustering algorithm. To identify a good partition of software entities k-means is iteratively performed.

Three are the meaningful characteristics of our approach: (i) it can be automatically used to recovery the architectural view of a given software system; (ii) it can be applied on software systems implemented in any programming languages; (iii) the use of fold-in and fold-out both to increase the computational efficiency and to reduce computational time. The latter point is very relevant since today the academy and the industry are manifesting an in increasing interest in the design and development of low consumption hardware and software systems (see for example [Gra09, RSR+08]). Such an interest is mainly due the International (e.g., the Kyoto treaty) and European directives and to the greater and greater feeling of the population for an ecosustainable society.

The validity of both the approach and the plug-in has been assessed on open source software systems with respect to: authoritativeness, stability, non-extremity of cluster distribution, computation time, and efficiency. According to these criteria we have empirically observed that our proposal (using both fold-in and fold-out) generally produced good results. However, we also observed that for each version of the considered systems there is a difference between the number of automatically identified clusters and the number of groups within the authoritative partition. In particular, for the Java systems this difference is in favor of the automatically detected clusters, namely the prototype identifies more clusters than the actual ones. Conversely, on the C/C++ systems the system prototype has identified a number of cluster less than the actual ones. This could be due to the fact that Java differently from C/C++ promotes the use of packages. To confirm or contradict such a result a further and special designed investigation is needed.

Another contribution of this paper concerns the comparison between our proposal and the ones presented in [BG09, CDMS10, CDMMS11, WH05]. Although we were not able to conduct a punctual comparison, we observed that our approach represents a good trade-off in case we need to optimize all the considered criteria. The lack of a more punctual comparison is due both to the absence of a public dataset and to the fact that considered research papers partially present the achieved experimental results (e.g., authors do not specify the analyzed versions and only the interval values of the considered measures are shown). A further analysis of the experimental data indicated that the use of the fold-in and fold-out mechanisms improves the efficiency and reduces the computational time needed to detect clusters.

Future work will be devoted to extend the approach in order to support the identification of topics in the recovered subsystems. A further direction for future work consists to adapt the approach to manage software entities at differ granularity level (i.e., at the method level rather than at the class level). We also plan to extend the approach to enable the management of software entities that are updated from different subsequent versions of a software system. Finally, we also plan to extend the software prototype to handle software systems implemented in programming languages different from C/C++ and Java.

References

[AFL99] Anquetil N, Fourrier C, Lethbridge TC (1999) Experiments with clustering as a software remodularization method. In: In Proceedings of the 6th working conference on reverse engineering. IEEE Computer Society, Washington, pp 235–255

[BD09] Bruegge B, Dutoit AH (2009) Object-oriented software engineering using UML, patterns, and Java, 3rd edn. Prentice Hall Press, New Jersy

[BG09] Bittencourt RA, Guerrero DDS (2009) Comparison of graph clustering algorithms for recovering software architecture module views. In: Proceedings of the European conference on software maintenance and reengineering, Washington. IEEE Computer Society, pp 251–254

[BHB99] Bowman IT, Holt RC, Brewster NV (1999) Linux as a case study: its extracted software architecture. In: Proceedings of the 21st international conference on Software engineering, ICSE '99. ACM, New York, pp 555–563
[CDMMS11] Corazza A, Di Martino S, Maggio V, Scanniello G (2011) Investigating the use of lexical information for software system clustering. In: Proceedings of the 15th European conference on software maintenance and reengineering, CSMR '11, Washington. IEEE Computer Society, pp 35–44
[CDMS10] Corazza A, Di Martino S, Scanniello G (2010) A probabilistic based approach towards software system clustering. In: Proceedings of the European conference on software maintenance and reengineering, pp 88–96
[DDL+90] Deerwester SC, Dumais ST, Landauer TK, Furnas GW, Harshman RA (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407
[DGLD05] Ducasse S, Gîrba T, Lanza M, Demeyer S (2005) Moose: a collaborative and extensible reengineering environment, pp 55–71. RCOST / Software Technology Series. Franco Angeli
[DLRST09] De Lucia A, Risi M, Scanniello G, Tortora G (2009) An investigation of clustering algorithms in the comprehension of legacy web applications. J Web Eng 8(4):346–370
[DLST07] De Lucia A, Scanniello G, Tortora G (2007) Identifying similar pages in web applications using a competitive clustering algorithm: Special issue articles. J Softw Maint Evol 19:281–296, September
[DMM99] Doval D, Mancoridis S, Mitchell BS (1999) Automatic clustering of software systems using a genetic algorithm. In: Proceedings of the software technology and engineering practice, pp 73–82, Washington. IEEE Computer Society
[EGK+01] Eick SG, Graves TL, Karr AF, Marron JS, Mockus A (2001) Does code decay? Assessing the evidence from change management data. IEEE Trans Softw Eng 27:1–12
[Gra09] Graefe G (2009) The five-minute rule 20 years later (and how ash memory changes the rules). Commun ACM 52:48–59
[Gut54] Guttman L (1954) Some necessary conditions for common-factor analysis. Psychometrika 19(2):149–161
[Har92] Harman D (1992) Ranking algorithms, pp 363–392. Prentice-Hall Inc., New Jersy
[IEE99] IEEE (1999) IEEE Standard for Software Maintenance, IEEE Std 1219–1998,vol 2. IEEE Press, Wiley
[JMF99] Jain AK, Murty MN, Flynn PJ (1999) Data clustering: a review. ACM Comput Surv 31:264–323
[Kai60] Kaiser HF (1960) The application of electronic computers to factor analysis. Educational Psychol Meas 20(1): 141–151
[KDG07] Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49:230–243
[Kle99] Kleinberg JM (1999) Authoritative sources in a hyperlinked environment. J ACM 46:604–632
[Kos00] Koschke R (2000) Atomic architectural component recovery for program understanding and evolution. Softwaretechnik-Trends
[Leh84] Lehman MM (1984) Program evolution. Inf Process Manag 19(1):19–36
[Mac67] MacQueen JB (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley symposium on mathematical statistics and probability, vol 1, pp 281–297, University of California Press
[MB07] Maqbool O, Babri H (2007) Hierarchical clustering for software architecture recovery. IEEE Trans Softw Eng 33(11):759–780
[MM01] Maletic JI, Marcus A (2001) Supporting program comprehension using semantic and structural information. In:Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, Washington. IEEE Computer Society, pp 103–112
[MM06] Mitchell BS, Mancoridis S (2006) On the automatic modularization of software systems using the bunch tool. IEEE Trans Softw Eng 32:193–208
[MOTU93] Müller HA, Orgun MA, Tilley SR, Uhl JS (1993) A reverse-engineering approach to subsystem structure identification. J Softw Maint Res Pract 5(4):181–204
[MRS08] Manning CD, Raghavan P, Schtze H (2008) Introduction to information retrieval. Cambridge University Press, New York
[Pig97] Pigoski TM (1997) Practical software maintenance: best practices for managing your software investment. Wiley, New York
[RPTG08] Ricca F, Pianta E, Tonella P, Girardi C (2008) Improving web site understanding with keyword-based clustering. J Softw Maint Evol 20:1–29
[RSR+08] Rofouei M, Stathopoulos T, Ryffel S, Kaiser W, Sarrafzadeh M (2008) Energy-aware high performance computing with graphic processing units. In: Proceedings of the 2008 Conference on Power aware computing and systems, pp 11–11, Berkeley, USA, USENIX Association
[SDDD10a] Scanniello G, D'Amico A, D'Amico C, D'Amico T (2010) Architectural layer recovery for software system understanding and evolution. Softw Pract Experience 40:897–916
[SDDD10b] Scanniello G, D'Amico A, D'Amico C, D'Amico T (2010) Using the kleinberg algorithm and vector space model for software system clustering. In: Proceedings of the IEEE 18th International Conference on Program Comprehension, ICPC'10, IEEE Computer Society, Washington, pp 180–189
[SRT10] Scanniello G, Risi M, Tortora G (2010) Architecture recovery using latent semantic indexing and k-means: An empirical evaluation. In: Proceedings of 8th IEEE International Conference on Software Engineering and Formal Methods, pp 103–112
[TH99] Tzerpos V, Holt RC (1999) Mojo: A distance metric for software clustering. In: Proceedings of the 6th Working Conference on Reverse Engineering, pp 187–193
[TH00] Tzerpos V, Holt RC (2000) On the stability of software clustering algorithms. In: Proceedings of the 8th International Workshop on Program Comprehension, pp 211–218
[Ton01] Tonella P (2001) Concept analysis for module restructuring. IEEE Trans Softw Eng 27(4):351–363
[TP04] Tonella P, Potrich A (2004) Reverse engineering of object oriented code. Monographs in Computer Science Series. Springer Science
[vDHK+04] van Deursen A, Hofmeister C, Koschke R, Moonen L, Riva C (2004) Symphony: View-driven software architecture reconstruction. In: Proceedings of the 4th Working IEEE/IFIP conference on software architecture, IEEE Computer Society, Washington, pp 122–132

[WH05] Wu AE, Hassan J, Holt RC (2005) Comparison of clustering algotithms in the context of software evolution. In: Proceedings of the 21st IEEE international conference on software maintenance, IEEE Computer Society, pp 525–535
[Wig97] Wiggerts TA (1997) Using clustering algorithms in legacy systems remodularization. In: Proceedings of the fourth working conference on reverse engineering (WCRE '97). IEEE Computer Society, Washington, pp 33–43
[Wil07] Wild F (2007) An optimal algorithm for mojo distance. In: Proceedings of international conference on latent semantic analysis in technology enhanced learning, pp 11–12
[WT03] Wen Z, Tzerpos V (2003) An optimal algorithm for mojo distance. In: Proceedings of the 11th IEEE international workshop on program comprehension, pp 227–235
[ZSG79] Zelkowitz MV, Shaw AC, Gannon JD (1979) Principles of software engineering and design. Prentice Hall Professional Technical Reference

Received 14 January 2011
Revised 15 April 2011
Accepted 24 June 2011 by José Fiadero, Stefania Gnesi and Tom Maibaum
Published online 25 August 2011