Turning Large Software Component Repositories into Small Index Files

Marcos Paulo Paixão, Leila Silva
Computing Department
Federal University of Sergipe
Aracaju, Brazil
marcospsp@dcomp.ufs.br, leila@ufs.br

Talles Brito, Gledson Elias
Informatics Department
Federal University of Paraíba
João Pessoa, Brazil
talles@compose.ufpb.br, gledson@di.ufpb.br

Abstract—**Software component repositories have adopted semi-structured data models for representing syntactic and semantic features of handled assets. Such models imply key challenges to search engines, which are related to the design of indexing techniques that ought to be efficient in terms of storage space requirements. In such a context, by applying clustering techniques before indexing component repositories, this paper proposes an approach for reducing the number of assets in the repository, and consequently, the size of index files. Based on an illustrative repository, outcomes indicate a significant optimization in the number of assets to be indexed, and, as a consequence, produces significant gains in storage requirements. Besides, it has been assessed in terms of two different clustering evaluation methods, evincing that the proposed approach can be considered a good clustering algorithm because produces compact and well-separated clusters.**

Keywords - Component repositories; clustering techniques; indexing.

I. Introduction

By enabling different software developers to share software assets, software component repositories have the potential to improve software reuse level. However, reuse of software assets is in general a hard task, particularly when search and selection must be conducted over large-scale asset collections. Therefore, in repository systems, it is important the development of search engines that can help searching, selecting and retrieving required software assets.

According to Orso *et al.* [1], the aim of a repository system is not to store software assets only, but also metadata describing them. Such metadata provides information employed by search engines for indexing stored assets. In such a direction, as endorsed by Vitharana [2], component description models can adopt high level concepts for describing component metadata, making possible to express syntactic and semantic features, and so, facilitating developers to search, select and retrieve assets. In practice, currently available component description models have adopted approaches based on semi-structured data, more specifically XML, allowing structural relationships among elements to aggregate semantic to textual values. As examples, it can be mentioned RAS [3] and X-ARM [4].

However, indexing techniques based on textual restrictions are not efficient for semi-structured data. Such techniques are unable of indexing structural relationships among terms, compromising query precision with false-positives. Thus, the adoption of semi-structured data implies challenges related to the design of indexing techniques that ought to be efficient in terms of storage space requirements, processing time and precision level of queries, which can be constrained by textual and structural restrictions.

Several proposals can be found in the literature for dealing with such problems. Despite their relevant contributions, existing techniques do not meet storage space and query processing time requirements [5], and also query precision level [6]. In such a scenario, the proposal presented by Brito *et al.* [7] represents a noticeable indexing technique based on semi-structured data, which can be considered precise and efficient in terms of query processing time, but suffers from problems related to storage space requirements. Such problems occur because generated index files are bigger than the input database. Thus, in the context of large-scale software component repositories, it is still a challenging open issue to design indexing techniques that minimize the storage space requirements without excessively impacting on query processing time and precision.

In such a context, based on the adoption of clustering techniques, this paper proposes an approach for reducing the number of assets in the repository, and consequently, optimizing the storage space requirements. It is an extended and improved version of [8]. The clustering heuristic proposed is based on the classical hierarchical algorithm and *K*-means [9]. Taking into account a large-scale component repository, the proposed approach identifies clusters (groups) of similar software assets and generates new representative assets, which in turn must be handled by the indexing technique supported by the search engine of the repository. Each representative asset has a simplified description, also based on semi-structured data, which makes reference to all original assets that belong to its cluster of similar assets. In order to do that, the paper also proposes a similarity metric that has the aim of indicating the set of assets that belongs to the same cluster. The bigger the similarity among assets in the repository, the lesser is the number of identified clusters, and as a result, the lesser is the number of representative assets that must be indexed by the search engine, enabling to save storage space. In order to validate the proposed approach, a random database composed of 14.000 assets has been generated and results indicate that there is a significant optimization in terms of the number of assets to be indexed.

The remainder of this paper is structured as follows. Section II describes related techniques, evincing the original initiative of applying clustering techniques in the context of indexing software component repositories. The adopted component description model, called X-ARM, is briefly presented in Section III, identifying the main types of assets and their relationships. Then, Section IV presents the proposed clustering approach for reducing the number of assets to be indexed, and so, optimizing storage space requirements. After that, some outcomes observed in a preliminary evaluation performance are presented in Section V. In conclusion, Section VI presents some final remarks and delineates future work.

II. Related Techniques

Taking into account that the problem of data clustering is NP-hard, several heuristics have already been proposed. Xu and Wunsch [10] present an interesting review of the research field. In [11], Feng shows that clustering algorithms, in particular, hierarchical algorithms and *K*-means [9], are equivalent to optimization algorithms of a fitness function.

Clustering techniques have been used in several software engineering domains. For example, Mancoridis *et al.* [12] applied clustering in the domain of software maintenance, by introducing the concept of software modularization as a clustering problem for which search is applicable. A tool called Bunch [13] is proposed allowing the application of several clustering heuristics to perform search based software modularization. Chiricota *et al.* [14] investigates the application of clustering techniques in the domain of reverse engineering, in particular, adopting such techniques to recover the structure of software systems. Wu *et al.* [15] compares several clustering approaches proposed in the context of software evolution. In [16], Li *et al.* proposes the adoption of clustering techniques for encapsulating software requirements. Cohen *et al.* [17] showed how search based clustering algorithms could be applied to improve garbage collection in Java programs.

Although clustering techniques are applied in several problems of software engineering, for the best knowledge of the authors, these techniques have never been adopted in the context of indexing software components repositories. Therefore, it seems an original contribution to apply such techniques when indexing component repositories.

III. THE X-ARM MODEL

In order to express syntactic and semantic features of software components, Frakes [18] suggests the adoption of component description models, which provide a set of information that allows search systems to index and classify all types of related assets. In such a direction, this paper explores the X-ARM description model, which adopts a XML-based semi-structured data model, expressing not only syntactic information but also semantic properties [4]. Besides, X-ARM enables describing several types of software assets, which can be produced in component-based development processes, proving the required semantic for representing their relationships.

As illustrated in Fig. 1, X-ARM allows describing component and interface specifications, as well as component implementations. The component and interface specifications can be described in a way that is independent or dependent of component model. On the one hand, independent specifications do not take into account any feature or property of component models, such as CCM, JavaBeans, EJB and Web Services. On the other hand, dependent specifications ought to consider features and properties related to the adopted component models.

In X-ARM, both dependent and independent interface specifications are described as a set of operations. Each operation has a name, a set of input or output parameters and a return value. In component-based development processes, dependent interface specifications must be in conformance with their independent counterparts. So, in Fig. 1, it can be observed that dependent interface specifications must reference to their respective independent interface specifications.

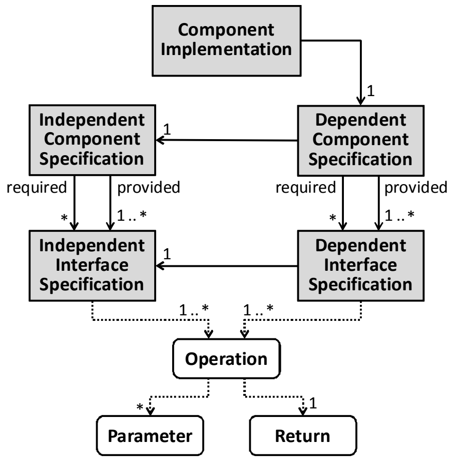

Component Implementation
1
Independent Component Specification
1
Dependent Component Specification
required
provided
*
1..*
required
provided
*
1..*
Independent Interface Specification
1
Dependent Interface Specification
1..*
1..*
Operation
*
1
Parameter
Return

Figure 1. Relationships between artifacts.

Dependent and independent component specifications can make reference to a set of provided and required interface specifications. However, it must be noticed that independent component specifications can refer to independent interface specifications only. Similarly, dependent component specifications can refer to dependent interface specifications only. In component-based development processes, dependent component specifications must be in conformance with their respective independent counterparts. Therefore, note that dependent component specifications must make reference to their respective independent component specifications.

In summary, dependent interface and component specifications must be in conformance with their respective independent specifications. Besides, for each independent specification, several dependent specifications can be described, each one in conformance with a given software component model.

In a similar way, in component-based development processes, component implementations must be in conformance with their respective dependent component specifications. So, in Fig. 1, note that component implementations must refer to their correspondent dependent component specifications. Besides, for each dependent component specification, several component implementations can be realized.

As an example of the description of an asset in X-ARM, Fig. 2 illustrates a fragment of a dependent component specification. In Fig. 2, all lines are numbered and many details have been suppressed for didactic purposes. Line 1 represents the asset header, in which can be found the asset identifier (id). Lines 2 to 4 make reference to the independent component specification, from which the described asset must be in conformance with. Then, lines 5 to 14 refer to all dependent interface specifications, which are provided by the described dependent component specification. Although note illustrated in Fig. 2, required interfaces can also be specified in a similar way.

```
01 <asset name="dependentCompSpec-X"
         id="compose.dependentCompSpec-X-1.0-beta">
02   <model-dependency>
03     <related-asset name="independentCompSpec-Z"
                      id="compose.independentCompSpec-Z-1.0-stable"
                      relationship-type="independentComponentSpec"/>
04   </model-dependency>
05   <component-specification>
06     <interface>
07       <provided>
08         <related-asset name="dependentInterface-A"
                          id="compose.dependentIntSpec-A-2.0-stable"
                          relationship-type="dependentInterfaceSpec"/>
09       </provided>
10       <provided>
11         <related-asset name="dependentInterface-B"
                          id="compose.dependentIntSpec-B-3.0-stable"
                          relationship-type="dependentInterfaceSpec"/>
12       </provided>
13     </interface>
14   </component-specification>
15 </asset>
```

Figure 2. Component specification in X-ARM.

IV. A Clustering Based Indexing Approach

As largely recognized in the literature, the task of indexing repositories based on semi-structured data is a relevant issue [5][6][7]. One of the major challenges is to provide an indexing mechanism that reduces storage space requirements, but without excessively impacting on query processing time and precision level.

In such a context, this paper proposes a solution for optimizing the storage space required by index files. To do that, the proposed approach constructs a clustered repository, which is composed of representative assets of the set of software assets stored in the original repository. Therefore, instead of indexing the original repository, the adopted search service ought to index the reduced set of representative assets, which make reference to the original assets. In order to identify the groups of similar assets, and, consequently, to construct the representative assets that compose each group, the paper also proposes the adoption of data clustering techniques.

Clustering techniques [9] consist of three basic phases: (i) extraction of features that express the behavior of the elements to be clustered; (ii) definition of the similarity metric in order to compare evaluated elements; and (iii) adoption of a clustering algorithm. The phase of extracting features consists in defining what information is relevant to express the evaluated element and how information is quantified. Such information defines an attribute vector and thus an element can be represented as a point in the multidimensional space. The similarity metric expresses in quantitative terms the similarity between elements. In general, a function is defined for such a purpose, in which the Euclidean distance [9] between two points (elements) is one of the more common adopted metrics. Finally, the data clustering algorithm is a heuristic that has the aim of generating groups of elements, in which each group is composed of similar elements, according to the adopted similarity metric.

A. *Relevant Features*

The approach proposed herein applies the clustering technique taking into account the five types of assets that can be stored in the repository, that is: dependent and independent component specifications, dependent and independent interface specifications and component implementations. The clustering technique is applied separately for each type of asset. Therefore, each type has a distinct attribute vector for representing its features.

The relevant features of an independent interface specification are its defined operations, considering their names, input and output parameters and return values. Consequently, different independent interface specifications are considered similar when they have in common a considerable subset of defined operations.

Taking into account dependent interface specifications, the relevant features are the referenced independent interface specification together with their operations. Thus, different dependent interface specifications are considered similar when they refer to the same independent interface specification or have in common a considerable subset of defined operations.

In relation to independent component specifications, for each one, the relevant feature is the set of provided independent interface specifications. So, different independent component specifications are considered similar when they have in common a considerable subset of provided independent interface specifications.

For a dependent component specification, the relevant features are its referenced independent component specification, as well as its set of provided dependent interface specifications. Therefore, different dependent component specifications are considered similar when they refer to the same independent component specification or have in common a subset of provided dependent interfaces.

Finally, for a component implementation, the relevant feature is its referenced dependent component specification. Hence, different implementations of the same dependent component specification are considered similar.

As an example, Table I presents the attribute vector of the asset illustrated before in Fig. 2. As can be noticed, the asset is a dependent component specification. Therefore, the attribute vector is composed of its referenced independent component specification (lines 2 to 4) and its set of provided dependent interface specifications (lines 5 to 14).

TABLE I. ATTRIBUTE VECTOR OF THE ASSET X.

ID	compose.dependentCompSpec-X-1.0-beta
Independent Component Specification	compose.independentCompSpec-Z-1.0-stable
Dependent Interface Specification	compose.dependentIntSpec.A-2.0-stable compose.dependentIntSpec.B-3.0-stable

B. Similarity Metric

The similarity metric is defined based on the asset attribute vector. Since the attribute vector differs between distinct types of assets, the similarity metric is also different for each type of asset. In this approach the similarity between two assets is quantified by an integer number, called *distance*. To avoid negative distances, we defined that the initial default distance (d_i) between two assets is 300. The similarity criterion is applied and this value may decrease, in such a way that assets are considered more similar when the final distance (d_f) between them approximates to zero.

For two dependent component specifications *a* and *b*, the similarity is defined by (1), where *k(a,b) = 0* if both assets refer to distinct independent component specifications; otherwise *k(a,b) = 200*. Let *I* be the number of dependent interface specifications provided by both assets and *U* be the set of dependent interface specifications provided by at least one of them. The term *p(a,b)* is defined as *p(a,b) = I/U*. As can be noticed, when *p(a,b)* is 1 both assets provide the same set of dependent interface specifications, and thus they are more similar.

$$d_f(a,b) = d_i - k(a,b) - p(a,b) \times 100 \tag{1}$$

In the case of two independent component specifications *a* and *b*, the similarity is given by (2), where *p(a,b)* is calculated as explained before for dependent component specifications, but considering the number of independent interface specifications provided by both assets. So, let *I* be the number of independent interface specifications provided by both assets and *U* be the set of independent interface specifications provided by at least one of them. The term *p(a,b)* is defined as *p(a,b) = I/U*. Similarly, when *p(a,b)* is 1 both assets provide the same set of independent interface specifications and thus, they are more similar.

$$d_f(a,b) = d_i - p(a,b) \times 300 \tag{2}$$

Analogously, for two dependent interface specifications *a* and *b*, the similarity is calculated as expressed in (3), where *l(a,b) = 0* if both assets refer to distinct independent interface specifications; otherwise, *l(a,b) = 200.* The term *op(a,b)* is the ratio of common operations of both assets in relation to the union of operations of these assets. Two operations are considered similar if they have the same name, the same return type and a percentage of coincidence in parameters; the value of the percentage is defined by the user.

$$d_f(a,b) = d_i - l(a,b) - op(a,b) \times 100 \tag{3}$$

Taking into account two independent interface specifications *a* and *b*, the similarity is calculated by (4), where *op(a,b)* represents the percentage of common operations provided by both interfaces, exactly as explained before for dependent interface specifications.

$$d_f(a,b) = d_i - op(a,b) \times 300 \tag{4}$$

Finally, for two component implementations *a* and *b*, the similarity is given by (5), where *q(a,b) = 0* if both assets refer to distinct dependent component specifications; otherwise *q(a,b) = 300.* As can be noticed, when *q(a,b)* is 300 both assets implement the same dependent component specification, and thus they are similar.

$$d_f(a,b) = d_i - q(a,b) \tag{5}$$

As an example, consider two dependent component specifications *C* and *D*, whose attribute vectors are given in Table II and Table III, respectively. As these assets refer to distinct independent component specifications, according to (1), *k(C,D) = 0*. In this example, *C* and *D* have a common interface and together provide three different interfaces. Thus, *I = 1*, *U = 3* and *p(C,D) = 1/3*. Hence, $d_f\,(C,D) = d_i - k(C,D) - p(C,D)*100 = 300 - 0 - 0.33*100 = 276{,}67$.

TABLE II. ATTRIBUTE VECTOR OF THE ASSET C

ID	compose. depCompSpec-C-2.0-beta
Independent Component Specification	compose.indepCompSpec-A-3.0-stable
Dependent Interface Specification	compose.depIntSpec-A-4.0-mature compose.depIntSpec-C-4.0-mature

TABLE III. ATTRIBUTE VECTOR OF THE ASSET D

ID	compose. depCompSpec-D-3.0-mature
Independent Component Specification	compose.indepCompSpec-C-3.0-stable
Dependent Interface Specification	compose.depIntSpec-B-2.0-beta compose.depIntSpec-C-4.0-mature

C. *Clustering Algorithm*

The proposed clustering algorithm has two stages. In the first stage, initially, assets are randomly chosen from the respective storage unit and stored in the primary memory. It is suggested to exhaust the memory capacity with this operation. Next, but still in the first stage, the classical hierarchical clustering algorithm [9] is applied to these assets. In the beginning of the algorithm each asset is considered a cluster. Then, the algorithm groups successively the two nearest clusters, until the distance between clusters is greater than an established threshold, specified by the user. The algorithm considers the similarity metric described previously to compute the distance. The combined cluster is considered a representative asset of the joined clusters. For each type of asset, the representative asset includes the relevant features for the similarity metric and also references to the joined assets. At the end of the iteration, a directory containing all formed representative assets (clusters) is stored in secondary memory.

Fig. 3 illustrates the main steps of the first stage: (a) assets are randomly selected from the repository; (b) clusters composed of similar assets are constructed by applying the hierarchical clustering algorithm; and (c) representative assets are created for representing each cluster.

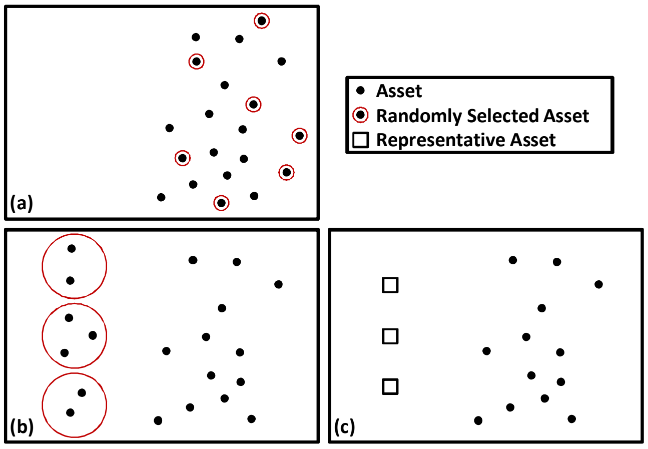


Figure 3. The first stage.

In the second stage, a *K*-means based algorithm [9] is adopted. In general terms, representative elements are considered centroids. However, differently from *K*-means, such centroids are not recalculated in the proposed approach. Indeed, each asset, not yet clustered in the first stage, is compared with each representative asset. The asset is candidate to be included in a cluster when the distance between the asset and the respective representative asset is lesser than the threshold. Fig. 4 shows the second stage.

As depicted in Figs. 4a, 4b, and 4c, considering all candidate clusters, the asset is included in the cluster that has the minor distance and then the representative element of the cluster is reconstructed considering the features of the included asset. Otherwise, as shown in Figs. 4d, 4e and 4f, if the asset is not a candidate to any cluster, the own asset becomes a new representative element and so a new cluster.

To conclude the description of the approach, it remains to explain how the relevant features of representative assets are determined. A representative asset, resulted from the combination of two clusters composed by dependent component specifications, includes all provided dependent interface specifications of the joined assets and the independent component specification they refer. This specification is the one that mostly occurs in the assets that form the combined cluster; in the case of a draw one specification is chosen arbitrarily.

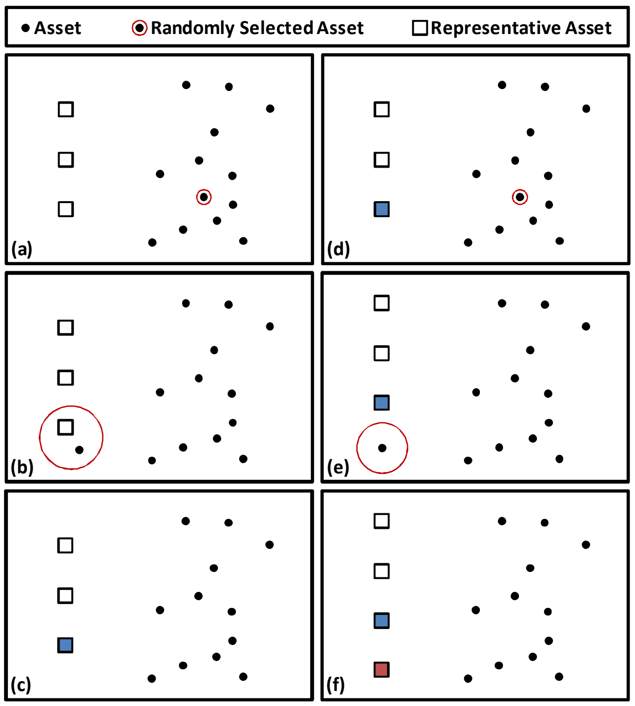


Figure 4. The second stage.

For a representative asset resulted from the combination of two clusters composed by independent component specifications, the relevant features are all provided independent interface specifications of the joined assets.

A representative asset resulted from the combination of two clusters composed by dependent interface specifications include as relevant features all operations of the joined assets, as well as the independent interface that the representative asset implements. This interface is the one mostly referred by the joined clusters.

Taking into account a representative asset resulted from the combination of two clusters composed by independent interface specification assets, the relevant features are all provided operations of the joined assets.

Finally, for a representative asset resulted from the combination of two clusters of component implementations, the relevant feature is its referenced dependent component specification. This specification is the one mostly frequent in the joined assets.

As an example of the construction of representative assets, considers two original assets as shown in Fig. 5 and Fig. 6. The representative asset resulted from the combination of these assets is described in Fig. 7. As both assets are dependent component specifications, observe that the representative asset includes all provided dependent interface specification (lines 6 to 16 of Fig. 7) and the independent component specification that occurs more frequently in the original assets. (line 3 of Fig. 7).

```
01 <asset name="depCompSpec-K"
           id="compose.depCompSpec-K-1.0-alfa">
02    <model-dependency>
03       <related-asset name="indepCompSpec-O"
                        id="compose.indepCompSpec-R-6.0-beta"
                        relationship-type="independentComponent"/>
04    </model-dependency>
05    <component-specification>
06       <interface>
07          <provided>
08             <related-asset name="depIntSpec-R"
                              id="compose.depIntSpec-R-3.0-mature"
                              relationship-type="dependentInterface"/>
09          </provided>
10          <provided>
11             <related-asset name="depIntSpec-S"
                              id="compose.depIntSpec-S-7.0-alfa"
                              relationship-type="dependentInterface"/>
12          </provided>
13       </interface>
14    </component-specification>
15 </asset>
```

Figure 5. Dependent component specification K.

```
01 <asset name="depCompSpec-L"
           id="compose.depCompSpec-L-2.0-pre-alfa">
02    <model-dependency>
03       <related-asset name="indepCompSpec-O"
                        id="compose.indepCompSpec-R-6.0-beta"
                        relationship-type="independentComponent"/>
04    </model-dependency>
05    <component-specification>
06       <interface>
07          <provided>
08             <related-asset name="depIntSpec-O"
                              id="compose.depIntSpec-O-1.0-alpha"
                              relationship-type="dependentInterface"/>
09          </provided>
10          <provided>
11             <related-asset name="depIntSpec-S"
                              id="compose.depIntSpec-S-7.0-alfa"
                              relationship-type="dependentInterface"/>
12          </provided>
13       </interface>
14    </component-specification>
15 </asset>
```

Figure 6. Dependent component specification L.

V. Results and Discussion

In order to evaluate the proposed distributed clustering approach, a set of experiments has been carried out. The purpose of such experiments is three-fold. First, it is intended to identify the gains in terms of the number of representative assets to be indexed when compared with the number of original assets. The second purpose is to discover the gain in terms of storage space requirements between the clustered repository and the original repository. Lastly, such experiments have evaluated the quality of the clustering approach using well-know metrics.

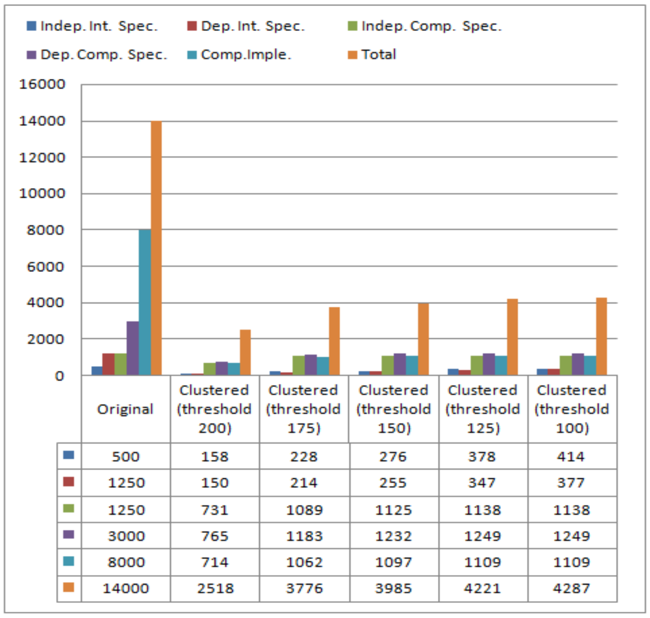
```
01 <asset name="repDepCompSpec-A1"
          id="compose.repDepCompSpec-A1">
02    <model-dependency>
03       <related-asset name="indepCompSpec-O"
                        id="compose.indepCompSpec-R-6.0-beta"
                        relationship-type="independentComponent"/>
04    </model-dependency>
05    <component-specification>
06       <interface>
07          <provided>
08             <related-asset name="depIntSpec-O"
                              id="compose.depIntSpec-O-1.0-alpha"
                              relationship-type="dependentInterface"/>
09          </provided>
10          <provided>
11             <related-asset name="depIntSpec-R"
                              id="compose.depIntSpec-R-3.0-mature"
                              relationship-type="dependentInterface"/>
12          </provided>
13          <provided>
14             <related-asset name="depIntSpec-S"
                              id="compose.depIntSpec-S-7.0-alfa"
                              relationship-type="dependentInterface"/>
15          </provided>
16       </interface>
17    </component-specification>
18 </asset>
```

Figure 7. Representative dependent component specification.

In order to perform the experiments, it has been developed a customizable script that automatically generates a repository that stores the mentioned X-ARM assets. The generated repository has 14.000 assets of different types. After creating the repository, the proposed approach has been applied for grouping the stored assets in clusters, generating their respective representative assets.

A. Gain in Number of Assets

Fig. 8 presents the number of each type of asset in the original repository and the clustered repositories after the application of the proposed approach using different thresholds, which vary from 100 to 200 in steps of 25. As can be noticed, the proposed approach significantly reduces the number of assets. As expected, the number of resulting representative assets decreases as the threshold increases. When the threshold is increased, two assets have more chance of being considered similar, and so, more chance of being grouped together. Thus, for example, when the threshold is increased from 100 to 200, the total number of original assets is reduced to 4,287 and 2,518 representative assets, respectively.



	Original	Clustered (threshold 200)	Clustered (threshold 175)	Clustered (threshold 150)	Clustered (threshold 125)	Clustered (threshold 100)
🟦	500	158	228	276	378	414
🟥	1250	150	214	255	347	377
🟩	1250	731	1089	1125	1138	1138
🟪	3000	765	1183	1232	1249	1249
🟦	8000	714	1062	1097	1109	1109
🟧	14000	2518	3776	3985	4221	4287

Figure 8. Number of assets.

For each considered threshold, the gain in number of assets has been identified and evaluated. Fig. 9 illustrates the gain in terms of the number of assets. For example, when the threshold is 150, the number of stored assets in the original repository is reduced around 28.5%, dropping from 14,000 original assets to 3,985 representative assets. As can be noticed in Fig. 9, the proposed approach performs a significant reduction in the number of stored assets, achieving relevant gains between 82% and 69.4%.

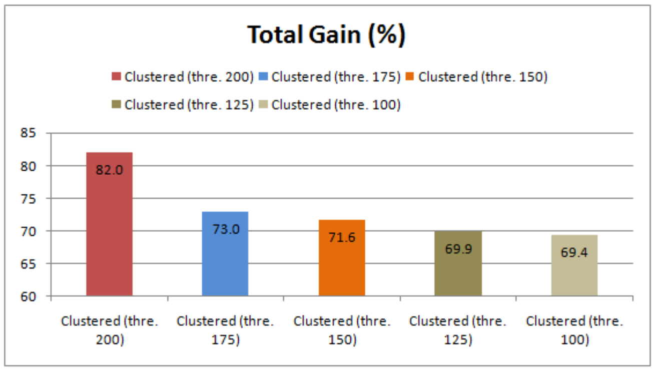


Figure 9. Total Gain in number of assets (%).

However, as shown in Fig. 10, the gains are different for each type of asset. Note that, in general, the better gains are achieved for component implementations and dependent interfaces. Considering component implementations, the gains become a little bit more expressive, varying between 91.1% and 86.1%. For dependent interface specifications, the gains are between 88% and 69.8%. In the former case, such higher gains can be explained by the considerable amount of assets of those types. As can be seen in Fig. 8, the original repository has 8,000 component implementations. Thus, this type of asset is the prevalent one in the evaluated repository, increasing the likelihood of identifying similar assets. Furthermore, considering that component implementations are considered similar when they refer to the same dependent component implementation, it is also possible to correlate such a good gain with the existence of different implementations of the same component specification, not only for different target platforms but also for meeting a variety of non-functional requirements, like performance, security and cost. Therefore, considering the various methods, techniques and algorithms that can be employed to meet non-functional requirements, it is obvious that such multiple implementations impact on the likelihood of identifying similar component implementations.

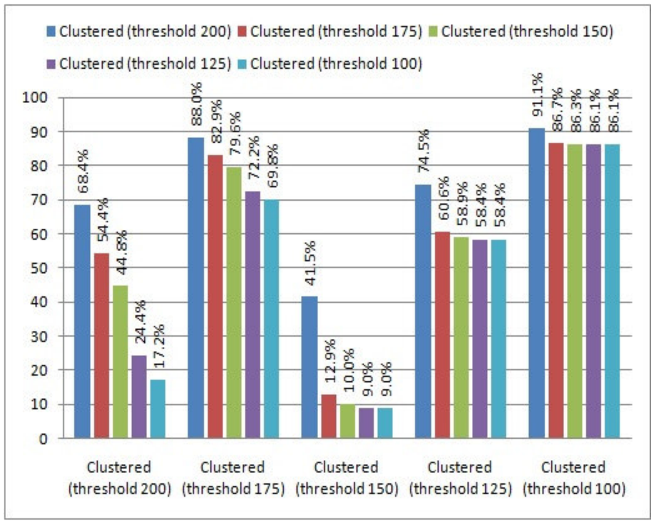


Figure 10. Gains in number of assets for different types of assets.

In the case of dependent interface specifications, the gains become better due mainly to two reasons. First, in software projects, it is not rare to implement different versions of software systems for different target platforms. So, in component-based software projects, different versions imply on several dependent interface specifications for each independent interface specification. Considering that dependent interface specifications are considered similar when they refer to the same independent interface specification, it is easy to see that multiple implementations impacts on the likelihood of identifying similar dependent interface specifications. The second reason is a consequence of the high gains in independent interface specifications. For instance, consider two dependent interface specifications ($depInt_i$ and $depInt_j$) that refer to two independent interface specifications ($indepInt_x$ and $indepInt_y$), respectively. Now, consider that $indepInt_x$ and $indepInt_y$ are clustered as the representative asset $indepInt_c$. As a consequence, now, both dependent interface specifications $depInt_i$ and $depInt_j$ refer to the same representative independent interface specification $indepInt_c$. Then, taking into account that dependent interface specifications are considered similar when they refer to the same independent interface specification, $depInt_i$ and $depInt_j$ are clustered and produce the representative asset $depInt_c$.

Clearly, a high gain in clustering independent interfaces has a significant impact in the gain in clustering dependent interfaces.

In terms of dependent component specifications, the gains range from 74.5% to 58.4%. One reason for this gain is the expressive number of assets in the repository (3,000 assets according Fig. 8). Furthermore, the existence of different versions of software systems for different target platforms implies on several dependent component specifications for each independent component specification. Considering that dependent component specifications are considered similar when they refer to the same independent component specification, it is clear to notice that multiple implementations impacts on the likelihood of identifying similar dependent component specifications.

In relation to independent component specifications, the gains are notably low, varying from 41.5% to 8.9%. Besides, as can be noticed in Fig. 10, the gain of 41.5% occurs for the higher threshold only. When the threshold is 175 and 125, the respective gains decrease to 12.9% and 8.9%. Such gains are relatively low and indeed not expected. As mentioned before, independent component specifications are considered similar when they have in common a considerable subset of provided independent interfaces. Thus, it is possible to infer that such low gains are a consequence of the difficulty of finding two or more independent component specifications that share a reasonable subset of independent interfaces.

As can be noticed, the clustering gains in independent interfaces specifications impact positively on the gains in dependent interface specifications, but give the impression that do not impact on the gains in independent component specifications. Furthermore, the clustering gains in independent component specifications impact on the gains in dependent component specifications, which in turn impact on the gains in component implementations.

B. *Gain in Storage Requirements*

As already mentioned, the adoption of semi-structured data for representing metadata about software components implies challenges related to the design of indexing techniques that ought to be efficient in terms of storage space requirements. Therefore, it is not enough to be efficient in reducing the number of assets, but also in downgrading storage space requirements for index files.

In such a direction, the gain in terms of storage space required by index files has been evaluated in the original repository, containing 14.000 X-ARM assets of different types. After generating the clustered repositories by applying the proposed approach for different thresholds, the original repository and the clustered repositories have been indexed using the indexing technique proposed in [7]. Fig. 11 presents the storage space required by the original repository and the clustered repositories, after applying the indexing technique.

As can be noticed, the proposed approach significantly reduces the required storage space. As expected, the required storage space decreases as the threshold increases. When the threshold increases, the number of representative assets reduces, and, as a consequence, the storage space required by index files also downgrades. Thus, when the threshold increases from 100 to 200, the storage space required by index files reduces from 10.9 to 7.3MB.

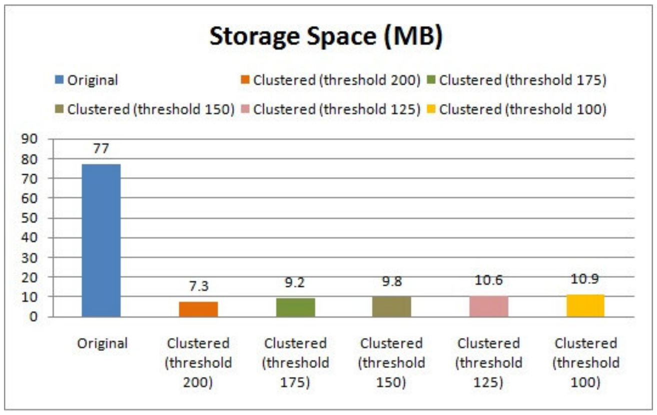


Figure 11. Storage space requirements for different thresholds.

For each considered threshold, the gain in storage space requirements has been identified and evaluated. Fig. 12 illustrates the gain in terms of storage space requirements. For example, when the threshold is 150, the storage space required by index files is reduced around 87.3%, dropping from 77 to 9.8 MB. As can be noticed in Fig. 12, the proposed approach performs a significant reduction in the storage space requirements, achieving relevant gains between 85.8% and 90.5%.

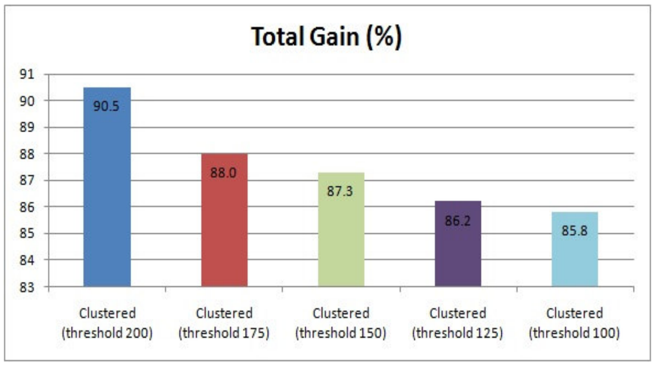


Figure 12. Total gain in storage requirements.

C. *Clustering Quality*

Of course, it is not enough to evaluate the gains in terms of number of assets and storage space requirements. It is also imperative to assess the quality of the clustering approach. In such a direction, the clustered repositories have been assessed in terms of two different clustering evaluation methods: *Davies-Bouldin index* and *Silhouette index*.

The Davies-Bouldin index [19] is a clustering evaluation method based on internal criterion. It is a function of the ratio of the sum of intra-cluster distances (within-cluster scatter) to inter-cluster distances (between-cluster separation), as defined in (6), where n is the number of clusters, c_i is the representative element of cluster i, σ_i is the average distance of all elements in cluster i to representative element c_i, and $d(c_i,c_j)$ is the distance between representative elements c_i and c_j. As widely mentioned in the literature, a good clustering algorithm must produce clusters with low intra-cluster distances (high intra-cluster similarity) and high inter-cluster distances (low inter-cluster similarity). Based on that, a good clustering algorithm has a small value of Davies-Bouldin index, representing compact and well-separated clusters [20].

$$DB = \frac{1}{n}\sum_{i=1}^{n} \max_{i \neq j}\left\{\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}\right\} \tag{6}$$

As can be seen in Fig. 13, the Davies-Bouldin index of the clustered repositories for all evaluated thresholds varies between 9.60 and 1.66. Such low values for the threshold from 100 to 150 evinces that the proposed approach can be considered a good clustering algorithm because has produced compact and well-separated clusters. Note that the Davies-Bouldin index increases as the threshold increases. Such a trend is already expected and indicates that lower thresholds produce higher intra-cluster similarity and lower inter-cluster similarity, and so higher quality clusters.

The Silhouette index [21] is based on the comparison of the tightness and separation of the clustered elements. The silhouette for each element is calculated as illustrated in (7), where a_i is the average intra-cluster dissimilarity of element i to all other elements within the same cluster, and b_i is the lowest average inter-cluster dissimilarity of element i to all other elements in another cluster. Note that the silhouette value varies between -1 and 1. Based on the silhouette for each element, the overall average silhouette for all elements can be easily calculated. If the overall average silhouette is close to 1, it means that elements are well-clustered and are assigned to very appropriate clusters. If the overall average silhouette is close to -1, it means that elements are misclassified and so poorly clustered.

$$S(i) = \frac{b_i - a_i}{max\{a_i, b_i\}} \tag{7}$$

As also illustrated in Fig. 13, the overall average silhouette of the clustered repositories for all evaluated thresholds varies between 0.62 and 0.84. Such values close to 1 are evidences that the proposed approach is a good clustering algorithm because elements are well-clustered and are assigned to appropriate clusters. Note that the silhouette index decreases as the threshold increases. Again, such a trend is already expected and indicates that lower thresholds produce lower intra-cluster dissimilarity and higher inter-cluster dissimilarity, and so higher quality clusters.

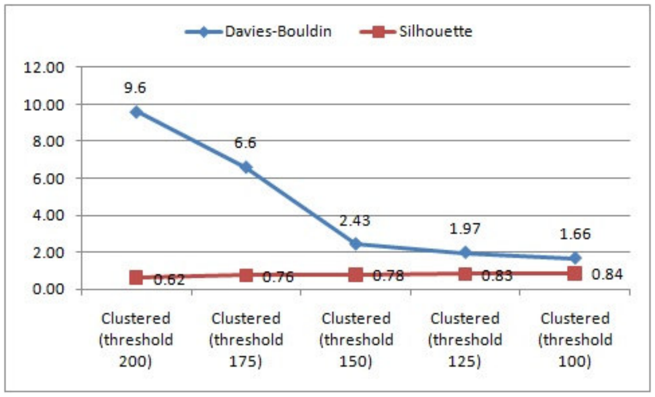


Figure 13. Quality indexes for different thresholds.

VI. Conclusion

Based on the preliminary results, it can be clearly evinced as benefits the potential of the proposed approach in significantly clustering an X-ARM repository and consequently reducing storage space requirements. It must be highlighted that, the bigger the original repository in terms of the number of stored assets, the more expressive the likelihood of clustering assets, and so the better the gain in terms of storage space requirements.

Taking into account that the indexing technique proposed by Brito *et al.* [7] adopted for indexing the clustered repository, the experiments reveal the reduction in the size of the original repository implies in an expressive reduction in the size of index files of the clustered repository. Besides, considering that the technique proposed by Brito *et al.* has an excellent performance in query processing time, even in large-scale index files, it is expected a reasonable gain in terms of query processing time due to the expressive reduction in the size of index files. Therefore, the proposed approach clearly makes possible to map large software component repositories into small index files.

However, as often informally said, there is no free lunch. That is, in formal words, such expressive gains in terms of storage space requirements and query processing time have an impact on the query precision level, since the process of clustering assets introduces some degree of information loss in representative assets. For the experiments of the previous section the query precision level vary from 0.41 for the threshold of 100 to 0.31 for the threshold of 200. Such results can be considered very attractive because, as indicated in experiments presented in [22], highly popular and adopted search engines like Google and Altavista have achieved inferior precision indexes around 0.29 and 0.27, respectively. Moreover, in all thresholds the recall index is about 0.67. Again, such results can also be considered interesting because, as also indicated in [22], Google and Altavista have obtained inferior recall indexes around 0.20 and 0.18, respectively.

Although these preliminary results indicate the usefulness of the approach, a large number of experiments must be performed to better evaluate the heuristics and the similarity metric here introduced. In these experiments we must investigate several configurations of the repository differing on the amount of assets of each type, as well as the possibilities of relations among them. Besides, it is also under investigation a comparative analysis contrasting the proposed heuristics and other ones available in the literature, but applied in different research fields.

ACKNOWLEDGMENT

This work was supported by the National Institute of Science and Technology for Software Engineering (INES – www.ines.org.br), funded by CNPq, grants 573964/2008-4.

REFERENCES

[1] A. Orso, M.J. Harrold, and D.S. Rosenblum, “Component Metadata for Software Engineering Tasks”, Proc. 2nd Int. Workshop on Engineering Distributed Objects, 2000, pp. 126-140.

[2] P. Vitharana, F. Zahedi, and H. Jain, "Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis", IEEE Transactions on Software Engineering., vol. 29, issue 7, July 2003, pp. 649-664.

[3] OMG, Reusable Asset Specification: OMG Available Specification – v2.2, 2005.

[4] G. Elias, M. Schuenck, Y. Negócio, J. Dias, and S. Miranda, “X-ARM: An Asset Representation Model for Component Repository”, Proc. 21st ACM Symposium on Applied Computing (SAC 2006), France, 2006, pp. 1690-1694.

[5] W. Meier, “eXist: An Open Source Native XML Database”, NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems, 2002.

[6] R. Goldman and J. Widom, “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases”, Proc. 23rd Int. Conf. on Very Large Data Bases (VLDB 1997), Greece, 1997, pp. 436-445.

[7] T. Brito, T. Ribeiro, and G. Elias, “Indexing Semi-Structured Data for Efficient Handling of Branching Path Expressions”, 2nd Inter. Conf. on Advances in Databases, Knowledge, and Data Applications (DBKDA 2010), France, 2010, pp. 197-203.

[8] M.P. Paixão, L. Silva, T. Brito and G. Elias, “Large Software Component Repositories into Small Index Files”, Proc. 3rd International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA 2011), pp. 122-127, 2011.

[9] A.K. Jain and R.C. Dubes, Algorithms for Clustering Data, Prentice Hall, 1984.

[10] R. Xu and D. Wunsch, “Survey of Clustering Algorithms”, IEEE Transactions on Networks, vol.16, issue 3, May, pp. 645-678.

[11] A. Feng, “Document Clustering – An Optimization Problem”, ACM SIGIR 2007, pp. 819-820.

[12] S. Mancoridis, B.S. Mitchell, C. Rorres, Y.F. Chen, and E.R. Gansner. “Using automatic clustering to produce high level system organizations of source code”. Proc. IEEE International Workshop on Program Comprehension, pp. 45–53, 1998.

[13] B.S. Mitchel and S. Mancoridis. “On the Automatic Modularization of Software Systems Using the Bunch Tool”, IEEE Transaction on Software Engineering, vol. 32, issue 3, pp. 1-16, March 2006.

[14] Y. Chiricota, F. Jourdan, and G. Melançon, “Software Component Capture using Graph Clustering”, Proc. IEEE International Workshop on Program Comprehension, 2003.

[15] J. Wu, A.E. Hassan, and R.C. Holt, "Comparison of Clustering Algorithms in the Context of Software Evolution", Proc. 21st Int. Conf. on Software Maintenance, 2005, pp. 525-535.

[16] Z. Li, Q.A. Rahman, and N.H. Madhavji, “An Approach to Requirements Encapsulation with Clustering”, Proc. 10th Workshop on Requirement Engineering, 2007, pp. 92-96.

[17] M. Cohen, S.B. Kooi, and W. Srisa-an. “Clustering the Heap in Multi-Threaded Applications for Improved Garbage Collection”, Proc. of the 8th annual Conference on Genetic and Evolutionary Computation, Vol. 2, pp. 1901-1908, July 2006.

[18] W. Frakes and K. Kang, “Software Reuse Research: Status and Future”, IEEE Transactions on Software Engineering, vol.31, issue 7, July 2005, pp. 529-536.

[19] D.L. Davies, and D.W. Bouldin, “A Cluster Separation Measure”, IEEE Trans. Pattern Anal. Mach. Intelligence, vol. 1, pp. 224–227, 1979.

[20] S. Theodoridis and K. Koutroumbas, Patternn Recognition, Academic Press, 2009.

[21] P.J. Rousseeuw, “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis”, Journal of Computational and Applied Mathematics, vol 20, pp. 53-65, 1987.

[22] S.M. Shafi and R.A. Rather, “Precision and Recall of Five Search Engines for Retrieval of Scholarly Information in the Field of Biotechnology”, Webology, vol. 2, number 2, 2005.