Large Software Component Repositories into Small Index Files

Marcos Paulo Paixão, Leila Silva
Computation 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 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.**

Keywords - Component repositories; indexing; clustering techniques.

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.* [11], 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 [13], 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 [10] and X-ARM [3].

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 [9], and also query precision level [6]. In such a scenario, the proposal presented by Brito *et al.* [1] represents a noticeable indexing technique based on semi-structured data, which can be considered precise and efficient in terms of query processing time, but suffer 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. 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.

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 [15] present an interesting review of the research field. In [4], Feng shows that clustering algorithms, in particular, hierarchical algorithms and K-Means [7], are equivalent to optimization algorithms of a fitness function.

In this paper, a two-stage, heuristic clustering approach is proposed, based on the classical hierarchical algorithm and K-Means. In order to validate the proposed approach, a random database composed of 27.000 assets has been generated and results indicate that there is a significant optimization in terms of the number of assets to be indexed.

However, for the best knowledge of the authors, clustering 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. Despite the mentioned originality, several other proposals have already adopted clustering techniques in problems of the software engineering. For instance, Wu *et al.* [14] compares several clustering approaches proposed in the context of software evolution. In [8], Li *et al.* proposes the adoption of clustering techniques for encapsulating software requirements. Chiricota *et al.* [2] investigates the application of clustering techniques in the domain of reverse engineering, in particular, adopting such techniques to recover the structure of software systems.

III. X-ARM

In order to express syntactic and semantic features of software components, Frakes [5] 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 [3]. 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.

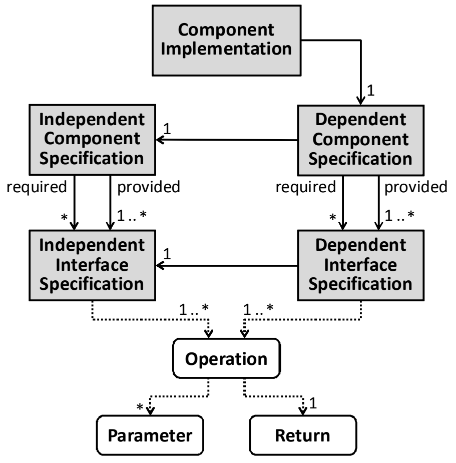

Component Implementation
1
Independent Component Specification
1
Dependent Component Specification
required
provided
required
provided
*
1..*
*
1..*
Independent Interface Specification
1
Dependent Interface Specification
1..*
1..*
Operation
*
1
Parameter
Return

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 assed 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>
```

Figura 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 [1][6][9]. 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 [7] 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 [7] 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.

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

In turn, for each 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 considerable subset of provided dependent interface specifications.

In relation to independent component implementations, the relevant feature of each one is its 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.

Taking into account dependent interface specifications, the relevant features of each one are its 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.

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

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 attribute vector of the asset. Since the attribute vector differs between distinct types of assets, the similarity metric is also different for each type of asset. Due to space limitation, the adopted metrics are not completely described (see [12] for details). In order to illustrate the composition of the metric, consider the case of determining the similarity between two dependent component specifications. In such a case, if two dependent component specifications have the same reference to a given independent component specification, then a certain value, called distance, is assigned to the similarity among them. Besides, the intersection and union sets of their provided dependent interface specifications are calculated. A weight is assigned to the ration among the size of the intersection and union sets in such a way that dependent component specifications are considered more similar when the ration is closer to one, and considered more different when the ration tends to zero.

As an example of calculating the similarity metric, consider the dependent component specifications that have the attribute vectors illustrated in Table I and II. The similarity between them is established using their attribute vectors. Such a similarity is expressed by a numeric value, which can be calculated according the following equation:

$$D_f = D_i - k - (intersection/union)*100. \qquad (1)$$

In Eq. (1), the terms have the following values. The term D_i is a default initial distance (D_i = 300). In turn, the term k can be the value 200, if both specifications make reference to the same dependent component specification, or otherwise the value 0. The term *intersection* expresses the number of provided dependent interfaces that both specifications have in common. Finally, the term *union* represents the number of provided dependent interfaces that both specifications have together. In the example, D_i = 300, k = 0; *intersection* = 1 and *union* = 3. Thus, D_f = 300 – 0 – 33 = 267.

TABLE II. ATTRIBUTE VECTOR OF THE ASSET Y.

ID	compose.dependentCompSpec-Y-1.0-beta
Independent Component Specification	compose.independentCompSpec-W-1.0-beta
Dependent Interface Specification	compose.dependentIntSpec.A-2.0-stable compose.dependentIntSpec.C-1.0-stable

C. *Clustering Algorithm*

The proposed clustering algorithm has two stages. In the first stage, the classical hierarchical clustering algorithm [7] is applied adopting the concept of threshold. Thus, the clustering algorithm is performed until the distance between the clusters is greater than the threshold, which is specified by the user. For each identified cluster, a representative asset is constructed and stored in nonvolatile memory. In order to make the performance better, the implementation of the algorithm loads the assets to be clustered in volatile memory, until reaching its storage capacity. During such a stage, the assets are randomly selected from the repository.

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.

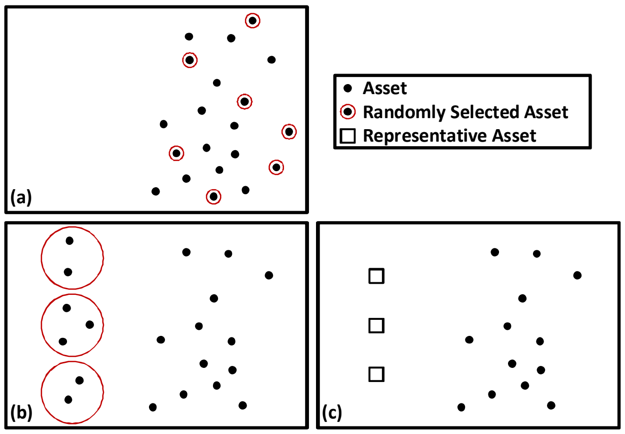


Figura 3. The first stage.

In the second stage, a K-Means based algorithm [7] 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.

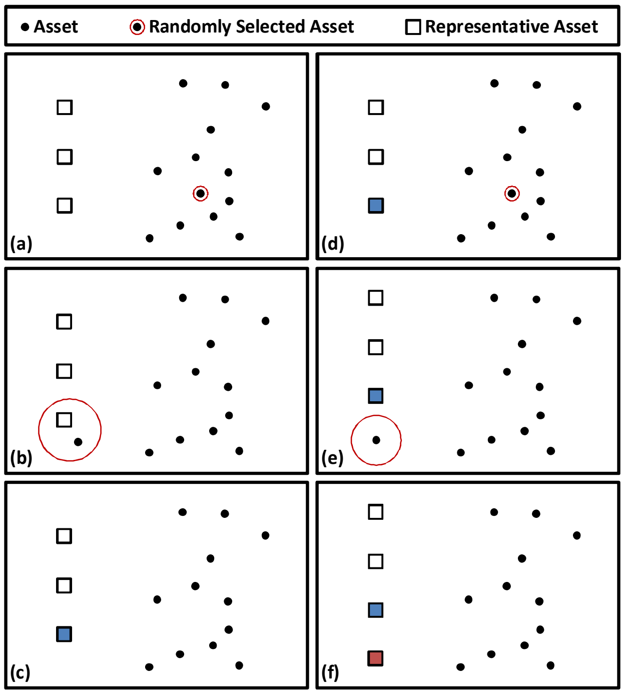
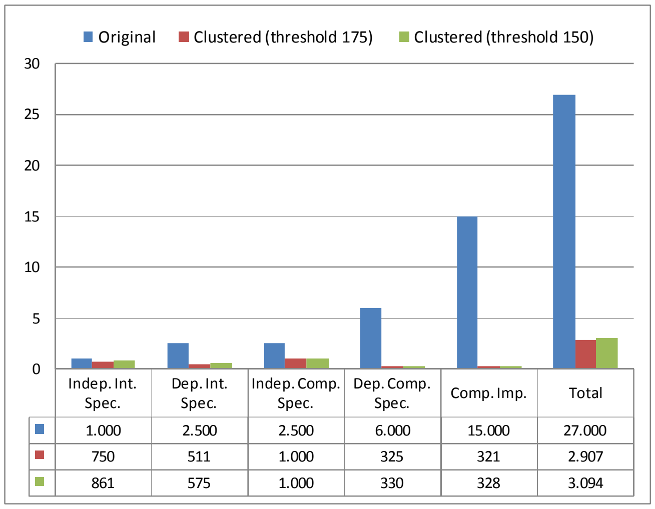


Figure 4. The second stage.

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.

V. Performance Evaluation

In order to evaluate the proposed approach, it has been developed a customizable script that automatically generates a repository that stores the mentioned X-ARM assets. The generated repository has 27.000 different types of assets. After creating the repository, the proposed approach has been applied for grouping the stored assets in clusters, generating their respective representative assets. Fig. 5 presents the number of each type of asset in the original repository and the clustered repositories after the application of the proposed approach using the threshold of 175 and 150.



	Indep. Int. Spec.	Dep. Int. Spec.	Indep. Comp. Spec.	Dep. Comp. Spec.	Comp. Imp.	Total
🟦	1.000	2.500	2.500	6.000	15.000	27.000
🟥	750	511	1.000	325	321	2.907
🟩	861	575	1.000	330	328	3.094

Figure 5. Number of Assets.

As can be noticed in Fig. 5 and Fig. 6, the proposed approach significantly reduces the original repository. For example, when the threshold is 175, the number of stored assets in the original repository is reduced around 89,2%, dropping from 27.000 original assets to 2.907 representative assets. In terms of storage space, the proposed approach reduces the storage space requirements around 43%, dropping from 18 MB in the original repository to 10 MB in the clustered one.

When the threshold is reduced, as expected, more representative elements can be constructed because more clusters are created. Thus, when the threshold is reduced from 175 to 150, the number of original assets is reduced from 27.000 to 3.094 representative assets, which still represents a significant reduction in the number of stored assets around 88,5%.

Consequently, as illustrated in Fig. 6, in terms of the number of assets, the gains of applying the proposed approach are significantly relevant, varying between 89,2% and 88,5% for the thresholds of 175 and 150, respectively.

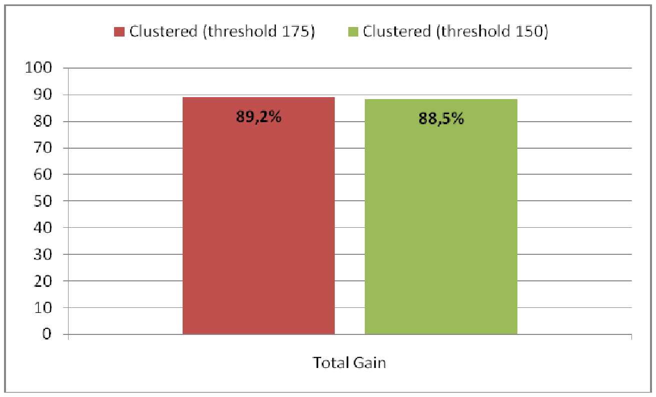


Figure 6. The total gain

However, as depicted in Fig. 7, the gains are different for each type of asset. For independent interface specifications, the gains are around 25% and 13,9% for thresholds of 175 and 150, respectively. Such lower gains can be explained by the difficulty of finding two or more interfaces that has a reasonable set of common operations, which are evaluated in terms of their names, the types of their input and output attributes, and also the type of their return values.

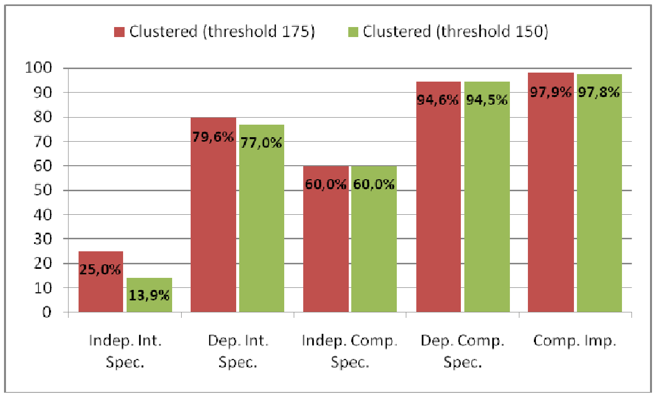


Figure 7. The gains for different types of assets

Considering dependent interface specifications, the gains become more expressive, increasing to 79,6% and 77% when thresholds are 175 and 150, respectively. Part of the reason for that is that, during the generation of the original repository, the adopted script creates 2 or 3 dependent interfaces that refer to the same independent interface, representing that each independent interface is specified for at least 2 or 3 different component models in practice. So, as the similarity metric for dependent interfaces is based on their referenced independent interface together with their operations, it is already expected such expressive gains, as demonstrated in the experiments.

In relation to independent component specifications, the gains are around 60% for both thresholds. Such gains are relatively high and indeed not expected. However, as mentioned before, independent component specifications are considered similar when they have in common a considerable subset of provided independent interfaces. Considering that independent interfaces have expressive clustering rates, such gains make possible to group several interfaces in a unique representative interface, increasing the likelihood of independent component specifications to refer to the same provided interfaces, and consequently, justifying the high gains for both thresholds.

In terms of dependent component specifications, the gains become much more expressive, increasing to 94,6% and 94,5% when thresholds are 175 and 150, respectively. The rationale for that is that, during the generation of the original repository, the adopted script creates 2 or 3 dependent components that refer to the same independent component, representing that each independent component is specified for at least 2 or 3 different component models in practice. So, such better gains are understandable because the similarity metric for dependent components is based on their referenced independent components, which already have expressive clustering rates.

Finally, for component implementations, the gains become higher, around 97,9% and 97,8% when thresholds are 175 and 150, respectively. Again, the rationale for that is that, when generating the original repository, the adopted script creates 2 or 3 component implementations for each dependent component specification, representing that each dependent component has at least 2 or 3 different implementations in practice. So, such higher gains are expected because the similarity metric for component implementations is based on their referenced dependent components, which already have expressive clustering rates.

As can be noticed, the clustering gains in independent interfaces specifications impact on the gains in both dependent interfaces specifications and independent component specifications. Similarly, 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.

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.* [1] will be adopted for indexing the clustered repository, it is taken for granted that 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, almost certainly have an impact on the query precision level, since the process of clustering assets introduces some degree of information loss in representative assets. It must be stressed that the tradeoff between the best threshold and the query precision level has not yet been investigated. In such a sense, the evaluation of the impact of the proposed approach in terms of query processing time and precision level constitutes future work. Besides, it is also under investigation a comparative analysis contrasting the proposed approach 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] 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.

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

[3] 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.

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

[5] 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.

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

[8] 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.

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

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

[11] 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.

[12] M.P.S. Paixão, T.B. Viana, L. Silva, and G. Elias, G. "Optimizing the Search Space in Distributed Component Repositories", Technical Report, June 2010. http://www.compose.ufpb. br/reports/component-repository-cluster.pdf (in Portuguese).

[13] 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.

[14] 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.

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