ISSN: 1803-3814 (Printed), 2571-3701 (Online)
https://doi.org/10.13164/mendel.2019.1.031

SENSITIVITY ANALYSIS OF EVOLUTIONARY ALGORITHM FOR REUSABLE SOFTWARE COMPONENT IDENTIFICATION

Amit Rathee, Jitender Kumar Chhabra

Department of Computer Engineering, National Institute of Technology, Kurukshetra, India
amit1983_rathee@rediffmail.com, jitenderchhabra@gmail.com

Abstract

Fast and competitive software industry demands rapid development using Component Based Software Development (CBSD). CBSD is dependent on the availability of the high-quality reusable component libraries. Recently, evolutionary multi-objective optimization algorithms have been used to identify sets of reusable software components from the source-code of Object Oriented (OO) software, using different quality indicators (e.g. cohesion, coupling, etc.). Sometimes, these used quality indicators are quite sensitive towards the small variations in their values, although they should not be. Therefore, this paper analyzes the sensitivity of the evolutionary technique for three quality indicators used during the identification: Frequent Usage Pattern (FUP), Semantic and evolutionary coupling. The sensitivity analysis is performed on three widely used open-source OO software. The experimentation is performed by mutating the system to different degrees. Results of the empirical analysis indicate that the semantic parameter is most sensitive and important. Ignoring this feature highly degrades the quality; FUP relation is uniformly sensitive and evolutionary relations's sensitivity is non-uniform.

Keywords: *evolutionary algorithm, optimization, software component, FUP, sensitivity, robustness, CBSD.*

Received: 30 April 2019
Accepted: 5 June 2019
Published: 24 June 2019

1 Introduction

Component Based Software Development (CBSD) is a software engineering approach that is widely being used in software development in order to reduce effort, cost and time. It highly promotes the software reuse principle. It promotes software development by accentuating the use of reusable software components in the design and development of the system [12, 14]. It also promotes the software reuse by defining, implementing, and composing loosely coupled independent components under a well-defined software architecture [7]. A component is a coarse-grained collection of fine-grained classes, objects, and relationships. The CBSD success is dependent on the availability of a good quality component library that aims at promoting software-for-reuse reusability [9]. Software-for-reuse is the ability to build applications that can be used as all or part of it in another application. Reusable components can be extracted from the source code of existing well-structured and tested OO applications. These extracted components can be reused while developing other similar applications by using them from the component library. Many different approaches are proposed in the literature for extracting reusable components from the source-code of existing OO applications or legacy systems [3, 5, 8, 6, 1].

Evolutionary multi-objective optimization techniques are widely being used in component identification from underlying source-code either directly or indirectly by performing clustering [13, 10, 11, 15]. These evolutionary techniques are based on various features present between different software elements, namely structural, semantic and evolutionary relations. These features are used as independently or in combination to perform clustering based on different quality indicators. The authors in [13] proposed a direct approach for the identification of reusable software component using the underlying source-code and show that the evolutionary algorithms generally surpass the traditional ones. The authors utilize three kinds of relations together namely structural, semantic and evolutionary. The structural relations are identified as Frequent Usage Pattern (FUP) sets [12] and the semantic relations are identified by tokenizing the source-code at the software element level. Similarly, evolutionary coupling(also called as evolutionary relation) is extracted from the change-history of the system. The multi-objective NSGA-III algorithm is used for clustering different elements as components based on three quality indicators. First two objectives, measure the cohesion based on structural and semantic relations and the third measures the coupling based on the evolutionary relations.

All approaches for the reusable software component identification from source-code using evolutionary algorithm aim at clustering different software elements. Multiple clustering solutions exist out of which those solutions are acceptable which possess better quality. Each individual component's characteristics are reflected using different structural/semantic relations quality indicators such as FUP etc. This paper uses three types of characteristics namely FUP, tokenizing based semantic and evolutionary coupling. These three parameters are collectively henceforth referred in this paper as quality indicators. It is important to study that how sensitive is the clustering corresponding to a slight variation in the quality indicators used for the clustering [4]. Normally, the clustering approach will be considered as more sensitive if a small variation in a quality indicator results into a larger change in the clustering solution, and vice versa. Hence, it is important to study the sensitivity of these quality indicators to help determine the robustness of the proposed approach. Any approach that is less sensitive to the deviation in the quality indicator values is said to be robust and vice-versa [4]. Such approaches are always preferred by software developer's community. Thus, the sensitivity analysis of any approach is very important in order to determine whether the approach is robust or not. In this paper, we perform the sensitivity analysis of our reusable component identification approach as proposed in [13]. Here, the sensitivity is analyzed by doing variations in the three quality indicators and analyzing the obtained results.

The authors have planned this paper as follows: Section 2 gives the related works description, Section 3 provides the background of the approach being analyzed. Section 4 gives details about the sensitivity analysis approach used. Section 5 gives details about the experimental setup. Section 6 explains the experimental results. Finally, section 7 details about the conclusion and future works.

2 Related Works

The related works are divided into two categories. The first category surveys and presents the most recent work towards the identification of reusable software components. The second category presents the details about the literature work engaged towards the sensitivity analysis.

Shatnawi et al. proposed a re-engineering approach for converting an OO API into its equivalent component-based API based on the frequent usage pattern mining technique [15]. The authors have extracted the underlying frequent usage pattern based on the interaction between the OO API and the number of client applications using this API. Kebir et al. [17] proposed a genetic algorithm-based approach for the automated refactoring of component-based software. The authors applied the genetic algorithm for detecting bad smells and removing them by finding the best sequence of refactoring actions. The authors in [13] proposed a multi-objective reusable software component identification approach from the source-code of a software system.

The authors in [16] performed the sensitivity analysis of a clustering approach called K means-sharp (k-means#) that was proposed as an alternative to the classical k-means. The authors performed the sensitivity analysis in order to determine the algorithm's capability for detecting the outliers. Zou et al. performed the sensitivity analysis of an integrated software reliability analysis platform in order to determine the location of the most sensitive code blocks in the system [18]. The authors in [10] performed the sensitivity analysis of different quality indicators in the software module clustering process.

3 Background Study

This section discusses in brief about the approach under investigation for Reusable Components Identification (RCI) from source-code of an OO software system [13]. The authors in [13] proposed a multi-objective search-based evolutionary approach. It identifies the reusable components set as different mutually exclusive clusters using the NSGA-III algorithm. The NSGA-III algorithm helps in obtaining the optimized grouping of underlying software elements in terms of cohesion and coupling. The proposed approach is dependent on three kinds of relations existing among different elements namely Frequent Usage Pattern (FUP) relations, semantic relations, and evolutionary relations. The FUP and semantic relations are used to measure the cohesion using two different characteristics, whereas the evolutionary relations are used to measure the co-change coupling. These coupling and cohesion values are measured using the metric proposed in [13] and are used as three different objective functions for the NSGA-III algorithm. The computation of cohesion and coupling is dynamic in nature and is always computed at the component level during different iterations of the NSGA-III algorithm. The brief discussion about the different objective functions are as follows:

3.1 Frequent Usage Pattern (FUP) Information-based Cohesion

The FUP based cohesion makes use of the extracted FUP information at the software element level (classes) and the metric proposed in [13]. Here, the FUP information extracted at the element level is defined as the set of different member variables of the software that is directly or indirectly used within it. The indirect use refers to the member variable uses due to the function call to other elements. This FUP information of a software element is represented in the form of a vector. The computation of the cohesion using the proposed metric is based on this representation of FUP. The NSGA-III algorithm groups different elements such that FUP based cohesion at cluster level is maximized.

3.2 Semantic Information-based Cohesion

The semantic-based (conceptual) cohesion makes use of the lexical information extracted at the software element level by tokenizing the underlying source-code. The token extraction is done from three main parts of the source-code: 1) class/ interface names along with the names of inherited classes/ interfaces, 2) data type names for different member variables excluding elementary data types; and 3) signature of different member functions. The extracted information is represented in the form of a vector and is used to compute the semantic-based cohesion using the metric proposed. The computation of semantic cohesion is again at the component level. It is computed dynamically during different iterations of the NSGA-III algorithm. The dynamically computed semantic cohesion value is used as an objective function and is always maximized.

3.3 Co-change based Coupling i.e. Evolutionary Relations

In order to measure the co-change based coupling, the maintained change-history of the software system is used. The change-history represents the evolutionary relations among different software elements. The extracted change-history information is represented in the form of a vector and is used to represent the software system in the form of a commonly used representation named as Module Dependency Graph (MDG) [19]. The MDG represents different software elements as nodes, the co-change coupling as the edges between nodes. The well-known Support & Confidence metrics are used to measure the co-change coupling (referred as evolutionary relations henceforth) among different software elements. The computed co-change coupling is used as the objective function in the NSGA-III algorithm and is always minimized.

4 Sensitivity Analysis Approach

The sensitivity analysis for any approach aims at determining how a slight change in the values of the quality indicators affects the quality. Various steps used to determine the sensitivity of the approach proposed in [13] are depicted in Fig. 1.

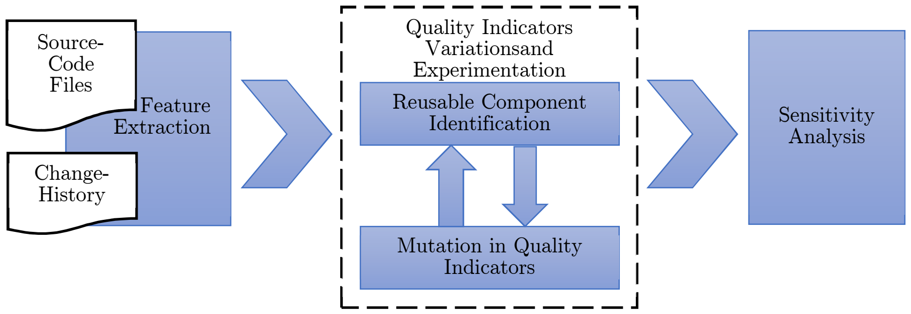
Source-Code Files
Change-History
Feature Extraction
Quality Indicators Variationsand Experimentation
Reusable Component Identification
Mutation in Quality Indicators
Sensitivity Analysis

Figure 1: The proposed Sensitivity Analysis Phases.

This paper aims at determining the sensitivity of three key dependency relations used by [13] for the identification of reusable components. This is done by mutating them to different degrees. This analysis helps the researchers in weighing their future proposed approaches based on the sensitivity of different considered dependency relations. The sensitivity analysis process, as depicted in the Fig. 1, consists of three main steps. The first step includes the extraction of the three main features on which the quality indicators are based. The FUP and Semantic features are extracted from the source code of the system under study. The evolutionary relations are extracted by parsing the change-history reports of the considered system. The second step includes the application of the proposed approach by [13] on the extracted features of the system. It is repetitive in nature and is repeated for each of the variations introduced in the extracted relations of the system. During experimental evaluation of the sensitivity, we have introduced 0%, 5%, 10%, 15%, and 20% variation in the system and repeated our experimental study for each of the mutated systems. Here, for each of the experimentation, the underlying results are recorded. The third step targets the sensitivity analysis of the results obtained during different experimentation performed in a second step. Here, the sensitivity analysis is measured by computing the amount of deviation corresponding to the deviation introduced in the quality indicator function. This deviation is computed in terms of percentage, using equation (1).

$$\text{percentage deviation} = \Delta(\mathrm{x}) = \frac{\|f(x') - f(x)\|}{\|f(x)\|} \tag{1}$$

Here, f(x) denotes the value of the underlying quality indicator corresponding to the ideal system in which 0% variation is performed. Similarly, $f(x')$ denotes the value of the underlying quality indicator measured for the mutated system.

5 Experimental Setup

The experimental analysis in this paper is performed on 3 real-world systems. Table 1 summarizes these different studied systems. The considered dataset consists of open-source software systems that are commonly being used for experimentation among the research community [15, 14]. To perform the experimentation, firstly all three kinds of relations are extracted by utilizing a custom tool designed by the authors, which analyses the underlying source-code and the corresponding change-history available for the studied system. In the second step, the studied systems are analyzed for different deviations. This deviation is performed in the identified three kinds of dependency relations: FUP, Semantic, and Evolutionary. The values of these relations are modified within the range of 0-20%, and this change is called as mutation, which is one of the common terms used to change the data in genetic algorithms. The modified values represent a new mutated system corresponding to the specific relation and helps us to analyze the importance of the corresponding relation for the process of reusable component identification. During the analysis, the approach is applied multiple times for a different percentage of mutation introduced in the three kinds of relations.

Table 1: Overview of the Studied Software Systems.

ID	Software Names	Version	# Classes	# Commits	Periods	Description
1	JFlex	1.5	61	1254	3-11-2006 To 30-11-2018	A lexical analyzer generator for Java.
2	JUnit	4.10	164	5643	1-1-2003 To 30-11-2018	A testing framework used for java program testing
3	JDOM	1.1.3	62	1630	3-10-2002 To 30-11-2018	A project to access, manipulate, and output XML data from Java code.

The percentage mutation varies in the range of 0-20% interval. The proposed approach applies the NSGA-III algorithm on the mutated systems and measures the performance of the identified reusable software components using well-known TurboMQ metric [2], where MQ represents Modularization Quality. TurboMQ is a standard metric used to measure the design quality of software in terms of intra-connectivity and inter-connectivity of different components of the software. Higher value of TurboMQ is always desirable which indicates more intra-connectivity within the component (also called as cohesion) and less inter-connectivity with other components (also called as coupling). Two or more individual reusable components are grouped together to form a cohesive cluster and all such clusters with higher values of TurboMQ are identified in the cluster solution of the Pareto front obtained. Finally, the obtained results are investigated using the metric proposed in equation (1). As part of the analysis, the following research questions are formulated:

RQ1. How does the value of the quality indicator (TurboMQ) vary with the mutations in the system? It aims at checking the robustness of the approach under study my measuring TurboMQ value for different percentages of mutations introduced in the software under study.

RQ2. What is the importance of different relations in identifying reusable components from source-code? It aims at checking the individual importance of each dependency relation used as the basis of the approach under study. The investigation is done by varying single relation while keeping others unchanged. To analyze the results, well-known F1-score metrics are used. Here, the comparison is done between a mutated system and the actual 0% mutated system.

6 Results and Analysis

This section of the paper presents the experimental results and discussions about the sensitivity of the system. The sensitivity analysis is performed by answering the research questions that are formulated.

6.1 RQ1

Table 2 shows the value of the quality indicator obtained for different versions of the mutated system. Column 2 gives the percentage of mutation introduced in the studied system. Columns 3, 4 and 5 give the corresponding quality indicator values using TurboMQ as an overall quality indicator for different versions of the mutated system.

Table 2: TurboMQ Quality Indicator Values.

S.N.	% Mutation	TurboMQ Quality Indicator Value		
		JFlex	JUnit	JDOM
1	0	2.143	1.432	1.862
2	5	2.023	1.360	1.769
3	10	1.928	1.289	1.676
4	15	1.821	1.217	1.583
5	20	1.715	1.146	1.489

Fig. 2 shows the plot for the percentage of change in quality indicator value for each percentage of mutation for different studied systems. Here, we plot the actual obtained values (solid line) and show the ideal linear trend line (dashed lines) for each system. The difference between these two lines helps distinguish how the approach deviates from the ideal case. This difference between these two lines helps us to determine the robustness of the approach under analysis. Fig. 2 also shows the linear regression analysis performed using R-squared statistical measure. It helps us identify the Goodness-of-Fit for a linear model and denotes how close the data is to the fitted regression line. The average obtained value of this statistical measure is 99% and it denotes that the analyzed approach is capable of handling all the variability of the response data around its mean. The deviations in the used quality indicator's (shown as a solid line) value (TurboMQ) are very much closer to the ideal line (shown as dotted). Here, the ideal line is linear because, for a robust approach, the deviations are directly proportional to the mutation introduced into the system. The two lines in Fig. 2 are very much close to each other. Hence, from the analysis, it is clear that the variations in the quality indicator are consistent with the introduced mutation in the system. Hence, the studied approach is robust in nature.

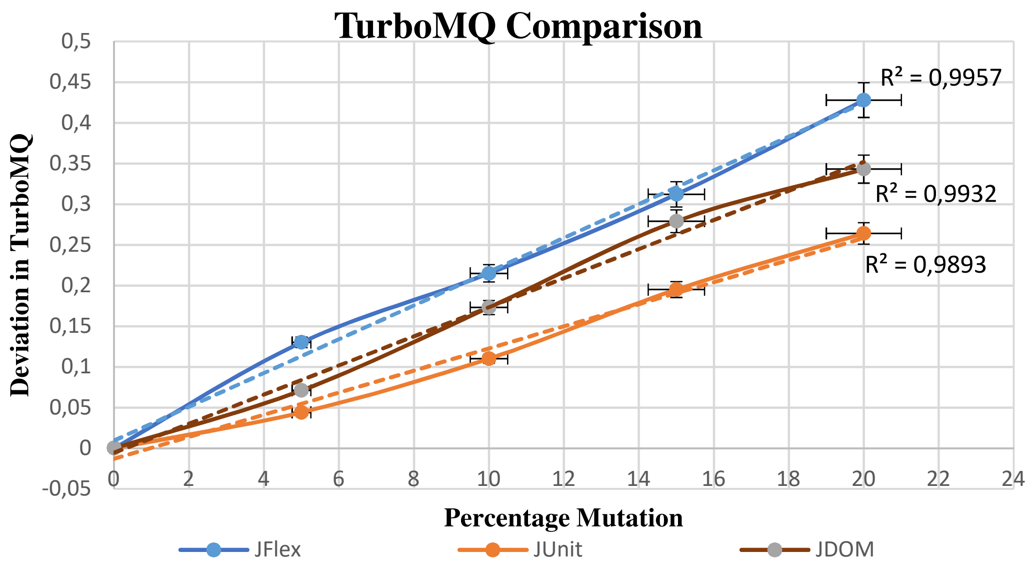

Figure 2: Plotting of deviation in Quality Indicator vs Percentage Mutation to Systems.

6.2 RQ2

In order to test the importance of different relations in the reusable component identification, each of the individual relations is mutated one by one while keeping the other two relations fixed. Table 3 shows the experimental results under three schemes: (i) only FUP relations mutated, (ii) only semantic relations mutated, and (iii) only evolutionary relations mutated. The values in the table denote the obtained F1-score (Here, 0% mutation is considered as the base while measuring F1-score under other mutations). Fig. 3 shows the boxplots of F1-score for three studied systems. Here, the comparison is done under three constituted schemes for each studied system.

Table 3: F1-Score under Different Individual Mutation in Relations

Sr	**% Mutation**	**F1-Score**								
		Only FUP Relations Mutated			**Only Semantic Relations Mutated**			**Only Evolutionary Relations Mutated**		
		JFlex	**JUnit**	**JDOM**	**JFlex**	**JUnit**	**JDOM**	**JFlex**	**JUnit**	**JDOM**
1	0	97%	98%	95%	97%	98%	95%	97%	98%	95%
2	5	94%	96%	92%	92%	94%	91%	95%	97%	92%
3	10	89%	91%	89%	85%	87%	83%	91%	93%	87%
4	15	85%	87%	85%	73%	81%	74%	85%	91%	80%
5	20	81%	82%	81%	59%	74%	62%	83%	88%	78%

F1 score is widely used here for statistical evaluation of our experiment. It indicates the amount of change in the reusability value based on percentage of mutation. For example, if we refer to 4th row (15% mutation) and 4th column (only Semantic relation for JFlex) in Table 3, 73% F1-score value indicates that 15% mutation in semantic relation brings down the reusability metric value to 73% (very sensitive). On the other hand, same 15% mutation in evolutionary relation for Junit changes it to 91 % (less sensitive). In Fig. 3, it can be clearly observed that the boxplot under the scheme where only the FUP/ semantic relation is mutated keeping the other two relations unchanged, shows more variations for every studied system. For FUP as well as semantic relations, this variation is more than the corresponding mutation done in the system, and it is higher for semantic than FUP. Hence, it can be concluded that when only semantic relation is mutated, the deviation in the quality indicator (F1-score) is highest as compared to other schemes, making it most sensitive, critical and necessary for the studied approach. In addition, by doing a manual inspection, we confirmed that the Semantic relation is vital for the approach as it helps in grouping related classes and interfaces together. Similarly, when the only FUP relation is mutated during experimentation, the variation in F1-score is proportional to the mutation percentage indicating its reasonable level of sensitivity. On the other hand, variation in quality corresponding to change in evolutionary (co-change) relation is normally lesser than percentage of mutation, but for JDOM software, it is similar or more also. Hence, it can be claimed that semantic relations are most sensitive and FUP relations are normally sensitive. The evolutionary relations are less sensitive for JFlex and JUnit software, but more sensitive for JDOM. This conclusion is backed by the fact that during mutation we randomly left out some of the corresponding dependency relations (0% - 20% for the different degree of mutation). Therefore, at least the deviation in the overall software quality should be proportional to the percentage of mutation done. Finally, when the only evolutionary relation is mutated, the results are not consistent. For example, in JDOM system, the variations are more whereas in Junit case the variations are less. Similarly, for JFlex system, the deviation is proportional. One possible reason for this variation can be that the evolutionary relations are dependent on the accuracy and timely maintenance of the change-history for a software system. If it is not timely and properly maintained, then, the correct dependency relations are not reflected.

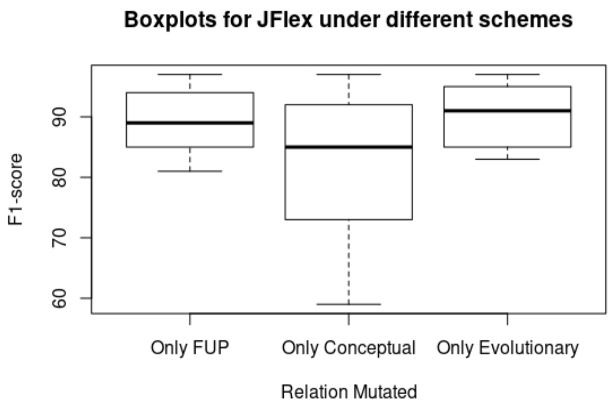

Figure 3a: Boxplot for JFlex.

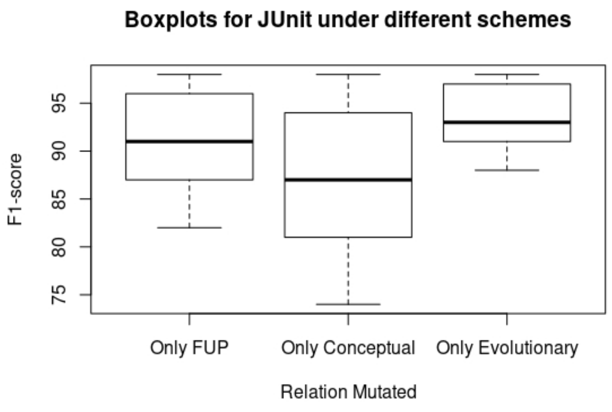

Figure 3b: Boxplot for JUnit.

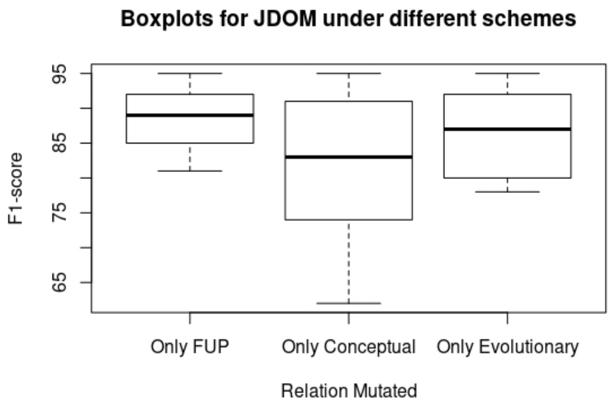

Figure 3c: Boxplot for JDOM.

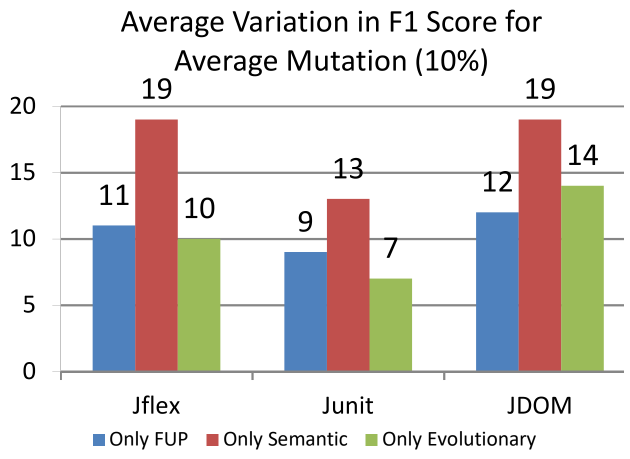

Figure 3: Average Variations in F1-score for Different Relation Schemes for Average Mutation.

In the above Fig. 4, the first bar for all three software corresponds to mutation in FUP only and it indicates that FUP variation is near 10 % for all three software. The second bar corresponds to the most sensitive parameter i.e. semantic and its average value in all three case is more than 10 %. For the evolutionary (co-change) relation, sensitive is less than 10 % for JUnit, but more than 10 % for JDOM. Hence it is re-assured that semantic relations are most sensitive and important, FUP relations are uniformly sensitive and evolutionary relations are not uniform.

7 Conclusion and Future Works

This paper has presented an approach to evaluate the robustness of the evolutionary algorithm based approach for reusable component identification of the software. The paper has carried out a detailed sensitivity analysis of the studied approach. The aim of the analysis is to determine which of the underlying three parameters (dependency relations) affects the reusable component identification from source-code most. As a part of experimentation, we have also defined a metric to access sensitivity. In order to check the robustness, the experimentation is performed by mutating the system with different degrees (0% to 20%) and measuring the quality of new solutions using the TurboMQ quality indicator. Similarly, in order to check the sensitivity of different parameters, we have formulated three schemes in which are mutated only a single relation at a time and keeping the others unchanged. The obtained results clearly indicate that the semantic relation of the studied approach is the most sensitive parameter. The FUP relation is uniformly sensitive and the sensitivity of evolutionary relation is non-uniform. Future works will focus on topics including comparisons with different approaches, sensitivity analysis of resource allocation problems, and other sensitivity of software attributes.

References

[1] Alshara, Z., Seriai, A.-D., Tibermacine, C., Bouziane, H. L., Dony, C., and Shatnawi, A. 2015. Migrating large object-oriented applications into component-based ones: Instantiation and inheritance transformation. In *GPCE 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* ACM New York, NY, USA, Vol 51, Issue 3, pp. 55–64. DOI: 10.1145/2936314.2814223

[2] Andritsos, P. and Tzerpos, V. 2005. Information-theoretic software clustering. *IEEE Transactions on Software Engineering* 31, 2, pp. 150–165.

[3] Birkmeier, D. Q. and Overhage, S. 2013. A method to support a reflective derivation of business components from conceptual models. *Information Systems and e-Business Management* 11, 3, pp. 403–435.

[4] Harman, M., Swift, S., and Mahdavi, K. 2005. An Empirical Study of the Robustness of two Module Clustering Fitness Functions. In *Proceedings of GECCO'05.* ACM, Washington DC, USA, pp. 1029–1036.

[5] Hasheminejad, S. M. H. and Jalili, S. 2015. CCIC: Clustering analysis classes to identify software components. *Information and Software Technology* 57, pp. 329–351.

[6] Kebir, S., Seriai, A. D., Chardigny, S., and Chaoui, A.: Quality-centric approach for software component identification from object-oriented code. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture.* No. 13115578, IEEE, pp. 181–190. DOI: 10.1109/WICSA-ECSA.212.26

[7] Lau, K.–K. 2004. *Component-Based Software Development: Case Studies.* (Series on Component-Based Software Development), World Scientific Press, New Jersey.

[8] Mishra, S., Kushwaha, D., and Misra, A. 2009. Creating reusable software component from object-oriented legacy software through reverse engineering. *Journal of Object Technology* 8, 5, pp. 133–152.

[9] Ouyang, Y. and Carver, D. L. 1996. Enhancing design reusability by clustering specifications. In: *Proceedings of the 1996 ACM symposium on Applied Computing.* ACM, Philadelphia, Pennsylvania, USA. DOI: 10.1145/331119.331431

[10] Prajapati, A. and Chhabra, J. K. 2014. An empirical study of the sensitivity of quality indicator for software module clustering. In *Seventh International Conference on Contemporary Computing (IC3).* No. 14583918, IEEE. DOI: 10.1109/IC3.2014.6897174

[11] Prajapati, A. and Chhabra, J. K. 2018. MaDHS: Many objective discrete harmony search to improve existing package design. *Computational Intelligence,* pp. 1–26. DOI: 10.1111/coin.12193

[12] Rathee, A. and Chhabra, J. K. 2017. Improving Cohesion of a Software System by Performing Usage Pattern-Based Clustering. In *Inter. Conf. on Smart Computing and Communication (ICSCC).* Procedia Computer Science, Elsevier, Vol. 125, pp. 740–746.

[13] Rathee, A. and Chhabra, J. K. 2019. A Multi-Objective Search Based Approach To Identify Reusable Software Components. *Journal of Computer Languages* 52, pp. 26–43.

[14] Rathee, A. and Chhabra, J. K. 2018. Clustering for software remodularization by using structural, conceptual and evolutionary features present in software artifacts. *Journal of Universal Computer Science* 24, 12, pp. 1731–1757.

[15] Shatnawi, A., Seriai, A. D., Sahraoui, H., and Alshara, Z. 2017. Reverse engineering reusable software components from object-oriented APIs. *Journal of Systems and Software* 131, pp. 442–460.

[16] Olukanmi, P. O. and Twala, B. 2017. Sensitivity analysis of an outlier-aware k-means clustering algorithm. In *2017 Pattern Recognition Association of South Africa and Robotics and Mechatronics (PRASA-RobMech).* No. 17520825, IEEE, pp. 68–73. DOI: 10.1109/RoboMech.2017.8261125

[17] Kebir, S., Borne, I., and Meslati, D. 2017. A genetic algorithm-based approach for automated refactoring of component-based software. *Information and Software Technology* 88, pp. 17–36.

[18] Zou, B., Yang, M., Benjamin, E. R., and Yoshikawa, H. 2017. Reliability analysis of Digital Instrumentation and Control software system. *Progress in Nuclear Energy* 98, pp. 85–93.

[19] Prajapati, A. and Chhabra, J. K. 2017. Improving Modular Structure of Software System using structural and lexical dependency. *Information & Software Technology* 82, pp. 96–120.