A MULTI-OBJECTIVE SEARCH BASED APPROACH TO IDENTIFY REUSABLE SOFTWARE COMPONENTS

Amit Rathee, Jitender Kumar Chhabra

PII: S1045-926X(18)30183-6
DOI: https://doi.org/10.1016/j.cola.2019.01.006
Reference: COLA 889

To appear in: *Journal of Computer Languages*

Received date: 11 October 2018
Revised date: 30 November 2018
Accepted date: 3 January 2019

Please cite this article as: Amit Rathee, Jitender Kumar Chhabra, A MULTI-OBJECTIVE SEARCH BASED APPROACH TO IDENTIFY REUSABLE SOFTWARE COMPONENTS, *Journal of Computer Languages* (2019), doi: https://doi.org/10.1016/j.cola.2019.01.006

Highlights

- The Reusable Software Component Identification problem is modeled as a searchbased Multi-Objective optimization using three kinds of relations among different software elements namely FUP, semantic relatedness and co-change relations.
- A new cohesion metric known as Rank, which measures the relatedness among different software elements, is proposed.
- Suitable identification and design of objective functions to model it as a Multi- Objective search- based optimization is proposed. Moreover, to increase the effectiveness of search-based optimization, a modified representation scheme for chromosomes is also proposed.
- The proposed approach to identify reusable components is evaluated using following two methods: 1. A well-known modularity metric known as TurboMQ is used to evaluate the modularity of original system and the obtained components. 2. The need for the simultaneous use of all three relations together is further confirmed by formulating total six different schemes. These schemes are designed by considering different relations as individual, pairwise. Finally, these formulated schemes are compared against the proposed approach (using all three relations together).
- The proposed approach is statistically analyzed by using two statistical tests namely ANOVA and two-tailed t-test. ANOVA test is used for validating the fact that the obtained results are statistically significant over total 30 runs of the NSGA-III algorithm. The two- tailed t-test is used to determine the statistical significance of the proposed approach against six formulated schemes

A MULTI-OBJECTIVE SEARCH BASED APPROACH TO IDENTIFY REUSABLE SOFTWARE COMPONENTS

Amit Rathee[a], Jitender Kumar Chhabra[b]

[a]*National Institute of Technology, Kurukshetra- 136119, Haryana (India)*
[b]*National Institute of Technology, Kurukshetra- 136119, Haryana (India)*

Abstract

Component-based-software-development (CBSD) is one of the most recent trends in the software development industry and its success majorly depends on the quality of the software components. Good quality software components are those components, which are internally strongly cohesive and are independent of others. Use of such components helps in faster development of a software and reduces the maintenance efforts in future. Such components can be identified from different software repositories and can be reused whenever needed. Hence, proper identification of such reusable components is a promising area of research, and the same has been targeted in this paper. This paper model Reusable Software Component Identification (RSCI) problem as a search-based multi-objective problem in order to identify optimized reusable software components from the object-oriented (OO) source code of a software system. For OO software paradigm, we consider the component as an individual class or group of connected classes which can be reused with least modifications. To identify this, three types of relationship are proposed in this paper, namely 1) Frequent Usage Pattern (FUP) based cohesion 2) semantic relatedness based cohesion and 3) co-change based coupling. The proposed approach optimizes by simultaneously maximizing both types of cohesion and minimizing the coupling of a given component. The FUP based cohesion maximization is based on the newly proposed cohesion metric called *PatternCohesion* which is computed using FUP information extracted from

Email addresses: amit1983_rathee@rediffmail.com (Amit Rathee), jitenderchhabra@gmail.com (Jitender Kumar Chhabra)

a given software element (class/ interface in an object-oriented language). The FUP of a class comprises of those member variables of all classes of the software, which are directly or indirectly accessed by member functions defined in the class under consideration. The *PatternCohesion* metric helps to measure the functional relatedness among different pairs of software elements (classes). The semantic relatedness based cohesion maximization is based on TF-IDF based Cosine Similarity measurement using tokens extracted from three main parts of the source code of a given software element namely class/ interface declaration statement, data types of different member variables and member function signatures. The co-change based coupling is computed from the change-history of the underlying software system using well-known data mining metric called Support. Finally, reusable components are identified as different cohesive groups (consisting of one or more connected classes) using NSGA-III Algorithm. The proposed approach is empirically evaluated over six open source software systems belonging to different domains. The need of considering all three types of relations is established by comparing their performance with relations taken individually and two at a time. The obtained results indicate a higher quality for different software components measured in terms of reusability characteristics.

Keywords: Software Component, Clustering, Functional Cohesion, Frequent Usage Patterns, Rank Score, Component Extraction, Component Identification, Multi-Objetive, Search-Based Optimization

1. Introduction

Software reusability is one of the latest software engineering techniques that help software development with improved quality, shorter development time, effort and reduced maintainability [1]. Reusability is defined as the process of using existing software assets repeatedly in software development. It plays a major role in Component-Based Software Engineering (CBSE). CBSE is a reuse-based approach of developing large complex software from loosely coupled independent software elements known as Components. A component is considered as a collection of several related objects that interact with each other in order to provide a specific system functionality [2]. In CBSE, a component can provide services to other components or it may require other components services in order to implement its intended functionality. To the best of our knowledge, there does not exist any IEEE/ISO

standard definition of the software component. However, [3] studied and categorized components based on a proposed classification scheme derived from various distinguishing factors. From their discussion, one of the identified categories ofthe component is Logical in nature and it is different from others in that this category of components do not consider technology, environmental and other constraints into account while defining them.
Software component identification in literature is studied at two levels: design level and object-oriented (OO) source code level. Several approaches have been proposed for component identification at the software design level [4, 5, 6]. All these approaches focus on extracting functionality of the underlying system and mapping it to a model in order to finally partition them to extract reusable components. Similarly, many approaches have been proposed for component identification from OO source code [7, 8, 9, 10]. These proposed approaches focus on component identification by analyzing the dependency relations among different software elements and using different clustering algorithms. In our present work, software components are identified from the source-code of a given software system.
The Reusable Software Component Identification (RSCI) is mandatory for utilizing the full potential of CBSE as it helps to build component library [11]. This component library helps to achieve reusability in software development using CBSE approach. The RSCI from source-code of a software system are identified as a group of related software elements and requires different elements to be clustered together. This helps us build a component library containing high quality and pretested components. Moreover, the growing library will enhance the scope for reusability using CBSE approach. The RSCI considers component identification from source-code of a software system as a graph-based problem in which different components (initially single class and later a group of classes) are represented as the vertex and the dependency among different component is represented with labeled edges. The identified components are obtained as the set of classes that are highly cohesive and least coupled in terms of functionality measured in terms of shared characteristics in the object-oriented paradigm.
The RSCI problem can be modeled as a search-based optimization problem as identified in this paper and discussed in section-3. In literature, several multi-objective search-based techniques are proposed and are mainly been used to perform the software remodularization [12, 13, 14]. Most of the work in the field of RSCI from the source-code of a software system is mainly based on analytical techniques [4, 5, 6, 7, 8, 9, 10]. Analytical techniques are

based on using greedy heuristics, hierarchical clustering, and other similar approaches. These are generally non-search based optimization techniques. They are supposed to provide solutions that are more accurate but their performance degrades as the size of the software system increases [15]. In such cases, these techniques require exponential computational time and hence make various soft computing techniques such as GA suitable targets to apply. Therefore, in this paper, a scalable multi-objective search-based approach has been proposed to identify reusable software components from the source code of a software system. Here a component is considered as a group of highly cohesive and loosely coupled functionally related software elements. The cohesion measurement for the identified reusable software components has been based on two factors- the Frequent-Usage Pattern (FUP) shared among different software elements and the semantic relatedness among them. Similarly, the coupling measurement for the same is computed in the literature by calculating co-change based dependency using the available change-history of the underlying software system. Hence, making the RSCI problem as multi-objective. The number of objectives (fitness functions) used is three out of which two maximizes the cohesion (FUP based cohesion and semantic relatedness) and one together minimizes the coupling (co-change dependency) computed at the component level.

This research paper first models the RSCI from source-code of a software system as a multi-objective search-based optimization process. It considers the software component as a logical one and identifies them as a unit containing one or more software elements. It also considers optimizing cohesion and coupling together. The cohesion of the identified software component is evaluated using two newly proposed metrics: FUP based *PatternCohesion* metric, semantic-based cohesion metric. Similarly, the coupling is evaluated as co-change coupling computed from the underlying change-history of the software system. The obtained results are compared using TurboMQ metric. The main contributions of this paper are summarized as follows:

1. The RSCI problem is modeled as a search-based multi-objective optimization using three kinds of relations among different software elements namely FUP, semantic relatedness, and co-change relations.
2. A new cohesion metric known as *PatternCohesion*, which measures the relatedness among different software elements, is proposed.
3. Suitable identification and design of objective functions to model it as a multi-objective search- based optimization is proposed. Moreover,

to increase the effectiveness of search-based optimization, a modified representation scheme for chromosomes is also proposed.

4. The proposed approach to identify reusable components is evaluated using the following two methods:
   (a) A well-known modularity metric known as TurboMQ is used to evaluate the modularity of original system and the obtained components.
   (b) The need for the simultaneous use of all three relations together is further confirmed by formulating total six different schemes. These schemes are designed by considering different relations as individual, pairwise. Finally, these formulated schemes are compared against the proposed approach (using all three relations together).
5. The proposed approach is statistically analyzed by using two statistical tests namely ANOVA and two-tailed t-test. ANOVA test is used for validating the fact that the obtained results are statistically significant over total 30 runs of the NSGA-III algorithm. The two- tailed t-test is used to determine the statistical significance of the proposed approach against six formulated schemes.

This research paper is organized as follows: section-2 provides a literature survey, section-3 gives an overview of the RSCI problem and how it can be modeled as a search-based multi-objective optimization approach. Section-4 gives a detailed description of the proposed approach, section-5 provides the experimentation details. Section-6 gives details of various experimental results that are obtained. Section-7 gives details of various threats to the proposed approach. Finally, section-8 summarizes and gives future possible work in this direction.

2. Literature Survey

In recent years, academician and software developers have shifted their interest to CBSE to minimize development cost, reduced maintenance efforts, etc. Components being the focus of CBSE, so, a lot of work is already tried in this direction that makes use of varieties of metrics, clustering and/ or pattern detection approaches. This section of the paper summarizes the research work already done in this direction by different researchers.
Washizaki et al. [16] proposed an approach to convert the reusable parts of

an OO software system into corresponding reusable JavaBeans components. Marx et al. [17] propose an approach for component identification from a set of predefined entities by recursively using refactoring along with Dependency Inversion Principle (DIP) in order to finalize the set of entities as a component. The proposed approach by [18] for extracting reusable components make use of a hierarchical clustering algorithm by modeling the system as a weighted graph. [19] proposed a set of guidelines and a set of metrics that help to re-engineer an OO software system into its equivalent component-based one. The proposed approach makes use of various composition and aggregation relations present among classes to re-engineer the underlying software systems architecture into corresponding component-based architecture. Chardigny et al. proposed a quasi-automatic approach called ROMANTIC that focus on recovering the underlying component-based architecture of an OO software system using the semantic and structural characteristics of the system [20]. In this approach, the recovered component-based architecture is obtained as the partition of the systems classes and they termed each partition as a component. Shatnawi et al. proposed an approach to mine the reusable software components from a set of similar OO software systems belonging to similar domains [21]. The mined software components are supposed to have higher reusability as compared to the software components which are extracted from only a single OO software system. Here, in the proposed approach, the grouping of different software elements into a component is based on the proposed quality measurement model.
Quinlan et al. [22] proposed an automated approach to identify components from legacy scientific software systems by extracting the language-independent part from source code. Theyconsider only those methods that are reusable from a user point of view and created interfaces from them. Kim et al. [23] studied and presented a method for connecting components together that have heterogeneous interfaces. However, they considered components in binary form rather defining them at source code level. Kebir et al. [8] proposed a bottom-up approach to mine component from OO source code by using the hierarchical clustering technique. The authors make use of a newly proposed software quality metric based on the specificity, autonomy and composability characteristics of a modeled software component. [24, 25] proposed an approach to transform an OO software system into its corresponding equivalent component-based one. They extracted the underlying services offered by the system by utilizing the use case, sequence diagrams and class collaboration diagrams. Finally, the extracted services are grouped to

define components. Similarly, Allier et al. proposed an approach to transform an OO software system to its equivalent component-based system. However, they defined their approach at run-time to identify the dynamic dependencies instead at design level [26]. [27] proposed a stable software component identification approach by utilizing use case diagrams and formal concept analysis (FCA) technique. [9] proposed a method to automatically transform an OO software system into its ready to use the component-based system. The proposed approach analyzes the underlying software system, both dynamically as well as statically, to finally cluster underlying classes using a hybrid genetic-algorithm and simulated-annealing meta-heuristic. [28] proposed an approach to mine reusable components from object-oriented APIs by finding frequent usage patterns using various client software using those APIs. Li et al. [29] presented a semi-automated framework that extracts reusable components from legacy OO software systems. They did this by proposing a reference model that arranges the extracted software elements into hierarchical order to finally group them as self-contained components. [30] proposed an approach for defining the provides and requires interfaces and for the extracted software component by capturing the dynamic relations using the execution traces of the system. The proposed approach results in a complete restructuring of the underlying extracted software component so that they are directly implementable within the concrete component framework. 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 [31]. 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.

Mancoridis et al. [32, 33] first suggested the search-based approach for module clustering. The approach aims to find an optimal solution in terms of MQ defined based on the connectivity between modules. Praditwong et al. [13] tried to improve search-based module clustering by considering the coupling and cohesion factor of MQ, which are in trade-of relationships to each other, as two separate objectives. They used a multi-objective search algorithm, called Pareto optimization algorithm, to find the Pareto front that consists of various solutions with various degrees of the trade-off between coupling and cohesion factors. Bavota et al. [34] proposed an interactive approach for integrating developers' knowledge in module clustering. They extended the MQ formula to penalize the meaningless clustering from the developers' point of view. [35] presented a new version of the differential evolution al-

gorithms that provides provision for self-adaptive control parameter settings specially for numerical problems. Amarjeet et al. [14] proposed a fuzzy Pareto-dominance based Artificial Bee Colony (FP-ABC) multi-objective based software-clustering approach. The authors validated the proposed approach by comparing it with other multi-objective optimization algorithms. Moreover, [36] studied about two important factors in a search-based solution of a problem namely exploration and exploitation in evolutionary algorithms by performing extensive survey over 100 existing works.

In literature lot of approaches have been proposed for software component identification. Existing RSCI techniques rely on measuring dependencies arising due to method calls, type sharing, use-case, sequence, class collaboration diagrams and/ or dynamic traces. These approaches further use various clustering techniques and/ or uses the model to perform grouping and identify components. However, no such approach focuses on measuring dependencies based on the member variable usage (FUP) in the system. It is the opinion of the authors that FUP based dependency in necessary for reusable component identification. Some of the approaches focus on APIs and they rely on the interaction between the API and client applications to model dependencies. The search-based approaches in the literature mainly focus on re-modularization of the software system. The re-modularization is different from RSCI in that it necessitates the presence of every software element in the final re-modularized solution. This condition is however not always applicable to RSCI. Therefore, to the best of our knowledge, no work is based on modeling the software component identification problem as a search-based problem. Moreover, some of the proposed component identification approaches in the literature are based on the dynamic analysis of the systems execution derived from the execution traces. The dynamic analysis approach is considered more complex and cumbersome as compared to the static analysis technique. The dynamic analysis may sometimes miss out some of the underlyingarchitectural classes because of not covering all the functionality defined in a given software system during a dynamic analysis [9]. This contributes to the research gap present in the literature and our proposed research work in this paper fills the research gap identified.

3. Reusable Software Components Identification (RSCI): An Overview

Software industries are currently demanding enhanced reusability in software development process in order to save development time and maintenance

cost. One method to enhance reusability in software development process is to shift to Component-Based Software Development (CBSD) paradigm [2]. The CBSD focuses on reusing the existing functionality (code) available in the form of components having predefined interfaces (provides and requires). However, CBSD requires that the needed functionality should be independent from others in order to be reused directly. Moreover, in order to reuse with CBSD, such independent functionality must be present as a component in the component library. This requires that the reusable components (set of independent functionality) need to be identified from existing software systems. This need of identifying components from source-code creates the Reusable Software Component Identification (RSCI) problem. In essence, RSCI is defined as an act of partitioning/ clustering the existing set of software elements (classes/ interfaces in Object-Oriented software) into groups (aka modules) such that the elements in a group are having higher cohesion and least coupling to other software elements present in other groups [37]. Mathematically, the software partitioning problem for a system having total N elements as $S = E_1, E_2, ..., E_N$ can be defined as the partitioning of the set S into another modular set of size k as $M = m_1, m_2, ..., m_k$ such that the following properties are satisfied:

- Each modular unit m_i is a subset of S i.e. $m_i \subseteq S$.
- Each modular unit m_i is non-empty in nature i.e. $\forall m_i \in M, m_i \neq \emptyset$.
- Each software element E_i can belong to only one modular unit m_1 at any instant of time, i.e., each modular unit is pairwise disjoint in nature, $\forall m_i, m_j; m_i \cap m_j = \emptyset$.
- The partitioning problem is having completeness property such that $\bigcup_{i=1}^{k} m_i = S$.
- Each modular unit m_i is having higher intra relations and lower inter relations in final partitioned solution.

The process of partitioning/ clustering aims to increase the cohesion within the cluster and decreases the coupling of the cluster with other clusters. Maximization of cohesion and minimization of coupling will lead to the best possible partitioning/clustering. Hence, RSCI is modeled as an optimization

problem. Moreover, in RSCI, we need to consider all permutations of grouping different software elements as a component and this complexity increases exponentially as the size (total number of software elements) of the software increases. Hence, the RSCI is modeled as a search-based optimization problem. Further, optimization problems can be both single/ multi-objective based on one or more criteria of optimization. Since the RSCI in our proposed approach is based on optimizing both cohesion and coupling. Therefore, we have formulated RSCI as a multi-objective problem. In this paper, the RSCI is done from the existing OO source code of the software and this problem is modeled as a multi-objective search-based clustering approach. The following sub-sections give details about the multi-objective search-based approach in general and how it can be modeled to solve RSCI problem. Furthermore, various objective functions used to evaluate the quality of solutions in search-based approach are discussed.

3.1. Multi-Objective Search-Based Software Engineering

Recently, Multi-Objective Evolutionary Optimization (MOEO) techniques gained popularity and are frequently being used to solve different software engineering problems [38]. MOEO is generally defined as an optimization process in which more than one (multiple) objective functions are simultaneously optimized (minimized and/or maximized) to have non-dominated Pareto-optimal solutions. For RSCI problem, the non-dominated Pareto-optimal solution is also desired because we are targeting both cohesion and coupling of a component together rather them as an individual. Hence, we need non-dominated solutions. Mathematically, it can be stated as follows:

$$\begin{cases} Max/Minf(x) = & [f1(x), f2(x), ..., fM(x)] \\ g_j(x) \geq 0; & j = 1, ..., P \\ h_k(x) = 0; & k = 1, ..., Q \end{cases}$$

Here, M is the total number of objective functions formulated for the given optimization problem identified as $f_1(x), f_2(x), ..., f_M(x)$ and generally $M > 2$. P denotes the total number of inequality constraints identified as $g_j(x)$. Similarly, Q denotes the set of equality constraints that must be satisfied by the optimization problem and is identified as $h_k(x)$. The solution at any instant is identified by x and X (s. t. $x \in X$) denotes the set of search space out of which feasible set of decision vectors (non-dominated Pareto-front) is

identified. In our RSCI problem, the decision vectors are based on the information extracted using three kinds of dependency relations considered in the proposed approach namely FUP, semantic, and co-change. Moreover, the obtained solution x represents the clustering of different software elements belonging to a software system. The set X must satisfy both inequality and equality constraints simultaneously in order to be a feasible solution.
In the current formulation of the MOEO, all objective functions must be simultaneously minimized in order to get a final optimized solution. However, MOEO techniques also support the maximization of an objective function and are implemented by negating the objective functions to be maximized using the duality principle. MOEO techniques generally result in non-dominated solutions for the optimization problem such that if x_1 and x_2 are two solutions than x_1 is said to dominate other solution x_2 iff:

$$\begin{cases} f_i(x_1) \leq f_i(x_2) & s.t. \forall i \in (1, ..., M) and \\ f_i(x_1) < f_i(x_2) & s.t. \exists i \in (1, ..., M) \end{cases}$$

3.2. Multi-Objective Reusable Software Component Identification

The RSCI from the source code of a software system requires the classification of the underlying architecture into modular units (aka software partition problem). By underlying architecture, we mean the original package structure of the software from which the reusable software components are to be identified. Each modular unit should have the characteristic of high cohesion (intra-relations among elements belonging to the modular unit) and low coupling (inter-relations with other modular units elements). The process becomes more complex and time-consuming when a group of classes can also be considered as a reusable component because the group as a whole is very cohesive, although individual classes may not be. The time taken in identifying the independent reusable modular units is directly proportional to the total number of elements present in the system and its complexity increases exponentially as the number of elements increases. This makes the RSCI problem as NP-hard in nature and hence makes the search-based evolutionary techniques suitable to tackle the problem [13, 14]. Although, the deterministic algorithms are capable of providing optimal solutions by evaluating every possible permutation and combination [39]. However, such algorithms are non-scalable and resource intensive with the increase in the size of the problem [40]. Thus making soft-computing techniques a good solution to handle

the exponential growth for non-scalable optimization problems. In this paper, the chromosomes for a search-based solution is represented as a linear integer array of size N where N is the total number of software elements present in the system. Each index of the linear array represents a unique software element and the corresponding integer number denotes the group number to which it belongs. Each integer value of the linear array varies between 1 and N and they are randomly assigned in order to preserve the diversity among population generated and to search every possible opportunity of grouping. In chromosome representation, first t indices denote classes and the rest of the indices represents the interfaces. The possible values at interface indices are chosen randomly from values in first t indices in order to associate interfaces with classes. Thus, the chromosome representation represents the possible partition of the system and together identifies various elements that belong to different groups. Figure-1 shows the representation of a typical chromosome used in the paper. Here, it may be noted that the current representation of the chromosome guarantees that there are no cyclic relations among software elements (i.e. index i has j value and index j has i value) and both the index and values are independent of each other. Moreover, during each iteration, the crossover and mutation operations change only the cluster number of elements such that no cyclic relations are created.

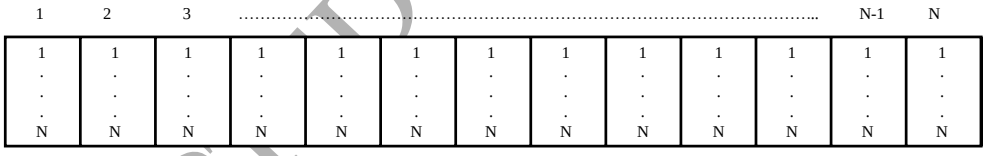


Figure 1: A typical representation of the Chromosome used in Search-Based Optimization.

3.3. Reusable Software Component Identification Objectives

The software elements can be arranged into different groups based on the different quality attributes considered as objective functions. For our RSCI problem, we considered the three metrics that can be used to perform optimization. These metrics are based on three different kinds of dependency relations, namely FUP, semantic and co-change relations. The three proposed metrics consider the cohesion and coupling parameters computed at the group (module) level. The proposed three metrics that are used as

objective functions in our proposed search-based solutions are discussed as follows:

3.3.1. Frequent Usage Pattern (FUP) Information-based Cohesion

The FUP based cohesion measurement helps to group functionally related software elements together in a single modular unit. The authors have already proposed a FUP based cohesion improvement approach in [41]. In this paper, FUP based cohesion measurement is extended to be applied in the the identification of reusable software components. FUP based approach measures cohesion based on the member variable usage pattern shared between different software elements in a software system. Here, if two or more software elements share the same group of variables, then, they must be grouped together in a single unit. FUP based cohesion measurement approach helps to achieve functional based grouping because if two or more software elements are accessing or modifying a similar set of member variables, then all of them are engaged in implementing single or more closely related functionality. The underlying idea of FUP can be more clearly understood by taking a suitable example that consists of two sample classes from a library management software namely *addBookDetails* and *modifyBookDetails*. Both these classes are engaged in modifying the features of the *Book* object present in the library (features available as member variables declared inside these classes) and thus should be always grouped (as a single unit) in order to be reused with least modification.

3.3.2. Semantic Information-based Cohesion

The semantic relatedness among different software elements is again used to increase the cohesion among different software elements belonging to a single modular unit. This is achieved by using a Latent semantic indexing (LSI) information retrieval technique [42, 43]. This technique focuses on tokenizing the underlying source-code of different software elements and measuring semantic relatedness using TF-IDF (Term Frequency Inverse Document Frequency) cosine similarity measure [44, 45]. The tokens extracted from different software elements are finally represented in the vector form. Each index in the vector representation denotes a unique token of the system and corresponding value represents its frequency of occurrence in a given software element. Here, a value of zero for any token of the system denotes that the corresponding token is not present in the software element. The semantic information of each software element in the system is represented in

the form of the vector and it is finally used to calculate the semantic relatedness among a different group of software elements using the cosine similarity measure.

3.3.3. Co-Change based Coupling

The change history of a software system stores and gives information about various software elements that undergo change together over time and thus gives change coupling among different software elements [46]. Such kind of evolutionary dependencies is implicit and they may or may not represent structural relatedness among different software elements. Change coupling has been considered a bad symptom in a software system [47]. Based on the manual study, it is noticed by us that not all classes in the set of co-change based coupled software elements (classes) are functionally related together. Hence, for the RSCI problem, we considered separating such co-change coupled set of classes. Therefore, Co-Change based coupling among different software elements belonging to a modular unit is used as another objective function in order to have a reusable set of components. [48] used the similar co-change based dependency for performing co-change based modularity structure.

The success of the proposed approach depends on the availability of accurate values of these three objective functions. FUP and semantic information can be obtained accurately only if proper design methodologies have been followed during the initial development and meaning naming of different software elements (data members and member methods etc.) have been followed. Similarly, co-change coupling information can be precisely extracted only through properly logged change-history of all iterations of maintenance. In the absence of such information, the results of the proposed approach may not be accurate, as the cohesion, as well as coupling, will not get properly optimized.
The formulation of these three objective functions is discussed in detail later in section-4 of this paper.

4. Proposed Methodology

This section proposes a methodology to identify independent reusable software components available in the source code of a given software system. In our proposed approach, the reusable software components are identified

as a single or a group of software elements (classes/ interfaces) which possess high cohesion among themselves and least coupling to the outer elements. In this paper, first of all, the cohesion and coupling among different software elements are computed using three parameters viz frequent usage pattern (FUP), semantic relatedness and co-change relations obtained from the underlying change history of the software system. The FUP and semantic relatedness are used to measure the cohesion strength and the co-change relations are used to measure the coupling strength.

In the proposed approach, the RSCI problem is modeled as a search-based problem and multi-objective evolutionary algorithm namely Non-dominated Sorting Genetic Algorithm (NSGA-III) is used for clustering different software elements into reusable components (as discussed in detail in subsection-3.2). In search-based modeling, the FUP, semantic and co-change relations are used to formulate multiple objectives in order to identify reusable software components from a given source code of the software. These three kinds of static relations are the basis of these three fitness functions (as discussed in subsection- 3.3) that finally guide the NSGA-III algorithm in identifying reusable software components.

The FUP relation based fitness function defined with the NSGA-III helps us to identify set of structurally cohesive software elements that are finally present in the form of a component (discussed in subsection- 4.1 below). Similarly, the semantic relation based fitness function used with the NSGA-III algorithm helps to simultaneously identify a semantically cohesive component. This makes the identified component both structurally and semantically cohesive in nature. Finally, the co-change based fitness function helps us to make the identified component to be more reusable by optimizing coupling among identified components. Here, it is important to note that the extracted information viz FUP, semantic and co-change relations remains static and do not change during optimization. Even then, the computation of these three fitness functions is dynamic in nature and they needs to be re-computed in each iteration of the NSGA-III algorithm. This is because, the crossover and mutation operations of NSGA-III algorithm changes the membership/ belongingness of different software elements to different groups (aka component in our proposed approach). Hence, the software elements belonging to a component keep on changing from iteration to iteration. It causes the FUP, semantic, and co-change relations belonging to a component changeable in nature. Hence, the cohesion and coupling values should be re-computed in each iteration.

Figure-2 shows a pictorial representation of the proposed approach. A description of how to model the RSCI problem as search-based and formulation of the objective functions has been already discussed in section-3 above of this paper.

The proposed approach for identifying reusable software components from

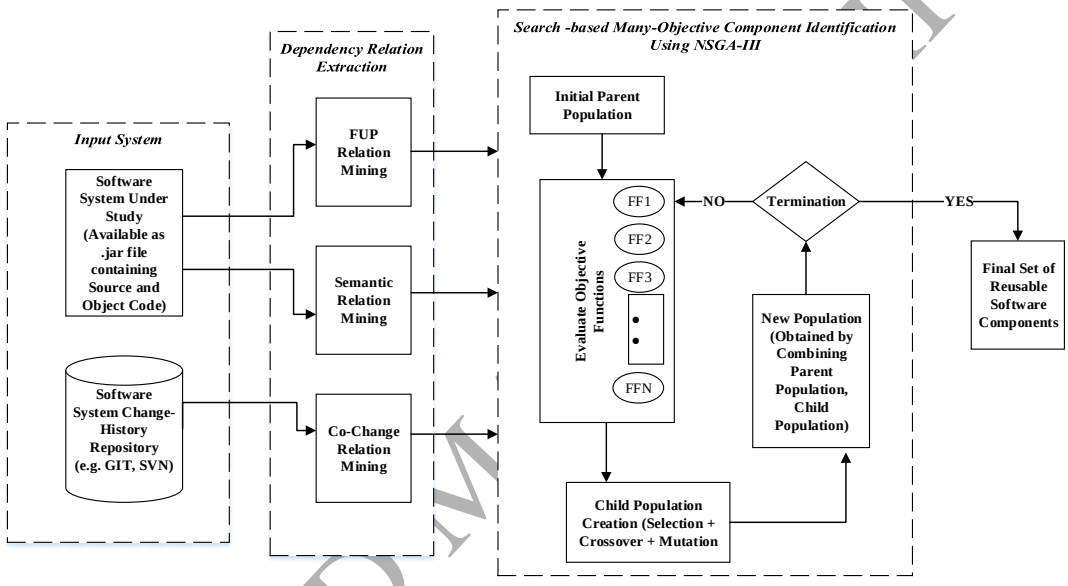


Figure 2: Proposed Methodology for Reusable Software Component Identification from Source Code.

the source code of a software system takes the source code of the software as input. The proposed approach also takes as input the underlying change-history associated with the software system available in the form of GIT/ SVN repository. The change-history of a software system records all changes performed on it during various maintenance activities. It also records which of the underlying software elements are involved in the change.

From the given inputs, first, the proposed approach extracts three kinds of relations among different software elements namely FUP information, semantic relatedness information, and co-change based information. FUP information includes member variable usage patterns, semantic relatedness information includes extracted tokens and co-change based information includes the co-usage based dependency. This information is extracted at a software element

level and constitutes different features present in the software system which finally helps us to identify reusable software components. The vector-based feature representation helps us to implement search-based optimization. All of the information is independently extracted with the help of a tool developed by the authors and is finally available as a different vector for each unique software element belonging to the software system. The vector representation signifies which of the total features are present in the given software elements. A value of zero signifies that the feature is absent in the given software element, whereas any non-zero integer number signifies the extent to which the corresponding feature is present in the given software system. Higher the integer value, more relatedness of the corresponding feature to the given software system. The extracted information (present as a vector for different software elements) is later used to compute different metric values that are used as fitness functions specially designed for the RSCI problem (as discussed in subsection- 3.3). The proposed approach considers solving RSCI problem using the search-based Non-dominated Sorting Genetic Algorithm (NSGA-III). The search-based technique in our proposed approach optimizes three objective functions namely 1) to maximize the frequent usage pattern (FUP) based cohesion; 2) to maximize the semantic relatedness within a modular unit (semantic based cohesion); and 3) to minimize the co-change based coupling determined from the underlying change history of the software system. Here, NSGA-III algorithm is chosen because it gives the solutions as non-dominated Pareto-front and in any case, we cannot further improve the solution without affecting the other parameters considered for optimization. The proposed approach finally gives the reusable software componentsavailable in the form of clusters. The process, which describes how this extracted information is used in different fitness functions, is discussed in detail in the subsequent subsections. The computation steps of the proposed approach are summarized in the form of pseudo code as shown below in Figure- 3:-

Pseudo Code for "Search-based Reusable Software Component Identification"

1) Extraction of different dependency relations

Input: Source-code and Object-code of a software system.

Output: Different dependency relations present in the form of Vectors for each software system.

STEPS:

// Extraction of FUP Information

a) For (i : 1 to N) */* N is the total number of software elements in a software system */*

b) {

c) Parse source-code of each software element i;

d) Extract member variable usage found in different functions belonging to i; */* this includes member variable usage found to same software element i or some other software element j */*

e) Construct usage pattern for i by counting total reference made to each member variable;

f) Represent the usage pattern for i using a Vector FUP_i of size M; */* here M is the total number of member variables in the software */*

g) }

// Extraction of Semantic Information

a) For (j : 1 to N)

b) {

c) Parse source-code of each software element j;

d) Extract elementary tokens from following three parts of j:-

a. Class/ Interface names along with names of inherited classes/ interfaces for software element j;

b. Data type names for different member variables excluding elementary data types;

c. Signature of different member functions;

e) Represent the tokens of j using a Vector SEM_i of size T; */* here T is the total number of unique elementary tokens in the software */*

f) }

// Extraction of Co-Change Information

a) For (k : 1 to N)

b) {

c) For (p : 1 to N)

d) Compute the co-change coupling between software element k and p using *Support* metric;

e) }

f) Represent the co-change information as an MDG; */* the nodes represents different software elements and edges represents coupling */*

2) Proposed search-based multi-objective NSGA-III algorithm

Input: Different dependency relations information available as Vectors/ MDG.

Output: Set of reusable software components available as different clusters.

STEPS:

a) Set various algorithmic parameter values viz *population size, total number of iterations, mutation and crossover rates*.

b) Set various problem-specific parameters such as T_C , T_I *and* T_E where T_C is the total number of classes present in the system, T_I is the total number of interfaces in the system and T_E is the total number of software elements present in the system. Here, $T_E = T_C + T_I$ always. */* these parameters are used in initial population initialization */*

c) // generate Initial populations P_0

For (i: 1 to PS) */* PS is the total number of initial population size */*

{ */* Here, C_j are the different positions in the chromosome representation */*

For (j: 1 to T_C) */* first randomly initialize the cluster number to different classes in the chromosome */*

$C_j \leftarrow RandInt\ (UB_i, LB_i)$ */* UB_i & LB_i are the upper and lower bounds and $UB_i = T_C$ & $LB_i = 1$ **

For (j: 1 to T_I) */* randomly initialize the cluster number to different interfaces in the chromosome */*

$C_j \leftarrow RandInt\ (C_1, C_2, \ldots\ldots, C_{T_C})$ */*here, $C_1, C_2, \ldots\ldots, C_{T_C}$ are the random cluster number generated for different classes of the system. This step ensures that the interfaces will randomly got one of the cluster numbers as assigned to classes */*

}

d) V_0 = Evaluate the fitness functions of the initial population P_0 and choose N_{pop} the feasible solutions.

e) $Iter = 1$

f) While ($Iter \leq Iter_{max}$) */* here, $Iter_{max}$ is the total number Iterations selected for the NSGA-III algorithm */*

g) {

h) Generate a new population of Offspring P_{off} based on P_{Iter};

i) Evaluate, group and sort P_{off} by dominance and crowding;

j) Select N_{pop} individual;

k) }

l) Obtain the set of Reusable Software Components as clusters in the best individual of obtained Pareto-front.

Figure 3: Pseudo ode for Search-Based Reusable Software Component Identification.

4.1. Frequent-Usage-Pattern (FUP) based Cohesion measurement

Member variables in a class reflect the properties of the real-life entities/ objects. Functionalities of the objects are implemented through methods, which operate on various member variables. Common usage of a variable by multiple methods indicates connected among these methods. If these methods belong to different objects, it is a reflector of relatedness of the corresponding objects. Therefore, any overlap in member variables signifies related real-world objects. This concept drives us to compute FUP based relations among different software elements belonging to a software system. In literature, only [31] has used the similar usage pattern mining based approach to convert an object-oriented API into its corresponding component-based API. However, their approach is highly dependent on the selection of different client application software taken as a starting point to measure the interaction between OO API and corresponding client application. This is because, in their proposed approach, the selected client applications are the basis to determine and identify the Frequent Usage Pattern (FUP) for the considered API. Here, the FUP determines which set of classes are used by the various client applications and any client application that is using the services of the underlying API with lesser extent will definitely deviate the results (compared to the client application that uses the API services with higher extent and it can provide more accurate usage patterns about the API). Similarly, we have also proposed a cohesion metric based on FUP in [41]. In contrast, our proposed FUP based cohesion measurement approach in this paper is robust and free from such high-dependencies on the choice of such starting points for FUP extraction. In addition, we have extended our previous cohesion metric of [41] in order to apply it for measuring cohesion at the component level. In this paper, the proposed cohesion metric is capable of measuring cohesion between one or more software elements as compared to [41] where the FUP based cohesion metric always measure cohesion between two software elements at any instant of time. In our proposed approach, the FUP is computed at the class level and is defined as the set of member variables that are directly or indirectly accessed by it. It is computed from the usage patterns of member functions belonging to the class. The usage pattern of a member function is defined as the set of those member variables that are directly or indirectly (by calling other functions) accessed and/ or modified by it. Mathematically, the whole concept of FUP can be further cleared by taking an example of a software element E_i that contains a total m member variable $M_1, M_2...M_m$ and a total n member functions $MF_1, MF_2...MF_n$.

Now, let us suppose that the usage pattern of different member functions is: $MF_1 = \{M_i, M_j ... M_k\}; MF_2 = \emptyset; - - - - -; MF_n = \{M_p, M_q ... M_t\}$, then, the FUP of E_i is represented as the union of all the member function sets, i.e. $FUP(E_i) = \{MF_1 \cup MF_2 \cup .... \cup MF_n\}$. The FUP of a whole software system is obtained by statically parsing and analyzing the OO source code of various software elements belonging to it. After determining the FUP set F_i of each software element E_i belong to the software system, F_i is represented in the form of a vector V_{F_i} of size n, where n represents a total number of member variables defined in a given software system. The vector V_{F_i} is defined as follows:

$$V_{F_i}[j] = \begin{cases} 0; & if M_j \notin F_i \\ \geq 1; & if M_j \in F_i \end{cases}$$

Here, M_j is the j^{th} unique member variable in the underlying system. If it does not belong to FUP set F_i then the corresponding entry is zero otherwise the entry is non-zero and denotes the frequency of occurrence of M_j in F_i . Using this scheme, the software system is represented as a set $V = \{V_{F_1}, V_{F_2}, ..., V_{F_S}\}$ where S is the total number of software elements belonging to the software system. This set V is taken as an input to our search-based reusable component identification approach and is used to compute the cohesion of the modular unit.

FUP based cohesion measurement and using it as an objective function in our proposed search-based solution is based on the idea of up to how much extent a set of software elements share a common usage pattern among themselves. Higher the extent of common sharing, more are the chances to group the software elements together. In this paper, we proposed a new FUP based cohesion computation metric known as *PatternCohesion* metric. It is based on the input set V. The PatternCohesion metric is computed at the modular unit level and for a modular unit U_i containing $E_j, E_k, ..., E_t$ as its software elements, the metric is computed as follows:

$$PatternCohesion(U_i) = \frac{\sum_{1}^{n} Min(Overlap\_Count(E_j, E_k, ..., E_t))}{\sum_{1}^{n} Max(Overlap\_Count(E_j, E_k, ..., E_t)) + \sum_{1}^{n} Max(NonOverlap\_Count(E_j, E_k, ..., E_t))}$$

Here, $PatternCohesion(U_i)$ computation is based on the measurement of two values: Overlap_Count and NonOverlap_Count, which are identified from the corresponding FUP vectors of $E_j, E_k, ..., E_t$ belonging to the modular unit U_i. The proposed *PatternCohesion* metric is dynamically computed during different iterations of NSGA-III algorithm and is also not limited to a module containing only two software elements. This is because, during each iteration, the association of different software elements with different clusters changes due to crossover, mutation and generation of new population. The proposed metric can be easily applied over module of any size (containing any number of software elements).

Overlap_Count counts the matching entries in usage vectors where all the corresponding entries in FUP vectors are non-zero and denotes the similarity in terms of usage pattern. Alternatively, in simple words, the Overlap_Count measure the extent to which different software elements belonging to a component share common usage for different member variables. Higher the sharing more will be the chances that functionally related elements are grouped as a component. Similarly, NonOverlap_Count counts the non-matching entries and identifies the entries where corresponding vectors are non-matching in terms of usage pattern, i.e. at least one vector has the corresponding entry as 0 while others may or may not have them as 0. It measures the extent to which different software elements do not share common usage for different member variables in a given component/ group. Higher the value for NonOverlap_Count, more are the chances that the grouped software elements are not functionally related.

Min and Max are the mathematical minimum and maximum functions that return the minimum and maximum value, respectively out of the corresponding entries of usage pattern vectors. These functions are used in the proposed *PatternCohesion* metric because it helps us to normalize the metric in the range [0...1]. The *Min* is chosen in the numerator of the proposed *PatternCohesion* metric in order to find the best group (closeness of a software element with other elements in the group) for a software element. This is because during optimization using the NSGA-III algorithm, a software element may be a part of different groups. The summation function is used because Overlap_Count and NonOverlap_Count are computed by traversing the FUP vector. Here, the summation help us to determine high cohesion by computing high overlap. This is done indirectly by choosing minimum overlap for a member variable (represented in the form of FUP vector) rather than maximum.

In our proposed approach, the PatternCohesion metric is used as a fitness function. Based on the values of Overlap_Count and NonOverlap_Count it measures the functional relatedness among different software elements in a component. For a given solution, this fitness function is evaluated for each underlying modular unit (specified in solution representation) and overall cohesion of the system is calculated as an average of all the cohesion score of different modular units.

4.2. Semantic relatedness based Cohesion measurement

During our manual investigation, we noted that a software system containing some software elements in the form of interfaces generally do not have member variables defined and contains only member function declarations. Such functions are defined inside other software elements that implement those interfaces. Such software elements (interfaces) are not covered under FUP and their FUP vector is $\emptyset$. So, the PatternCohesion metric is unable to group such elements. However, such software elements group (i.e. interfaces and associated implemented class) possess higher semantic relations. Therefore, semantic relatedness among software elements grouped in a modular unit is a key to increase the cohesion and hence helps in increasing the underlying quality of the reusable component identified.
In the proposed approach, semantic relatedness is obtained by tokenizing the underlying software elements and obtaining tokens from following parts of the source code:

1. *Class Name (CN) Tokens:* this category contains tokens collected from the class declaration statements in source code. It includes all the names specified in the class declaration statement along with the entire inherited software element names (superclasses and interfaces).
2. *Data Type (DT) Tokens:* this category contains tokens collected from the member variables data types declared inside the software element. During this collection, only non-primitive data types are considered and primitive data type tokens are simply discarded. This assumption helps us to improve the underlying quality of the obtained reusable software component.
3. *Function Signature (FS) Tokens:* this category contains tokens collected from the underlying signature of all the member functions defined in the software element. The tokens are extractedfrom function name,

return type and the parameters data type portion of the function signature. While collecting the data type tokens only non-primitive data types are considered.

After collecting tokens from the above-mentioned portions of the source code, the semantic information of each software element E_i is represented in the form of vector $SV_i = T_1, T_2, ..., T_n$. Here, n is the total number of unique tokens available in the underlying software system. The semantic vector SV_i is mathematically defined as follows:

$$SV_i[j] = \begin{cases} 0; & if T_j \notin E_i \\ \geq 1; & if T_j \in E_i \end{cases}$$

Here, T_j denotes the j^{th} unique token in the software system. In the semantic vector SV_i , the corresponding j^{th} entry is zero if the corresponding token T_j does not belong to semantic information collected for the software element E_i . Otherwise, the entry is non-zero and denotes the frequency of the occurrence of the specified token T_i in the source code. In this manner, the semantic information of each software element is represented in the form of semantic vector.

Once the semantic information of the software system is represented as a semantic vector, it is further used to measure the semantic cohesion of the underlying system. Again, the semantic cohesion is calculated at modular unit level. To measure cohesion, TF-IDF based cosine similarity measure method is utilized [45]. The cosine similarity for a given modular unit U_i containing $E_j, E_k, ..., E_t$ as its software elements is calculated using the following formula:

$$Sem\text{-}Coh = \frac{SV_j.SV_k.....SV_t}{\mid\mid SV_j \mid\mid * \mid\mid SV_k \mid\mid *.....* \mid\mid SV_t \mid\mid} = \frac{\sum_{i=1}^{n}(w_{j,i} * w_{k,i} * ..... * w_{t,i})}{\sqrt{\sum_{i=1}^{n} w_{j,i}^2}\sqrt{\sum_{i=1}^{n} w_{k,i}^2}.....\sqrt{\sum_{i=1}^{n} w_{t,i}^2}}$$

Here, $SV_j.SV_k.....SV_t$ denotes the dot product of the corresponding vectors. $||SV_j||$ denotes the magnitude of the vector SV_j and is calculated as the square root of the dot product of the vector with itself. Moreover, $w_{k,i}$ denotes the corresponding weight (token frequency) of token T_i in vector SV_k for software element E_k.

In our proposed approach, TF-IDF based cosine similarity measure Sem_Coh

is used as a fitness function to measure the semantic relatedness (cohesion) among different software elements belonging to a given modular unit and overall cohesion of the system (as specified by the searchbased solution) is calculated as an average of all the cohesion score of different modular units.

4.3. Co-Change based Coupling measurement

According to [49], change coupling (aka co-change/ co-evolution/ logical coupling) is a regular activity linked with periodic co-changes present with the software evolution during its lifetime. Co-evolution of various software elements in a software system can be expressed by their change coupling because of simultaneous undergoing modification [50, 51]. The co-change coupling of any two software elements is based on their change history and is a measure of the observation that the underlying two software elements always co-evolve or change together [52, 49]. The co-change coupling is identified by means of association rule mining approach of data mining where the association among elements is represented as X => Y [53, 54]. Here, X is called Antecedent and Y as Consequent. The association rule signifies that if event X happens then the event Y will also happen at the same time due to their higher possibility of occurring together. In software engineering, it denotes the possibility that if X changes in a commit, then Y, will also undergo modification. Co-Change information is derived by analyzing patterns,relationships and relevant information of source code change mined from multiple versions (of software systems) in software repositories (e.g., Subversion and Bugzilla).
To evaluate different association rules, two criteria are generally used namely Support and Confidence [54]. Support denotes how frequently a given set of items appears together in the dataset and Confidence denotes how many times a given association rule is found to be true. Support for an association rule X = Y is the measure of the probability that both the items X and Y appears together in an event and is calculated as:

$$Support(X = Y) = Support(XY) = C_{XY}$$

Where C_{XY} is the total number of commits in the change history in which both X and Y appeared together. Similarly, Confidence for an association rule X = Y denotes the probability that item Y will appear in a commit operation provided item X appears in that commit operation. It is calculated

by using the following formula:

$$Confidence(X = Y) = \frac{Support(X = Y)}{Support(X)} = \frac{C_{XY}}{C_X}$$

Where C_X is the total number of commits in which item X appeared independently or in association with other items and C_{XY} is the total number of commits in the change history in which both X and Y appeared together. For each pair of the software elements, we calculate both support and confidence measure and later using min_support and min_confidence threshold values, we created the co-change graph G = (V, E) where V is the set of all software elements and E is the set of edges between them. In our proposed approach, the co-change graph is used as an indication of co-change coupling. At any instant of time during each iteration of NSGA-III algorithm, different components are identified from the chromosome representation. Such identified components are dynamic in nature (software elements belonging to each component) and changes from iteration to iteration of the NSGA-III algorithm. In this scenario, the overall co-change coupling for a component is dynamically computed using the previously obtained co-change MDG using the *S*upport and *C*onfidence metric. Here, the overall co-change coupling of a component is computed as the average of the coupling strength denoted in the co-change graph. Here, edge strength present in the MDG between the different pair of software elements belonging to a component is used. Here, averaging of the pairwise coupling strength is simply chosen to have a coupling at the component level.

4.4. NSGA-III Based Optimized Clustering

This subsection of the paper describes how the different similarity measurements, which are already proposed in the above subsections, are used to perform optimized clustering. As the optimization is guided by three objective functions, therefore, in each iteration, the fitness of different solutions are evaluated based on these objective functions. The chromosome representation of different solution determines the association of different software elements with different clusters/ components. Based on this association information, the values of different metrics used to evaluate the fitness of a solution is computed. Initially, these metrics are computed at the component level, as indicated by chromosome representation, and then the fitness values for the solution is obtained by averaging the values already computed

at the component level. Now, based on the fitness values of different solutions, a new population is generated by using a crossover, mutation, and non-dominance sorting in NSGA-III. Since the chromosome representations of the different solution in each iteration are different, so the computation of the different objective functions is dynamic in nature due to changing association of software elements with different clusters. Moreover, NSGA-III based clustering aims at partitioning different software elements into various clusters without altering the internal structure of the software system in any way. Thus, after clustering, the software will still syntactically and semantically works just similar to its working prior to clustering. The NSGA-III based optimized clustering ultimately returns the set of reusable components in the form of obtained cohesive clusters. Different obtained clusters are finally counted as the total number of obtained reusable components.

5. Experimental Setup and Statistical Analysis

This section describes the details about the experimentation carried out in order to evaluate and validate our proposed approach. The experiment is conducted in order to identify a set of reusable components that can be stored in the component library. This component library can be reused in new software development using CBSE approach. This section of the paper first discusses about various parameters used with NSGA-III algorithm and how they are initialized. It also describes different software systems considered for experimentation and proposed approach evaluation. This section also provides details about the statistical analysis techniques used to validate the obtained results. The experiment is motivated by the fact that the FUPs present, semantic relatedness and co-change information present in a software system reflects the dependencies among different software elements and thus can be simultaneously used to perform optimized clustering in order to extract reusable software components. Various research questions formulated to evaluate and justify the proposed approach are also specified. The details of the experimentation and other factors are discussed in the following subsections:

5.1. Multi-Objective Algorithm used and Different parameters Used

The proposed approach of this paper uses the Non-Dominated Sorting Algorithm (NSGA-III) [55], in order to perform clustering and to finally obtain the reusable components as different clusters. It is a genetic-based meta-

heuristic algorithm which is generally applied as a Multi-Objective evolutionary algorithm in order to solve different optimization problems. In literature, NSGA-III is mainly used to perform software remodularization and clustering [12, 56, 57]. Proper parameter setting tuning plays an important role in the accuracy of the implemented algorithm. In our proposed approach, the NSGA-III algorithm uses single-point crossover, uniform mutation operation, and binary tournament selection approach to solve the RSCI problem. The crossover probability is set at 90% for the problem instance having less than 100 classes and is set at 100% otherwise. For the mutation probability, it is set 0.04 * log 2 (N), with N number of classes for all types of the problem instance. Initial population size used is 1000 and the total number of iterations used is 30000. These parameters and their corresponding values are commonly being used in the literature [56, 12] to perform clustering. Therefore, in this paper, we have also chosen the same values.

5.2. Analyzed Systems

The evaluation of the proposed approach is carried out on six real-world open-source software systems. Table-1 presents the summary of the studied systems for the experimentation. Here, open-source software systems of different sizes and belonging to different domains are considered for evaluation. The selected subject systems are of high quality and are commonly usable among researchers and the open-source community. Different columns of the table-1 represent the subject systems name, version, the total number of classes, the total number of modules present in the corresponding jar file and the total number of connections present in the system among different software elements. All these different characteristics of different subject systems are determined using a well-known structural analysis tool called PF-CDA.

5.3. Collecting Results from the experimentation

In our proposed approach, the NSGA-III algorithm is run 30 times with the above configuration setup in order to obtain the final result. The final result represents the clustering of different software elements and each obtained cluster represents an obtained reusable software component. Each run cycle of the experiment gives a set of Pareto-front. As the obtained Pareto-front is evaluated using TurboMQ metric so the solution giving the highest value for the said metric is recorded for each run cycle. Finally, the mean and standard deviation of TurboMQ is estimated using the result from 30 running cycles.

Table 1: Summary of the software systems under study.

S. No.	System Name	Version	# Classes	# Modules	# Connections
1	Apache Commons Email	1.5	20	3	98
2	JUnit	4.5	131	8	432
3	HTTP Components	4.5.5	145	16	642
4	JavaCC	1.5	154	6	722
5	JDOM	1.1.3	62	6	128
6	Apache Commons Logging	1.2	14	2	65

5.4. Statistical Tests Performed

Since metaheuristic algorithms are stochastic optimizers, so, they can provide different results for the same problem instance from one run to another. Therefore, 30 independent runs of the NSGA-III algorithm are performed to obtain results and during the statistical test, there mean value is used. The obtained results after experimentation are statistically analyzed using the ANOVA and two-tailed t-tests with 95% confidence level ($\alpha = 5\%$). ANOVA test is used to investigate the output values obtained as a result of a different independent run of the algorithm. In this paper, the two-tailed t-test is used to compare our proposed approach with six other approaches formulated as discussed in the subsection below. The result of both these tests is finally presented in section-6 of this paper. In this paper, the used two statistical tests are used to validate the following two hypothesis:

H0: the obtained reusable software components over the different run of the NSGA-III algorithm are samples from a distribution with an equal median.

H1: the obtained results of two different scenarios (one proposed and the other formulated by considering different relations as individual/ pairwise) are not samples from the distributions with an equal median.

The p-value of the considered statistical test in this paper represents the probability of rejecting the null hypothesis. The cases where the p-value is equal to or less than to $\alpha(\leq 0.05)$ represent that we are accepting the alternative hypothesis and rejecting the null hypothesis. However, the achieved

p-value that is strictly greater than $\alpha(> 0.05)$ denotes that we are rejecting the alternative hypothesis and accepting the null hypothesis. ANOVA test is used to validate the H0 hypothesis. Here, if the p-value is < 0.05 then it signifies that all the values obtained are statistically significant, otherwise, there isat least one significant value that is different from other values. Similarly, a p-value of two-tailed t-test is used to validate the H1 hypothesis. Here, if the p-value is > 0.05 then it signifies that the two considered scenarios are statistically different and they are not from the same distribution.

5.5. Research Questions

In our paper, we proposed an approach to identify reusable software components from OO source-code of a software system. The proposed approach makes use of three kinds of relations among different software elements viz FUP, semantic and co-change relations, and models the component identification problem as a search-based problem. Finally, it utilizes the NSGA-III algorithm to identify different components. Following research questions are designed as part of our study and they are answered in the section- 6 of this paper:

RQ1. **How well the proposed approach performs in identifying reusable software components?** In this paper, the performance of the proposed approach is evaluated using a well-known TurboMQ metric, which measures the balance between cohesion and coupling among different identified software components available as separate groups. This research question is formulated to show how well the proposed approach performs as compared to other similar approach proposed in the literature. Here, two approaches are compared by matching the total number of obtained reusable software components.

RQ2. **What is the effect of considering a different kind of relations viz FUP, semantic and co-change relations as an individual, pairwise and all together?** As the proposed approach in this paper considers the combined use of all the three kinds of relations. Therefore, this paper also formulates and finally evaluate different cases by considering different relations as individual and pairwise against our proposed approach. This helps in deciding whether there are any redundant or obsolete relations present in the proposed approach.

RQ3. **How capable is the proposed approach to separate the cohesive elements from a group of multiple software elements?**

Since the proposed approach aims at grouping related software elements in a single group and thus is capable of distinguishing between related and non-related software elements. This research question is formulated to justify the capability of the proposed approach in grouping all those classes as a single component, which are connected to each other due to related services or functional requirements.

5.6. Evaluation Criteria

To validate our proposed search-based reusable software component identification approach, popular modularization quality (MQ) fitness function, known as TurboMQ, is used [58]. TurboMQ tries to measure the optimum balance between inter-connectivity and intra-connectivity among different software elements in a given partition structure of the system under study. Here, inter-connectivity (coupling) is defined as a measure of interaction among different software elements belonging to different identified software components. Similarly, intra-connectivity (cohesion) is defined as a measure of interaction among different software elements belonging to a component with each other. This metric is chosen to evaluate our proposed approach because it provides a measurement of the optimum balance between cohesion and coupling. A higher score for TurboMQ indicates that the underlying system (identified as the collection of different reusable components identified) possesses higher cohesion along with low coupling and vice-versa. Thus, a higher score for a software system represented as a collection of the distinct individual identified components represent that our approach works perfectly in grouping dependent and related software elements as a component.
Moreover, because this paper considers three relations among different software elements namely FUP, semantic relatedness, and co-change. Therefore, we have also investigated the effectiveness of different considered three relations towards RSCI. In this paper, we further considered following six more variations/ different combination of the above three factors: 1) FUP; 2) semantic relatedness (SR); 3) co-change (CC); 4) FUP and semantic relatedness (FUP + SR); 5) semantic relatedness and co-change (SR + CC) and 6) FUP and co-change (FUP + CC) in order to investigate their role. These six formulated approaches are further compared with one that makes use of all three, namely (FUP + SR + CC).

6. Results

This section of the paper presents the detailed analysis of the results obtained during the experimental evaluation of the proposed RSCI for the different software systems. We have also provided the results of the comparative study performed of the proposed approach against six different formulated scenarios. The statistical analysis results are also provided and their significance is discussed in detail. This section also discusses and analysis various research question formulated in section- 5.

6.1. ***RQ1. How well the proposed approach performs in identifying reusable software components?***

This research question is answered by applying the proposed approach on different software systems under study and then evaluating the results by calculating TurboMQ metric value. Table-2 shows the mean and standard deviation values for TurboMQ metric for different software systems under study. In the table, the third column gives a total number of reusable components identified for different software systems under study. Column fourth gives the TurboMQ score of the original software system and finally column fifth, sixth and seventh gives mean, standard deviation (std. dev.) and percentage improvement in TurboMQ values after performing the experimentation. These factors are computed because the search-based evolutionary algorithm (NSGA-III) is used to perform the proposed experimentation. Column eight of the table gives p-values obtained after applying ANOVA-test on obtained TurboMQ values. The obtained p-value is less than the confidence interval (i.e. 0.05) for different software systems and it strongly indicates that obtained TurboMQ values during the different run are not statistically different from one another. Thus, no single obtained value is dominating the obtained result. The *B*efore column denotes the TurboMQ metric score based on the original modular architecture of the system under study. Similarly, the *A*fter column denotes the TurboMQ metric score after performing clustering using NSGA-III algorithm and in which different clusters represents a new improved modular architecture of the system. From the data in % change column, it is clear that after clustering using the proposed approach, the interdependency between modules (components) in our case is significantly reduced. Hence, the obtained components are more reusable due to fewer dependencies on other components.

Figure-4 shows the plot depicting the TurboMQ score of different software

Table 2: Mean and Standard Deviation of TurboMQ scores of different Software Systems under study.

S. No.	System	# Reusable Components	TurboMQ Score				ANOVA-test
			Before	After			
				Mean	Std. Dev.	% Change	
1	Apache Commons Email	5	1.65	2.31	0.126	28.57	< 0.0011
2	JUnit	24	1.25	2.05	0.172	39.02	< 0.0010
3	HTTP Components	32	1.87	2.08	0.238	10.09	< 0.0023
4	JavaCC	38	2.42	3.02	0.175	19.87	< 0.0010
5	JDOM	10	1.87	2.14	0.199	12.61	< 0.0020
6	Apache Commons Logging	4	0.72	1.08	0.185	33.33	< 0.0010

systems under study before and after the experimentation. Before experimentation, the original structure of the software system is used for TurboMQ evaluation and after experimentation; a collection of different reusable software components constitutes the underlying structure of the software system. From the plot, it is clear that the quality of the different systems under study increases by 10% to 39%. Here, the increase in TurboMQ score is an indication of the improved balance between cohesion and coupling among the identified reusable software components. The improvement is due to the fact that after grouping different software elements using the proposed approach, the inter dependency among different components is reduced significantly because the clustered elements are functionally related to a large extent. Hence, this improvement is an indication of an increase in the reusability of the identified component. Because a component is always a standalone entity with least dependencies with other components. Moreover, the cohesion in the proposed approach is computed based on member variables usage patterns and the semantic relatedness, therefore, the clustered software elements are functionally related also. Hence, clustered software elements are more likely to be called components because they are now (after clustering using the

proposed approach) more cohesive and least coupled.
This section of the paper also compares the results of the proposed approach with the approach proposed by [31]. The proposed approach by [31] aims at converting an OO API into the corresponding component based API having more understandability and reusability. The proposed approach by [31] focuses on grouping the underlying classes based on the interaction of the API with different considered application clients and their ability to form a quality-centric component. They have applied their proposed approach on four standard Android APIs namely android.view (handles creation and management of user interfaces), android.app (handles creation and management of android applications), java.util (contains utility classes), and android API (handles underlying platform and defines application permissions). For the sake of a fair comparison, this paper also considers the same set of APIs. Table-3 summarizes the obtained results. The second and third column of the table gives details about the studied APIs and their sizes in terms of a total number of classes and interfaces present. Similarly, the fourth and fifth column specifies the total number of components identified by [31] and our proposed approach of this paper. Both the approaches relies on clustering for grouping the underlying software elements that are together cohesive. Such groups are finally termed as reusable components. The last two columns of the Table-3 shows the TurboMQ score values for both the approaches. These values are used to compare the quality of the obtained reusable components. The higher values in our proposed approach signifies that the components are more cohesive as compared to [31]. Hence, the obtained components are more reusable. Moreover, when the experiment is carried out on large systems, it is observed that our proposed approach is equally scalable and the use of the NSGA-III help us to overcome the exponential time growth complexity linked with size of the software system.

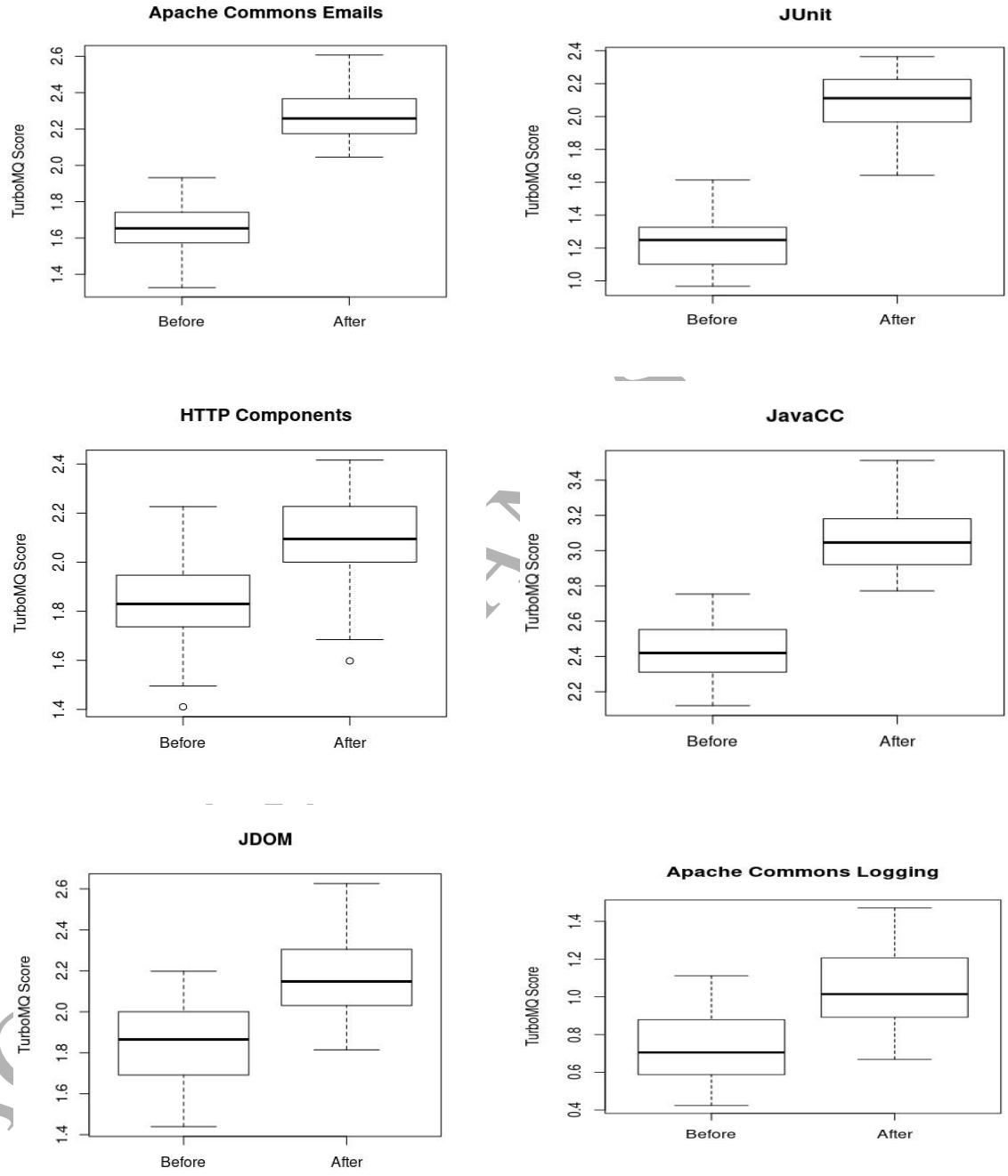


Figure 4: TurboMQ Score Comparison of different systems Before and After Component Identification using Boxplot.

Table 3: Comparison of Proposed Approach V/S Shatnawi et al. Approach [31].

S. No.	Software Name	# Classes	# Reusable Components Identified		TurboMQ Score Comparison	
			Shatnawi et al. [31]	Our Proposed Approach	Shatnawi et al. [31]	Our Proposed Approach
1.	Android.View	491	43	53	1.45	1.53
2.	Android.App	361	55	62	0.89	1.03
3.	Java.Util	846	147	163	2.15	2.23
4.	Android	5790	497	508	2.54	2.61

Figure-5 shows the comparison using a line plot. It compares the results of table-3. From the plot, it is clear that our proposed approach is capable of identifying more reusable components from the same software set as compared to other approach being compared. In fact, our approach is capable of identifying a minimum 10% more reusable components as compared to another approach. The enhanced results are because our proposed approach is free from dependencies to appropriate selection of client applications. However, in the proposed approach of [31], appropriate selection of client applications is necessary to obtain high-quality results. This is because the interaction between API and the selected client applications is the basis of their approach. This restriction is not applicable to our proposed approach.

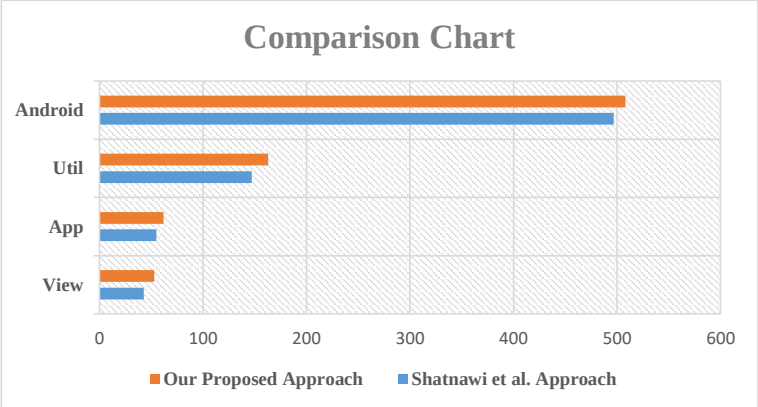


Figure 5: Comparison of Our Proposed Approach v/s Shatnawi et al.[31] Approach.

6.2. *RQ2. What is the effect of considering a different kind of relations viz FUP, semantic and co-change relations as an individual, pairwise and all together?*

Since, in this paper, our proposed approach that makes combined use of three relations (FUP + SR + CC) is compared with the other six formulated approaches as already discussed in section-5 of this paper. Therefore, table-4 gives details about the obtained results after experimenting and evaluating different systems under different formulated scenarios. In the table, the mean TurboMQ score under different considered and proposed schemes are given in third to ninth columns. Column tenth of the table gives results about the two-tailed t-test performed for statistical analysis. The t-test compares the results of our proposed approach (FUP + SR + CC) with six other approaches one by one and the results are presented in terms of significance score. In every case, the obtained p-value is ≥ 0.05 and thus indicates that the values of two considered cases for comparison purpose are not from the same normal distribution. The significance score of t-test performed is presented using three symbols viz +, -, and =. The symbol + indicates that our proposed approach shows significant improvement and the symbol = indicates that the proposed approach works equally with other considered approach. Finally, the symbol - is considered to indicate a situation where our proposed approach is underperforming as compared to other considered approaches.

Since t-test requires that input data must be normally distributed. There-

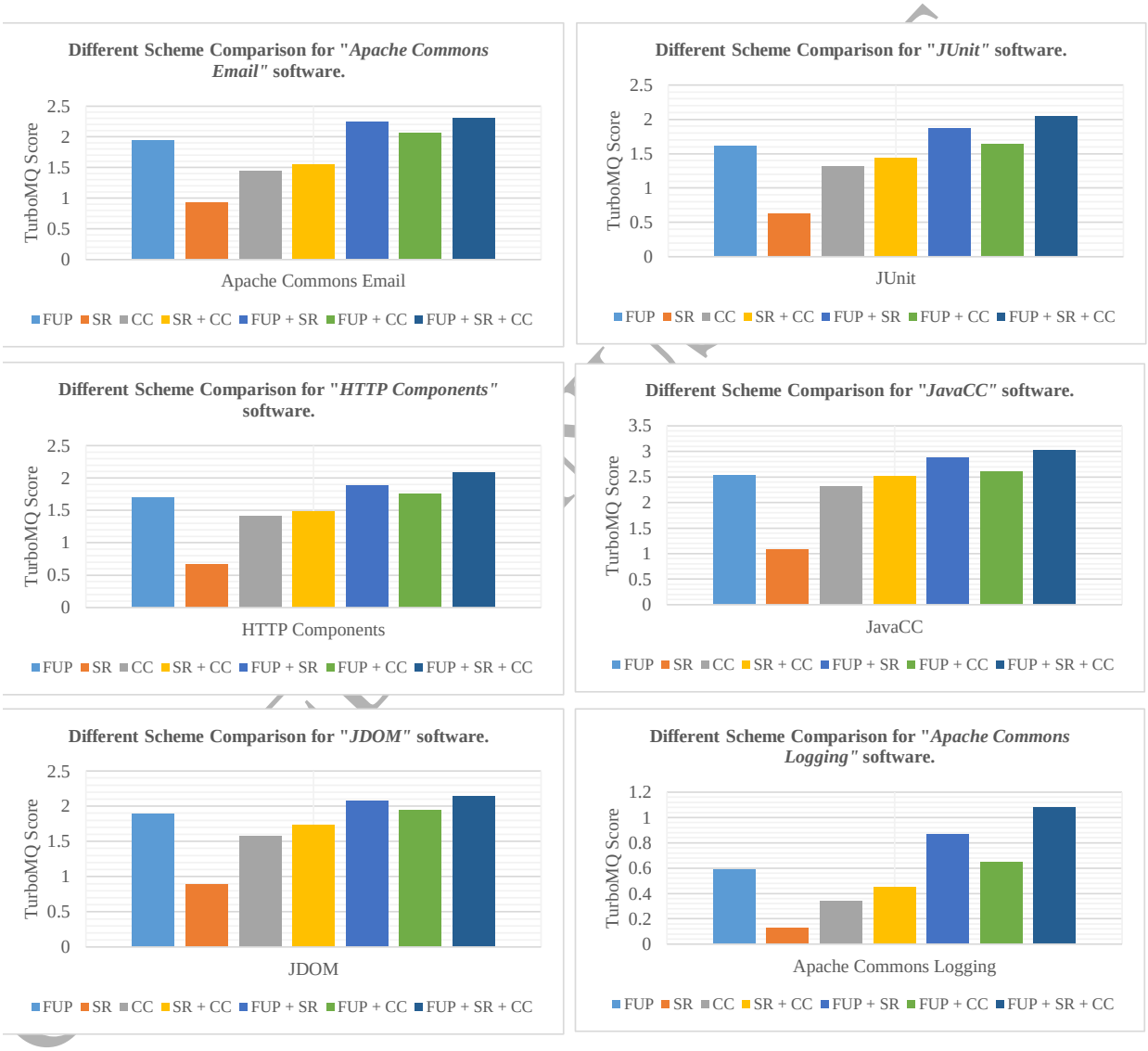


Figure 6: Comparison chart of different studied schemes for different software systems.

Table 4: Comparison of different Schemes viz FUP+SR+CC, FUP+SR, and FUP+CC.

S. No.	System	Mean TurboMQ Score under different Schemes							t-tests Significance	Shapiro Wilk test
		FUP	SR	CC	SR + CC	FUP + SR	FUP + CC	FUP + SR + CC		p-Value
1	Apache Commons Email	1.95	0.93	1.45	1.55	2.25	2.06	2.31	++++=+	0.511
2	JUnit	1.61	0.62	1.32	1.43	1.87	1.64	2.05	++++++	0.239
3	HTTP Components	1.69	0.66	1.41	1.49	1.89	1.76	2.08	++++++	0.123
4	JavaCC	2.53	1.07	2.31	2.52	2.88	2.61	3.02	++++++	0.098
5	JDOM	1.89	0.89	1.57	1.73	2.08	1.94	2.14	++++=+	0.486
6	Apache Commons Logging	0.59	0.13	0.34	0.45	0.87	0.65	1.08	++++++	0.249

fore, we used the Shapiro-Wilk test to determine the normal distribution of the data [59]. The ShapiroWilk test is a popular test used to check the normality in statistics. The null hypothesis of this test assumes that the input data is normally distributed. During the Shapiro-Wilk test, if the obtained p-value of the test is less than the considered significance level (taken as 5% in this case), then the null hypothesis is rejected and vice-versa. The obtained TurboMQ score under different dependency schemes (seven schemes) is considered as input and Shapiro-Wilk test is applied. The obtained p-value is recorded and is shown in column eleven of the table-3. The obtained p-values are greater than the considered significance level (0.05 or 5%), so, we cannot simply reject the null hypothesis. Hence, it is confirmed that the TurboMQ scores that are taken as input to the t-test are normally distributed. From the obtained results of the t-test significance score in table-4, it is clear that our proposed approach significantly gives a better improvement in results as against various considered schemes namely FUP, CC, SR, SR + CC, and FUP + CC. On the other hand, our proposed approach gives comparable results as against the considered scheme namely FUP + SR. In this case,

our proposed approach shows significant improvement in four out of total six cases considered and in the rest of the two cases, it shows comparable results. In any of the compared cases, our proposed approach never underperforms as indicated by the obtained t-test significance values.
Figure-6 shows a comparison chart of an obtained TurboMQ score of different software systems under different evaluation schemes specially constituted for our RSCI problem. Each chart shows achieved a TurboMQ score for seven different evaluation schemes (one including our proposed FUP + SR + CC approach). From the plots, it is clear that our proposed schemes outperform every other scheme constituted for evaluation of the proposed scheme. Therefore, while identifying reusable software components, considering all three kinds of relations viz FUP, semantic and co-change is necessary. Ignoring any one of these relations ultimately decreases the quality of the identified components measured as a balance between cohesion and coupling. Moreover, it is also clear that the FUP relations between different software elements play a major role in the overall identification of reusable software components and both semantic relatedness and co-change further enhance the quality.

6.3. ***RQ3: How capable the proposed approach is to separate the cohesive elements from a group of multiple software elements?***

This research question aims at justifying the ability of the proposed approach in separating related and unrelated functionality (classes). This justification is very important in the identification of reusable software components as unrelated classes belong to separate components. To perform this experiment, the paper mixes various elements (classes) belonging to two or more software systems and then perform clustering using the proposed approach. The obtained clusters are examined to determine the extent to which they contain classes belonging to a single software system.
For this purpose, various accuracy metrics already proposed in [60] are utilized. The accuracy metric set considered in this paper consist of three well known Information Retrieval (IR) metrics called precision, recall, and F-Measures. These metrics are based on the measurement of True Positive (TP), False Positive (FP), and False Negative (FN) cases [60]. In the obtained clustering results, the clusters may contain classes belonging to a single software system or they may contain classes belonging to two or more mixed software systems. In our proposed approach, each class belonging to a cluster is assigned a number that identifies to which software system it

Table 5: Different software groups considered for measuring precision, recall, and F-Measure.

Group No.	S. No.	Studied Software System	# Classes	Brief Description
G1	1	Apache.Commons. Emails 1.5	20	Apache API for sending emails
	2	Apache.Commons. Logging 1.2	14	Apache API useful to log information
G2	1	Easymock 2.4	65	A tool for test-driven development
	2	Junit 4.5	27	A unit testing framework for Java programming language
G3	1	JHotDraw 7.5.1	65	2-D graphics framework
	2	JEdit 4.5.0	35	Java text editor
	3	Apache Ant 1.9.2	67	Java library and command-line tool for building software

belongs. The assigned number varies from 1 to N for different clusters and it uniquely identifies a software system in a mixed group. Here, N is the total number of software systems mixed. All the classes in a cluster are assigned number equal to the software systems unique number to which the majority of classes in a cluster belongs. This can be more clearly understood by considering a cluster having 6 classes belonging to 3 software say A, B, and C. further suppose that this cluster contains one class from A and one class from B and 4 classes from system C. Then, all the 6 classes in this cluster are assigned unique number of system C. Assigning numbers to different classes helps us to identify TP and/ or FP and/ or FN cases.

This research question is answered based on the obtained results of expert criteria approach in which the accuracy is measured using three well-known Information Retrieval (IR) metrics called precision, recall and F-Measures [60]. The expert criteria approach focuses on measuring the accuracy of the obtained results against ground-truth architecture (aka Gold Standard of underlying architecture). In our experimentation, we randomly mixed the classes of two or more standard software systems. Table- 5 summarizes about different software systems considered for experimentation. During ex-

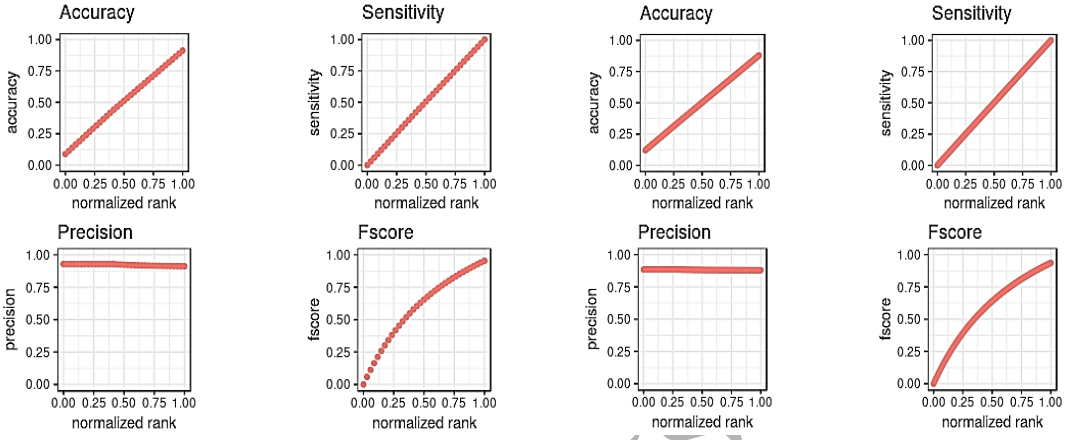
Table 6: Precision, Recall and F-Measure Scores for studied Systems.

Group No.	S. No.	System Name	Precision	Recall	F-Measure
G1	1	Apache.Commons.Emails	95%	92%	94%
	2	Apache.Commons.Logging	92%	89%	91%
G2	1	Easymock	91%	80%	85%
	2	Junit	94%	88%	90%
G3	1	JHotDraw	85%	89%	86%
	2	JEdit	84%	79%	81%
	3	Apache Ant	89%	81%	85%

perimentation, three groups are formulated containing two/ three different software systems and then these groups are evaluated using our proposed approach to have a new clustering of the mixed system. The obtained clustering results are compared with a gold standard and different metrics are computed [61, 34, 62, 63, 64]. The gold standard for any software represents the authority partition as identified by specialized developers/ maintainers. Moreover, we can also deploy software engineers having deep knowledge of the software system. While determining the underlying gold standard (authoritative partition), it is especially taken care of that the developers/ maintainers and/ or software engineers are unaware about our formulated research questions in order to minimize any chances of biasing. Table-6 shows the obtained results of the expert criteria approach.

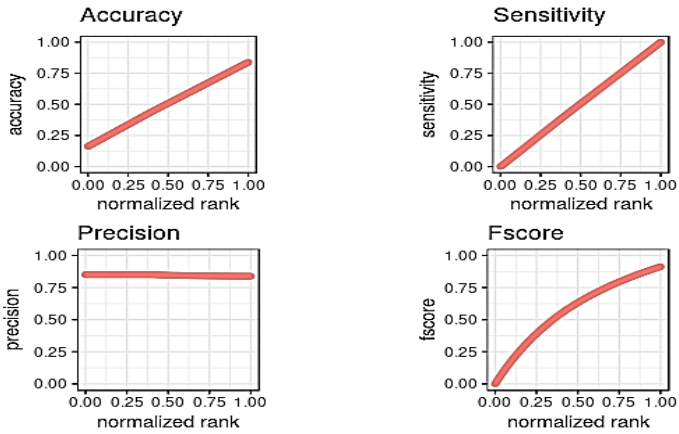
The F-Measure score here is significantly higher ($> 80\%$) and it signifies the ability of our proposed scheme in separating the related and unrelated elements by performing clustering. Thus, the proposed approach is capable of accurately modeling the actual coupling relations among different software elements present in a studied system. From the software component identification point of view, it is very important that the proposed approach is capable of separating functionally related and unrelated elements into separate groups because a reusable software component is a collection of functionally related elements only [65]. Our proposed approach stands firm in this and is capable of separating related and unrelated functionality apart in different groups. Since the studied different software systems considered in different groups are somewhat semantically related but still, our approach is capable of distinguishing them as separate to a good extent.

Figure-7 also shows plots for the obtained results in table-6 for different



(A). Different Curves for group G1 software systems.

(B). Different Curves for group G2 software systems.



(C). Different Curves for group G3 software systems.

Figure 7: Accuracy, Sensitivity (Recall), Precision & F-Measure Curve plots of different groups under study.

software groups under study using four criteria, namely Accuracy, Sensitivity (Recall), Precision and F-Measure. From the plots, it is clear that our proposed approach significantly performs as indicated by the Accuracy curve for each of the different software groups under study. The higher value for the accuracy curve in Figure-7 reflects the ability of our proposed approach in separating related and unrelated software elements separately considered under different groups. Similarly, the higher values for different Sensitivity, Precision, and F-Measure curves also reflects the ability of our proposed approach in differentiating related and unrelated software elements separately.

Based on the experimental results and the analysis done in this section, we can conclude the following:

1. Based on RQ1, the proposed approach can be very useful in identifying reusable software components and these components will be highly cohesive as well as least coupled.
2. Based on RQ2, we conclude that all of these three kinds of dependency relations i.e. FUP, semantic, and co-change, are necessary to represent the actual dependency relations, and none of these three can be ignored.
3. Based on RQ3, it is concluded that the proposed approach is capable of distinguishing between functionally related and unrelated software elements. Moreover, the proposed approach is capable of grouping functionally related elements together in a single group (aka component).

7. Threats To Validity

This section of the paper provides an explanation of various key threats to our proposed approach that could affect the obtained results. The three main threats identified for our proposed approach are as follows:
First, as our proposed approach is based on three kinds of dependence relations namely FUP, semantic and co-change. Out of these relations, FUP and semantic relations are identified from source-code of a software system and their authenticity is highly dependent on the proper design methodologies. Similarly, the correctness of the co-change relations is highly dependent on the accuracy of the maintained change logs. Inconsistencies in the maintenance of any of these three relations can significantly affect the performance of the proposed approach. Therefore, it is another threat to our proposed approach. However, this threat in the present study is minimized by considering the software system that has a high standard of design architecture and is being actively maintained continuously by a team of experts. Moreover, the selected software systems have been actively used by different researchers in literature to validate their respective proposed approach.
Secondly, the TurboMQ metric is used to access the reusability of the obtained components. This paper considers any obtained component to be more usable that possess higher underlying TurboMQ metric value. No specific threshold for the TurboMQ metric value are tested in this paper. Instead, the underlying NSGA-III algorithm used is ultimately responsible for identifying such components having highest TurboMQ value through different iterations. The TurboMQ never guarantee the best indicator of quality

and hence is a potential threat to the validity.
Thirdly, the ground-truth architecture of the studied system is being used to evaluate our proposed approach. However, sometimes there is not always a single unique definition of the ground-truth architecture [66]. Identifying correct ground-truth architecture requires heavy manual efforts from a team of experts. Therefore, it is always a challenging task to certify a particular obtained ground-truth architecture of a software system. Since we have used is to validate our proposed approach in RQ3. Therefore, identifying and certifying used ground-truth architecture is also a threat to our proposed approach. However, to reduce the effect of this threat, different committees of domain experts are formulated and finally after mutual concession the actual ground-truth architecture for a software system is decided.

8. Conclusion and Future Work

The CBSD being the growing trend in the software industry, which aims at minimizing cost, maintenance efforts and bugs by using high-quality components during software development. So, in this paper, an approach to identify the set of high-quality components is proposed from a given object-oriented software system.
The proposed approach is based on modeling the reusable software component identification problem as a search-based multi-objective optimization problem. In the proposed approach, three objective functions are used, namely FUP based cohesion maximization, semantic relatedness based cohesion maximization and co-change based coupling based minimization. The FUP set of a software element is used to measure its underlying cohesion using a newly proposed *PatternCohesion metric*. The *PatternCohesion metric* metric helps in estimating the cohesiveness of a given software element and its ability to be called as a software component. The semantic relatedness is again used to measure cohesion among different software elements belonging to the underlying software system. The semantic information is obtained by tokenizing the OO source-code of different software elements and getting the information from three main parts viz class declaration statement, data types of member variables and function signatures. The co-change coupling is calculated by extracting information from the change-history of the software system available in GIT/SVN repository. For grouping different softwareelements together, we have used the NSGA-III search-based algorithm. The identified software components using this approach are highly cohesive

and least coupled. Further, such components also have the characteristics that they provide a single or only a few related functionalities to the outside world. The proposed approach is evaluated over six open source OO software systems and our proposed approach stands firm on both optimized cohesion and coupling characteristics among different software elements belonging to a component. The proposed approach returns good quality components, which can be directly used in CBSD environment.

This future work regarding the approach proposed in this paper is related to use the frequent usage patterns in the field of architecture recovery. As we have formulated the crossover rate and mutation probability to be dependent on a total number of classes in a software system, therefore, an investigation can be further carried out to determine the need of such dependency. Moreover, the different clustering algorithm can be further evaluated and tested against the proposed clustering algorithm. Further, as the obtained values in the sub-section 6.3 varies in the 80% to 90% range and denotes set of classes that are misidentified. This set needs to be further investigated to determine any kind of possibilities of sharing of some common characteristics or to hypothesize on some reason for the misidentification.

References

[1] X. Wang, L. Wang, Software reuse and distributed object technology, in: 2011 Fourth International Joint Conference on Computational Sciences and Optimization, 2011, pp. 804–807.

[2] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] D. Birkmeier, S. Overhage, On component identification approaches – classification, state of the art, and comparison, in: G. A. Lewis, I. Poernomo, C. Hofmeister (Eds.), Component-Based Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–18.

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

[5] Z.-g. Cai, X.-h. Yang, X.-y. Wang, A. J. Kavs, Afuzzy formal concept analysis based approach for business component identification, Journal of Zhejiang University SCIENCE C 12 (9) (2011) 707.

[6] S. Hasheminejad, S. Jalili, Ccic: Clustering analysis classes to identify software components, Information and Software Technology 57 (2015) 329 – 351.

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

[8] S. Kebir, A. D. Seriai, S. Chardigny, A. Chaoui, 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, 2012, pp. 181–190.

[9] S. Allier, S. Sadou, H. Sahraoui, R. Fleurquin, From object-oriented applications to component-oriented applications via component-oriented architecture, in: 2011 Ninth Working IEEE/IFIP Conference on Software Architecture, 2011, pp. 214–223.

[10] Z. Alshara, A.-D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, A. Shatnawi, Migrating large object-oriented applications into component-based ones: Instantiation and inheritance transformation, SIGPLAN Not. 51 (3) (2015) 55–64.

[11] O. Nierstrasz, L. Dami, Component-oriented software technology, in: O. Nierstrasz, D. Tsichritzis (Eds.), Object-Oriented Software Composition, Prentice Hall, 1995, pp. 3–28.

[12] M. d. O. Barros, An analysis of the effects of composite objectives in multiobjective software module clustering, in: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO '12, ACM, New York, NY, USA, 2012, pp. 1205–1212.

[13] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multi-objective search problem, IEEE Transactions on Software Engineering 37 (2) (2011) 264–282.

[14] Amarjeet, J. K. Chhabra, Improving package structure of object-oriented software using multi-objective optimization and weighted class connections, Journal of King Saud University - Computer and Information Sciences 29 (3) (2017) 349 – 364.

[15] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proceedings - Software 150 (3) (2003) 161–175.

[16] H. Washizaki, Y. Fukazawa, A technique for automatic component extraction from object-oriented programs by refactoring, Science of Computer Programming 56 (1) (2005) 99 – 116, new Software Composition Concepts.

[17] A. Marx, F. Beck, S. Diehl, Computer-aided extraction of software components, in: 2010 17th Working Conference on Reverse Engineering, 2010, pp. 183–192.

[18] X. Wang, X. Yang, J. Sun, Z. Cai, A new approach of component identification based on weighted connectivity strength metrics, Information Technology Journal 7 (1) (2008) 56 – 62.

[19] E. Lee, B. Lee, W. Shin, C. Wu, Toward Component-Based System: Using Static Metrics and Relationships in Object-Oriented System, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 87–101.

[20] S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit, Extraction of component-based architecture from object-oriented systems, in: Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), 2008, pp. 285–288.

[21] A. Shatnawi, A. D. Seriai, Mining reusable software components from object-oriented source code of a set of similar software, in: 2013 IEEE 14th International Conference on Information Reuse Integration (IRI), 2013, pp. 193–200.

[22] D. Quinlan, Q. Yi, G. Kumfert, T. Dahlgren, M. Schordan, Toward the automated generation of components from existing source code, in: In Second Workshop on Productivity in High-End Computing, 2005.

[23] J. A. Kim, O.-C. Kwon, J. Lee, G.-S. Shin, Component adaptation using adaptation pattern components, in: 2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236), Vol. 2, 2001, pp. 1025–1029 vol.2.

[24] S. D. Kim, S. H. Chang, A systematic method to identify software components, in: 11th Asia-Pacific Software Engineering Conference, 2004, pp. 538–545.

[25] S. Allier, H. A. Sahraoui, S. Sadou, Identifying components in object-oriented programs using dynamic analysis and clustering, in: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '09, IBM Corp., Riverton, NJ, USA, 2009, pp. 136–148.

[26] S. Allier, H. A. Sahraoui, S. Sadou, S. Vaucher, Restructuring Object-Oriented Applications into Component-Oriented Applications by Using Consistency with Execution Traces, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 216–231.

[27] H. S. Hamza, A framework for identifying reusable software components using formal concept analysis, in: 2009 Sixth International Conference on Information Technology: New Generations, 2009, pp. 813–818.

[28] A. Shatnawi, A. Seriai, H. Sahraoui, Z. Al-Shara, Mining Software Components from Object-Oriented APIs, Springer International Publishing, Cham, 2014, pp. 330–347.

[29] S. Li, L. Tahvildari, Jcomp: A reuse-driven componentization framework for java applications, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 264–267.

[30] A. Seriai, S. Sadou, H. A. Sahraoui, Enactment of Components Extracted from an Object-Oriented Application, Springer International Publishing, Cham, 2014, pp. 234–249.

[31] A. Shatnawi, A.-D. Seriai, H. Sahraoui, Z. Alshara, Reverse engineering reusable software components from object-oriented apis, Journal of Systems and Software 131 (2017) 442 – 460.

[32] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, E. R. Gansner, Using automatic clustering to produce high-level system organizations of source code, in: Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on, 1998, pp. 45–52.

[33] S. Mancoridis, B. S. Mitchell, Y. Chen, E. R. Gansner, Bunch: a clustering tool for the recovery and maintenance of software system structures, in: Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on, 1999, pp. 50–59.

[34] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, R. Oliveto, Putting the developer in-the-loop: An interactive ga for software remodularization, in: G. Fraser, J. Teixeira de Souza (Eds.), Search Based Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 75–89.

[35] J. Brest, S. Greiner, B. Boskovic, M. Mernik, V. Zumer, Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems, Trans. Evol. Comp 10 (6) (2006) 646–657.

[36] M. Črepinšek, S.-H. Liu, M. Mernik, Exploration and exploitation in evolutionary algorithms: A survey, ACM Comput. Surv. 45 (3) (2013) 35:1–35:33.

[37] J. Lundberg, W. Lwe, Architecture recovery by semi-automatic component identification, Electronic Notes in Theoretical Computer Science 82 (5) (2003) 98 – 114, sC 2003, Workshop on Software Composition (Satellite Event for ETAPS 2003).

[38] K. Deb, H. Jain, Handling many-objective problems using an improved nsga-ii procedure, in: 2012 IEEE Congress on Evolutionary Computation, 2012, pp. 1–8.

[39] D. E. Kvasov, M. S. Mukhametzhanov, Metaheuristic vs. deterministic global optimization algorithms: The univariate case, Applied Mathematics and Computation 318 (2018) 245 – 259, recent Trends in Numerical Computations: Theory and Algorithms.

[40] A. A. Juan, J. Faulin, S. E. Grasman, M. Rabe, G. Figueira, A review of simheuristics: Extending metaheuristics to deal with stochastic combinatorial optimization problems, Operations Research Perspectives 2 (2015) 62 – 72.

[41] A. Rathee, J. K. Chhabra, Improving cohesion of a software system by performing usage pattern based clustering, Procedia Computer Science 125 (2018) 740 – 746, the 6th International Conference on Smart Computing and Communications.

[42] D. Poshyvanyk, A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, in: 15th IEEE International Conference on Program Comprehension (ICPC '07), 2007, pp. 37–48.

[43] A. M. Saeidi, J. Hage, R. Khadka, S. Jansen, A search-based approach to multi-view clustering of software systems, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 429–438.

[44] A. Rathee, J. K. Chhabra, Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering, in: D. Singh, B. Raman, A. K. Luhach, P. Lingras (Eds.), Advanced Informatics for Computing Research, Springer Singapore, Singapore, 2017, pp. 94–106.

[45] F. S. Al-Anzi, D. AbuZeina, Toward an enhanced arabic text classification using cosine similarity and latent semantic indexing, Journal of King Saud University - Computer and Information Sciences 29 (2) (2017) 189 – 195, arabic Natural Language Processing: Models, Systems and Applications.

[46] T. Ball, J. min Kim, A. A. Porter, H. P. Siy, If your version control system could talk... (1997).

[47] H. Gall, M. Jazayeri, J. Krajewski, Cvs release history data for detecting logical couplings, in: Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings., 2003, pp. 13–23.

[48] L. L. Silva, D. Felix, M. T. Valente, M. de Almeida Maia, Modularity-check: A tool for assessing modularity using co-change clusters, CoRR abs/1506.05754. arXiv:1506.05754.

[49] I. S. Wiese, R. T. Kuroda, G. A. Oliva, Do historical metrics and developers communication aid to predict change couplings?, IEEE Latin America Transactions 13 (6) (2015) 1979–1988.

[50] T. Zimmermann, S. Diehl, A. Zeller, How history justifies system architecture (or not), in: Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings., 2003, pp. 73–83.

[51] L. Yu, Understanding component co-evolution with a study on linux, Empirical Software Engineering 12 (2) (2007) 123–141.

[52] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: 2009 16th Working Conference on Reverse Engineering, 2009, pp. 135–144. doi:10.1109/WCRE.2009.19.

[53] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, SIGMOD Rec. 22 (2) (1993) 207–216.

[54] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, IEEE Transactions on Software Engineering 31 (6) (2005) 429–445.

[55] H. Jain, K. Deb, An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach, IEEE Transactions on Evolutionary Computation 18 (4) (2014) 602–622.

[56] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using nsga-iii, ACM Trans. Softw. Eng. Methodol. 24 (3) (2015) 17:1–17:45.

[57] L. Cai, S. Qu, Y. Yuan, X. Yao, A clustering-ranking method for many-objective optimization, Applied Soft Computing 35 (2015) 681 – 694.

[58] B. S. Mitchell, A heuristic approach to solving the software clustering problem, in: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., 2003, pp. 285–288.

[59] N. Razali, Y. B. Wah, Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests, Journal of Statistical Modeling and Analytics 2 (1).

[60] R. A. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[61] G. Scanniello, A. D'Amico, C. D'Amico, T. D'Amico, Architectural layer recovery for software system understanding and evolution, Software: Practice and Experience 40 (10) 897–916.

[62] C. Y. Chong, S. P. Lee, Automatic clustering constraints derivation from object-oriented software using weighted complex network with graph theory analysis, Journal of Systems and Software 133 (2017) 28 – 53.

[63] P. van der Spek, S. Klusener, Applying a dynamic threshold to improve cluster detection of lsi, Science of Computer Programming 76 (12) (2011) 1261 – 1274, special Issue on Software Evolution, Adaptability and Variability.

[64] H. Masoud, S. Jalili, A clustering-based model for class responsibility assignment problem in object-oriented analysis, Journal of Systems and Software 93 (2014) 110 – 131.

[65] G. Caldiera, V. R. Basili, Identifying and qualifying reusable software components, Computer 24 (2) (1991) 61–70. doi:10.1109/2.67210.

[66] J. Garcia, I. Krka, C. Mattmann, N. Medvidovic, Obtaining ground-truth software architectures, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 901–910.