PCI-PSO: Preference-Based Component Identification Using Particle Swarm Optimization

Article *in* Journal of Intelligent Systems · January 2017

DOI: 10.1515/jisys-2017-0244

2 authors, including:

Seyed Mohammad Hossein Hasheminejad
Alzahra University
21 PUBLICATIONS **283** CITATIONS

Seyed Mohammad Hossein Hasheminejad* and Shabnam Gholamshahi

PCI-PSO: Preference-Based Component Identification Using Particle Swarm Optimization

https://doi.org/10.1515/jisys-2017-0244
Received May 25, 2017.

Abstract: Nowadays, component identification is one of the main challenges of software analysis and design. The component identification process aims at clustering classes into components and subcomponents. There are a number of methods to identify components in the literature; however, most of them cannot be customized to software architect's preferences. To address this limitation, in this paper, we propose a preference-based method by the name of preference-based component identification using particle swarm optimization (PCI-PSO) to identify logical components. PCI-PSO provides a novel method to handle the software architect's preferences using an interactive (i.e. human in the loop) search. PCI-PSO employs a customized PSO to automatically classify classes into suitable logical components and avoid the problem of identifying the proper number of components. We evaluated the effectiveness of PCI-PSO with four real-world cases. Results revealed that PCI-PSO has an ability to identify more cohesive and independent components with respect to the software architect's preferences in comparison to the existing component identification methods.

Keywords: Component identification, software architect's preferences, particle swarm optimization.

1 Introduction

In the component-based software development (CBSD) process, identifying components for a given system is a crucial task. Component identification during the software design phase is a process in which the functionalities of a given system are partitioned into nonintersecting logical components to provide the starting point for designing software architecture [7, 27]. These logical components have two properties [7]: (1) homogeneity within a component (i.e. functionalities belonging to a component should be logically coherent) and (2) heterogeneity between components (i.e. functionalities of one component should be distinct from those belonging to other components).

Many automatic or semiautomatic methods, including graph partitioning method [3, 8, 36], clustering-based method [1, 19–21, 23, 25, 26, 29, 39], create, read, update, and delete matrices (CRUD)-based method [13, 28], and formal concept analysis (FCA)-based method [9, 15], have been proposed to identify components at an early stage of software design. However, these methods have two major drawbacks. First, they often use different classical clustering techniques like k-means, which are inefficient to deal with complex search landscapes due to their simple greedy and heuristic nature [42]. For this limitation, we proposed a genetic-based algorithm by the name of software component identification using genetic algorithm (SCI-GA) [19] to improve this challenge. The experimental results presented in Ref. [19] demonstrated that SCI-GA using a heuristic algorithm (i.e. GA) has far better performance in component identification than classical clustering techniques. Second, most of these methods introduce completely automated algorithms, which cannot be influenced by the software architect at all [8]. Consequently, they do not allow a software architect to identify and

***Corresponding author: Seyed Mohammad Hossein Hasheminejad,** Department of Computer Engineering, Alzahra University, Tehran, Iran, e-mail: SMH.Hasheminejad@Alzahra.ac.ir
Shabnam Gholamshahi: Department of Computer Engineering, Alzahra University, Tehran, Iran

reflect emerging conflicts during the identification process. For this limitation, we proposed another genetic-based algorithm by the name of search-based logical component identification (SBLCI) [20] to address software architect's preferences. In Ref. [20], software architects can determine the number of components, the size and complexity of each component, and the cohesion threshold of each component. Furthermore, in Ref. [20], a suitable hierarchy of components with their subcomponents according to software architect's preferences can be achieved. Moreover, in Ref. [21], we proposed a customized hierarchical evolutionary algorithm (HEA) by the name of clustering analysis classes to identify software components (CCIC) to automatically identify appropriate logical components. In CCIC [21], software architects can determine several constraints related to deployment and implementation frameworks in the component identification process.

However, SBLCI [20] and CCIC [21] cannot employ all knowledge of software architects to guide its component identification method, and with its constraints, a little knowledge of software architects is considered to identify component. Therefore, it is required to provide a new method for employing all knowledge and preferences of software architects to guide the component identification process. With respect to the above points, in this paper, we propose a preference-based method by the name of preference-based component identification using particle swarm optimization (PCI-PSO), which is based on PSO to identify logical components. The main contribution of PCI-PSO is employing software architect's preferences during the LCI process. In fact, the proposed PCI-PSO method is a semiautomatic method in which the opinion of software architects is taken and based on which a guided search for the near-optimal logical component is performed. The motivation of the proposed PCI-PSO method is that the previous works [19–21] have no ability to employ dynamically software architect's preferences in the search process. These previous works [19–21] used software architect's knowledge in the start time of the search process; accordingly, this makes it impossible to correct the search process when it converges in a deviation direction. Therefore, PCI-PSO provides a novel method to handle software architect's preferences using an interactive (i.e. human in the loop) search process. Consequently, logical components in PCI-PSO are obtained according to software architect's preferences.

The main challenge of PCI-PSO is a way to apply software architect's knowledge during the search process. Because the goal of this paper is employing dynamically the software architect's preferences in the search process, it is assumed that the input of PCI-PSO is a class diagram derived at an early stage of software design; then, it needs to have a way to calculate the connection strength between a pair of classes.

To evaluate PCI-PSO, we describe an empirical study involving a number of software engineers in four case studies of an early lifecycle software design. The results reveal that the proposed PCI-PSO method outperforms other clustering-based methods such as SCI-GA [19], SBLCI [20], CCIC [21], neural network [23], and hybrid methods [1].

The rest of this paper is organized as follows. In Section 2, the related works are discussed. In Section 3, an overview on PSO concepts is provided as a background. In Section 4, PCI-PSO is described with more details. Section 5 presents the case studies and the empirical analysis. Finally, the paper concludes with Section 6.

2 Related Works

The works related to the automatic component identification at an early stage of software design can be divided into four categories: graph partitioning method [3, 8, 36], clustering-based method [1, 19–21, 23, 25, 26, 29, 39], CRUD-based method [13, 28], and FCA-based method [9, 15], which are discussed in detail below.

2.1 Graph Partitioning

Albani et al. [3] mapped domain models (data objects, process steps, and actors) into vertices and edges of a graph; then, based on relation types between domain model elements and designer's preferences, the weights were assigned to edges. Finally, the graph was partitioned into components using a heuristic from

graph theories. Peng et al. [36] transformed the relationship model among business elements to a weighted graph. Then, a graph segmentation method was applied on the graph to identify mutually disjoint subgraphs as components. The authors claimed that their proposed method has achieved cohesive components with low coupling, but they did not evaluate their claim. In Ref. [8], the analytic hierarchy process (AHP) was used to extract designer's preferences to compute weights for each edge type of the graph. In fact, it attempted to determine systematically the weights of CRUD relationships according to the designer's preferences. It should be noted that, unlike other methods, the work presented in Ref. [8] could automatically identify the number of business components.

2.2 Clustering-Based Methods

Lee et al. [29] proposed a method for clustering classes into logical components with high cohesion and low coupling. At first, key classes are selected as candidate components; then, other classes are assigned to components with the highest level of dependency. Note that, in Ref. [29], identifying key classes was a critical problem and was determined manually by experts. Jain et al. [25] used hierarchical agglomerative clustering techniques to iteratively cluster two elements (i.e. classes) with the highest strength. The strength between elements is measured using weighted relations determined manually by experts. Kim and Chang [26] employed use case models, object models, and collaboration diagrams to identify components. For clustering-related functions, functional dependencies of use cases were calculated; then, related use cases were clustered. This work requires weighting and does not give any guidelines in this regard. Shahmohammadi et al. [39] proposed a feature-based clustering method to identify logical components in which several features like actors and entity classes are presented to measure the similarity between a pair of use cases. Therefore, several classical clustering techniques like k-means, hierarchical method, graph-based method, and fuzzy C-means were examined to achieve suitable software architecture. In Ref. [23], to cluster use cases into components, a method based on the neural network approach was proposed that achieves better results in comparison to Ref. [39]. In Ref. [1], to cluster use cases into components, a hybrid fuzzy clustering algorithm and a PSO algorithm were proposed. However, these works [1, 23, 39] did not pay attention to software architect's preferences during the component identification process. In Ref. [19], we proposed a clustering-based method by the name of SCI-GA, which is based on a GA, to identify software components from use case models of a given system. In Ref. [20], we proposed a clustering-based method by the name of SBLCI [20] to improve SCI-GA [19] and address software architect's preferences. In Ref. [20], software architects can determine the number of components and the size, complexity, and cohesion threshold of each component. Furthermore, in Ref. [20], a suitable hierarchy of components with their subcomponents according to software architect's preferences can be achieved. In Ref. [21], we proposed a customized HEA by the name of CCIC to identify automatically appropriate logical components. In CCIC [21], software architects can determine several constraints related to deployment and implementation frameworks in the component identification process. In CCIC [21], it is necessary to find a way to calculate the connection strength between a pair of classes. Therefore, we proposed five weighing methods to calculate the connection strength between a pair of classes. CCIC [21] addresses three design concerns from the software architect's viewpoint, including deployment constraints, implementation framework constraints, and legacy components to be reused.

2.3 CRUD-Based Methods

Lee et al. [28] presented a tool called COMO in which "use case/class matrix" was created with respect to use case diagrams and class diagrams; then, it was partitioned into blocks with tight cohesion as business components. Ganesan and Sengupta [13] presented a tool partly similar to COMO called O2BC, but it has several differences in their clustering technique, and used business events and domain objects as inputs. However, these two CRUD-based methods have a number of limitations similar to clustering-based methods, which are used as classical clustering techniques.

2.4 FCA-Based Methods

Hamza [15] initially proposed a framework based on FCA to partition a class diagram into logical components with several heuristics similar to clustering techniques. However, this framework emphasizes stability instead of cohesion and coupling metric as important metrics to identify components. Cai et al. [9] proposed a novel method based on fuzzy FCA. They transformed business elements and their memberships into a lattice; then, they used a clustering technique to identify components. They used dispersion and distance concepts to measure cohesion and coupling metrics, respectively. An important advantage of the method of Cai et al. [9] in contrast to other methods is that it considers cohesion and coupling simultaneously throughout the identification process. Note that this property leads to a suitable trade-off between these metrics. However, they used two dispersion and distance thresholds (i.e. T_D and T_S thresholds for computing cohesion and coupling, respectively) with a high effect on the performance of their method, which must be manually determined by practical experiences.

Although many methods with a variety of categories have been proposed to identify software components, several shortcomings remain. First of all, most of these methods need to determine the number of components in advance. Furthermore, most of these methods do not consider cohesion and coupling simultaneously throughout the identification process. Finally, they cannot be customized to project-specific preferences, except for Refs. [8, 20, 21], because they introduce completely automated procedures that cannot be influenced by software architects. To address these shortcomings, we propose an interactive PSO capable of identifying components according to software architect's preferences in which the software architects can interact with the proposed method and apply their preferences during the evolutionary process.

3 PSO

PSO is an artificial intelligence and computational technique. It has a capability of optimizing a nonlinear and multidimensional problem. It has better convergence among evolutionary algorithms such as GA. It is an evolutionary algorithm that belongs to the class of swarm intelligence algorithms, which are inspired from social dynamics and emergent behavior that arise in socially organized colonies like the social behavior of bird flocking or fish schooling. The main strength of PSO is its fast convergence in compression with many global optimization algorithms. In PSO, each particle tries to achieve the best result by updating its position and speed according to its own past as well as information of the current particle, which is the best among all particles in swarm (PSO combines self-experiences with social experiences). The basic concept of PSO is to create a swarm of particles, which flies through a search space. However, this search process is not carried out entirely randomly; there are some factors that influence this process, which are defined as follows: the best position visited by itself (its own best experience), the position of the best particle in its neighborhood (the social experience), and its current velocity. If a particle takes an entire population as its neighbors, the best value for the best social experience is a global best (called Gbest), and when it takes the smaller group as its neighbors, the best value is a local best (called Pbest). The performance of each particle is measured according to a predefined fitness function. The position and velocity of each one are updated using the following equations:

$$v_i^{t+1} = wv_i^t + c_1 r_1 (\text{Pbest}_i^t - x_i^t) + c_2 r_2 (\text{Gbest}^t - x_i^t) \tag{1}$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{2}$$

where x_i^t and v_i^t are defined as the current position and velocity of i^{th} particle at iteration t, respectively. Gbest^t and Pbest_i^t are the global and personal best positions coming out from particles' experience of i^{th} particle during iterations 1 to t, respectively. w is the inertia weight that controls the impact of the previous velocities on the current velocity. r_1 and r_2 are random variables in range [0,1] to provide casual weighting of

the different components participating in the particle velocity definition. c_1 and c_2 are using to control the impact of the personal best and the global best, respectively. After updating the velocity and position of a particle, its Pbest are updated according to Equation (3):

$$\text{Pbest}_i^{t+1} = x_i^{t+1} \; x_i^{t+1} \text{Pbest}_i^t \text{Pbest}_i^t \text{ Otherwise} \tag{3}$$

where $f(x_i^{t+1}) < f(\text{Pbest}_i^t)$ means that the new position x_i^{t+1} is better than the current Pbest of the i^{th} particle. After updating the velocity, position, and Pbest of all particles, the particle with the best fitness is selected as Gbest^{t+1}. These operations are repeated until a termination criterion is met (e.g. the number of iterations is performed or the adequate fitness is reached).

4 Proposed PCI-PSO Method

Over the last decades, several techniques have been proposed to solve optimization problems. Among them, evolutionary algorithms achieve considerable performance and solve different optimization problems such as GA [19–21], PSO [31], gradient-based water cycle algorithm (GWCA) [35], and harmony search (HS) [6]. Evolutionary algorithms have been used in different areas such as fraud detection [22], traffic accident severity prediction [16, 24], and class diagram design [41]. The LCI problem is an NP-complete problem [11]; therefore, the goal of this paper is to map the LCI problem to an optimization problem and solve this problem using the search-based approach.

In our previous work [31], a discrete PSO algorithm called CPSOII to solve partitional clustering problems was proposed, and the results of Ref. [31] revealed that CPSOII outperforms other PSO and GA algorithms in partitional clustering problems. In this paper, we propose a novel component identification method based on CPSOII idea, called PCI-PSO as shown in Figure 1, which is based on software architect's preferences.

The reason for using CPSOII in this paper is that the LCI is similar to partitional clustering problems in which some analysis classes are assigned to some logical components. On the contrary, based on the obtained results mentioned in Ref. [31], the CPSOII algorithm has a better average Davies-Bouldin index (DBI) [12] in comparison to GA and PSO for partitional clustering problems. Therefore, we employ CPSOII as a partitional clustering algorithm to achieve the better results in the LCI.

The main contribution of PCI-PSO is employing software architect's preferences during the LCI process. PCI-PSO provides a novel method to handle the software architect's preferences using the interactive (i.e. human in the loop) search process. Consequently, the logical components in PCI-PSO are obtained according to software architect's preferences. The inputs of PCI-PSO are a number of class diagrams and software architect or project preferences, and its output is a set of identified logical components.

A feature vector can quantitatively represent a class. In feature vector representation, each class is represented by a number of properties. In this paper, each class is represented by a vector of connection strengths with other ones. This connection strength can be calculated by binary weighting method (W_{B}), connection frequency weighting method (W_{CF}), connection frequency inverse class frequency weighting method (W_{CFICF}), connection frequency collection weighting method (W_{CFC}), or entropy weighting method (W_{E}) presented in Ref. [21]. After applying the weighting methods, to compute the connection strength between a pair of classes, it is necessary to use a similarity measure to evaluate the similarity between them. There are many similarity measures to compute the similarity between a pair of vectors in terms of features [42]. For this reason, two popular similarity measures, including Jaccard-NM [35] and unbiased Ellenberg-NM (E_u-NM) [35], are used in this study.

As shown in Figure 1, PCI-PSO consists of eight steps. In the first step of PCI-PSO, all defined weighting and similarity measures are calculated. In the second step of PCI-PSO, algorithm parameters, such as swarm size, maximum number of iterations, and parameters used in the velocity equation, are initialized. In the third step, random initial populations are generated. In the fourth and fifth steps, a fitness function for all

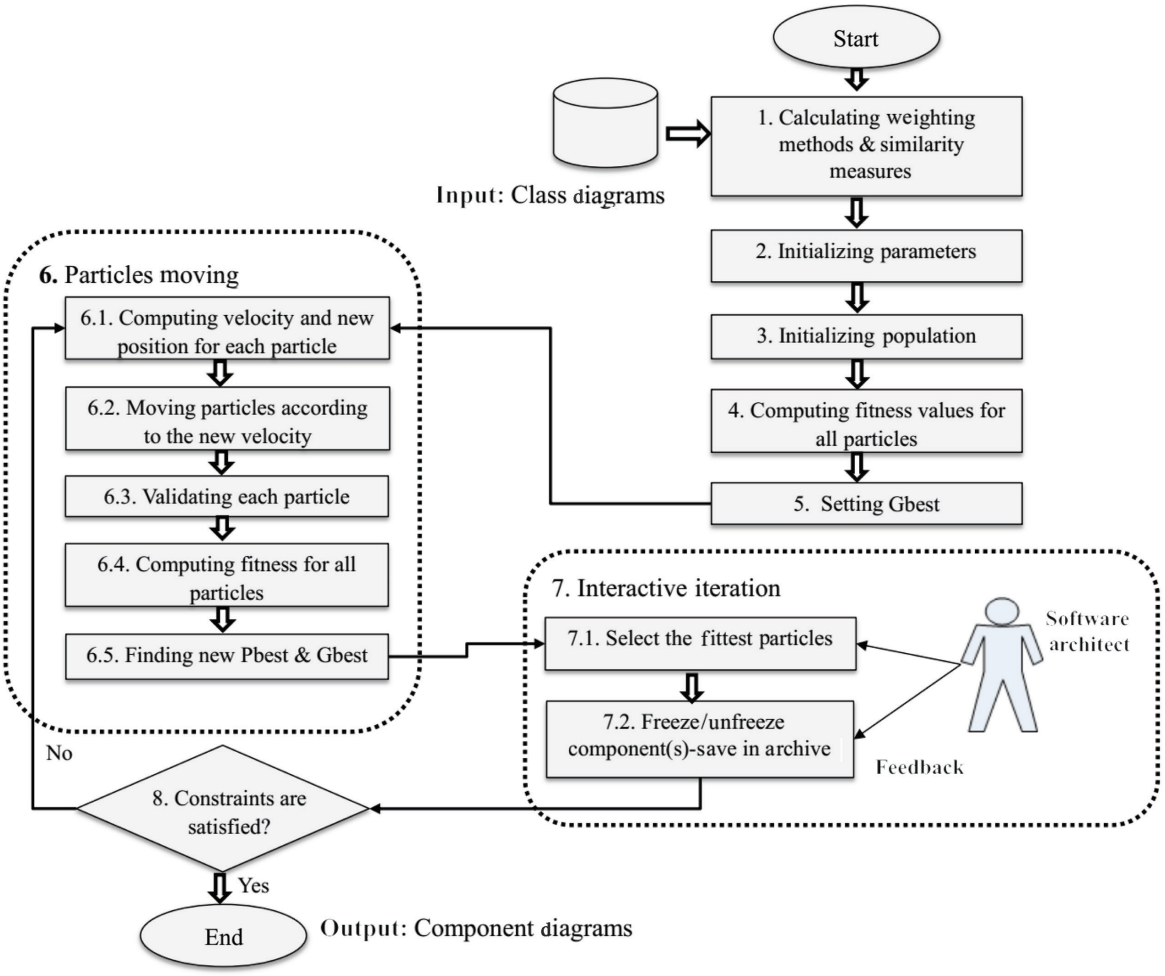

Figure 1: Process of the Proposed PCI-PSO Algorithm.

particles are evaluated, and the best particle is set as Gbest, respectively. The sixth step includes five substeps to move each particle and determine the new Pbest and Gbest. In the seventh step of PCI-PSO, there is an interaction iteration to communicate with software architects to support their preferences during the evolution process. In the interactive iteration step, at first, a software architect can select some fittest particles (i.e. component diagrams) and she/he can perform three actions on each solution including unfreezing/freezing component(s) and saving the solution in an archive. Note that, in the interactive iteration step, the software architect can see fittest solutions and guide the evolutionary process according to its preferences. Finally, this process is repeated until the fittest particle satisfies some conditions or the maximum number of iterations is exceeded. The following sections describe the steps of PCI-PSO.

4.1 Solution Representation

An important element in all search-based approaches is the representation of solutions (i.e. how to model logical components of a system). As mentioned in Section 3, each solution is represented by a particle in PSO. The classes of a given system are encoded in a particle in the PCI-PSO encoding. The values of each particle can be an integer number in the range of $[1,k_{max}]$, which represents the component number of the corresponding class. Note that k_{max} is a user-defined maximum number of components that can be determined by soft-

ware architects as an input at the beginning of PCI-PSO. Therefore, the length of each particle in PCI-PSO is equal to the number of all classes.

4.2 Swarm Initialization

The most usual technique for initialization population is random uniform initialization in which each particle of the initial swarm and consequently the initial best position are drawn by sampling a uniform distribution over the search space. To apply the evolution process of the proposed PCI-PSO method in step 3, as shown in Figure 1, it is necessary to first generate the initial population. Therefore, the position of each particle is initialized with a uniformly distributed random vector, with $K_{\min}$ and $K_{\max}$ for each specific attribute in the search space. After generating the initial particles of swarm, the initial velocity of all particles is assigned to zero and the initial Pbest of each particle is assigned to its current position.

4.3 Fitness Function

There are a number of criteria to identify suitable components. The coupling and cohesion aspects of software components are fundamental quality attributes that can seriously influence a software architecture, maintenance, evolution, and reuse. Coupling criterion denotes heterogeneity between components (i.e. classes in a component show the least dependency with other ones). Cohesion criterion denotes homogeneity within a component (i.e. classes in a component should be as similar as possible). Therefore, in this study, a component is considered as a cluster, and analysis classes within a component are considered as members of the cluster. We use a DBI [12] as popular clustering criterion.

The main objective of the DBI criterion is to maximize between-component separation and minimize within-component scatter. The DBI criterion combines both objects (i.e. between-component separation and within-component scatter) in one function. The within-component scatter for component Cmp_i is defined as follows:

$$S_{i,q} = \left(\frac{1}{\left|\mathrm{Cmp}_i\right|} \sum_{O \in \mathrm{Cmp}_i} \left\| O - c_i \right\|_2^q \right)^{\frac{1}{q}} \tag{4}$$

where $S_{i,q}$ denotes the qth root of the qth moment of the classes belonging to component Cmp_i with respect to their mean, and $\| O - c_i \|_2^q$ denotes the qth power of the differences between the individual feature vectors of O and c_i. The distance between component Cmp_i and Cmp$j$ is defined as follows:

$$d_{ij,t} = \left\| c_i - c_j \right\|_t \tag{5}$$

In Equations (4) and (5), q and t are integer numbers, where (q and $t \geq 1$), and they can be selected independently. Note that both of them are equal to 2 [12]. In the following equation, $R_{i,qt}$ for component Cmp_i is defined as follows:

$$R_{i,qt} = \max_{j, j \neq i} \left\{ \frac{s_{i,q} + s_{j,q}}{d_{ij,t}} \right\} \tag{6}$$

Finally, the DBI criterion is defined as follows:

$$\mathrm{DBI} = \frac{1}{k} \sum_{i=1}^{k} R_{i,qt} \tag{7}$$

Note that a small value of the DBI criterion shows better component solution.

4.4 Velocity Computation and Particle Position Movement

In PSO algorithms and its extension, there is a velocity vector that drives the optimization process. In PCI-PSO (in step 6.1 as shown in Figure 1), the position and velocity equations of each particle are modified by Equations (8) and (9) presented in CPSOII [31].

$$V_i^{t+1} = W_i^t \otimes V_i^t \oplus \{[R_1 \otimes (\text{Pbest}_i^t \ominus X_i^t)] \oplus [R_2 \otimes (\text{Gbest}^t \ominus X_i^t)]\} \tag{8}$$

$$X_i^{t+1} = X_i^t + X_i^{t+1} \tag{9}$$

where X_i^t and V_i^t are the position and velocity of particle i at iteration t, respectively. Pbest_i^t and Gbest^t are the best positions obtained by particle I and the swarm of particles during time t, respectively. R_1 and R_2 are vectors comprising 0 or 1 elements, such that the values of these vectors are randomly generated with probability r_1 and r_2, respectively.

Note that because the inertia weight [i.e. W_i^t in Equation (8)] is critical for the performance of PSO, which balances the global exploration and local exploitation abilities of the swarm [4, 5], we employ the APSO method [5] to achieve faster convergence speed and better results. In the APSO method [5], the inertia weight is dynamically adapted for every particle by considering a measure called adjacency index (AI) as described in the following equations.

$$\text{AI}_i^t = \frac{F(\text{Pbest}_i^1) - F_{\text{KN}}}{F(\text{Pbest}_i^t) - F_{\text{KN}}} - 1 \tag{10}$$

$$w_i^t = \frac{1}{1 + e^{-(a \times \text{AI}_i^t)}} \tag{11}$$

where $F(\text{Pbest}_i^t)$ is the fitness (i.e. the DBI criterion) of the best previous position of i^{th} particle, and F_{KN} is the known real optimal solution value that we assumed as zero. In fact, a small AI_i means that the fitness of i^{th} particle is far away from the global optimal value and it needs a strong global exploration; therefore, this particle needs a large inertia weight. On the contrary, for a big AI_i, that particle has a high adjacency to the global optimum and it needs local exploitation; therefore, that particle needs a small inertia weight. The value of parameter α controls the decreasing speed of inertia weight. To identify the best value of α on the variation of inertia weight, this value will vary from 0.1 to 1 with step size 0.1 in Section 5.

4.5 Software Architect's Preferences

It should be noted that the goal of automated LCI methods [1, 19, 23, 39] is to optimize the clustering fitness function to achieve the best possible solution. However, as mentioned earlier, software architects or project preferences have a strong impact on the performance of the component identification process, but all previous logical identification methods, except for Refs. [8, 20, 21], do not take them into account in their identification process. In PCI-PSO, the software architect has a second possibility to affect vigorously the component identification process, attempting to provide facilities to make use of the knowledge of software architects. In fact, PCI-PSO provides a way to apply software architect's preferences using interactive iterations in the evolutionary process. In PCI-PSO, software architects can apply their preferences with some actions including unfreezing/freezing component(s) and saving a solution in an archive (in step 7.2 of PCI-PSO).

Note that candidate solutions (i.e. the Gbest and 5 to 10 fittest Pbest particles) are presented in the form of UML components diagram to software architects. Therefore, software architects can freeze/unfreeze each component of them or save these solutions in the archive, which can be remembered by software architects to compare among the interesting solutions. After freezing/unfreezing the components of the candidate solutions by software architects, PCI-PSO attempts to ensure that the generated solutions survive in the next generation. This interaction is time consuming. Therefore, to reduce the time taken to discover good solutions, in PCI-PSO, after 10 iterations, the fittest solutions are displayed to the software architect.

4.6 Termination Condition

To end the process of particles evolution, a termination condition is necessary. In PCI-PSO, the evolution is terminated when the best fitness function value of all particles does not improve during several generations (e.g. 50 generations), or when the number of iteration exceeds the max number of iterations and then, the evolution of the particle swarm is completed.

5 Experimental Results

In this section, the proposed PCI-PSO method is evaluated and the obtained results are reported. Moreover, a metric is described to compare components identified by PCI-PSO to the ones identified manually by experts. We used four real-world systems in this study, which are different in terms of software architect and project preferences, number of classes, and application domains. Table 1 provides the details of the used systems. To illustrate these case studies, we introduce them as follows:

- Home appliance control system (HACS) [14]: This system is a centralized, remotely accessed, intelligent system responsible to act as an intermediary between the user and his home appliances.
- Online broker system (OBS) [34]: This system automates traditional stock trading using the Internet and gives faster access to stock reports, current market trends, and real-time stock prices.
- Loan generation system (LGS) [30]: The goal of this system is to generate loan automatically for the requests of customers. This system is implemented by Yass-System Company, a famous software house company in Iran, for several Iranian banks.
- Agri insurance system (AIS) [2]: This system provides farmers with financial protection against production losses caused by natural perils, such as drought, flood, hail, frost, excessive moisture, and insects. This system is designed and implemented by Yass-System Company for Agriculture Bank, one of the Iranian banks customized to the agriculture finance for Iranian farmers.

The proposed PCI-PSO method was stopped when the generation number exceeded 2000 or the best fitness function (i.e. the DBI criterion) value did not improve during the last 50 generations. To identify the impact of α on the performance of the employed APSO, different values of α, 500 particles, $c_1 = c_2 = 2$ (according to the results presented in Ref. [5]), and 2000 as the max number of iterations, were run without interactive iteration (i.e. Step 7 in the proposed PCI-PSO). Note that, for each value of α, 30 runs of the PCI-PSO without interactive iteration were performed. Figure 2 shows the mean best (the minimum) DBI values for the average of 30 runs. As shown in Figure 2, α values in range [0.1,1] are considered and the best value of α is equal to 0.6 for all the four cases.

As mentioned earlier, the values of r_1 and r_2 determine the exploration and exploitation abilities of the proposed PCI-PSO method. In PCI-PSO, high values of r_1 and r_2 lead to the movement of particles toward Pbest and Gbest, respectively. To improve the performance of the PCI-PSO search process, it is necessary to determine the best value for each of these parameters. For this reason, 30 runs of PCI-PSO with different values of r_1 and r_2 (for all cases of these parameters in the range [0,1] with step 0.05) and $\alpha = 0.6$ were

Table 1: Details of the Four Real-World Systems as Case Studies.

System	Number of use cases	Number of classes				Number of connections among classes
		Boundary	Control	Entity	Total	
HACS	13					193
OBS	30	4	10	6	20	227
LGS	36	10	15	54	79	1912
AIS	68	43	36	154	233	6471

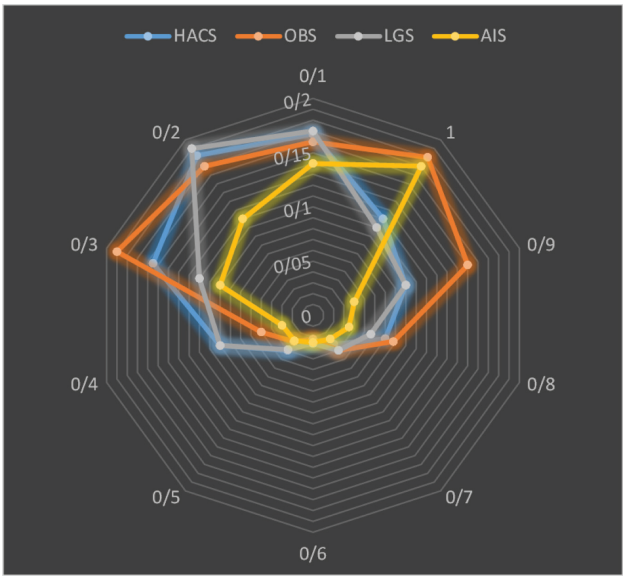

Figure 2: Mean Best DBI Values with Different Values of α for All Four Cases.

performed. Therefore, the average values of the best results for 30 runs were chosen. In this experiment, PCI-PSO was run without interactive iteration (i.e. Step 7 in the proposed PCI-PSO method) and the other parameters were set as follows: 500 particles, $c_1 = c_2 = 2$ (according to the results presented in Ref. [5]), and 2000 as the max number of iterations. Tables 2 and 3 illustrate the impacts of the r_1 and r_2 parameters on the DBI criterion using the four cases. The results revealed that the suitable value is equal to 0.65 for r_1 (Table 2) and 0.75 for r_2 (Table 3); therefore, in the following experiment, these values were used for setting parameters.

Table 2: Effects of r_1 Parameter on the DBI Criterion Using the Four Cases (Average of 30 Runs).

$r_2 = 0.75, \alpha = 0.6$	Case study			
r_1	HACS	OBS	LGS	AIS
0	0.156; 0.023	0.256; 0.015	0.352; 0.016	0.322; 0.014
0.05	0.132; 0.012	0.145; 0.014	0.253; 0.014	0.265; 0.011
0.10	0.144; 0.013	0.142; 0.011	0.247; 0.015	0.286; 0.012
0.15	0.104; 0.011	0.155; 0.013	0.235; 0.015	0.281; 0.015
0.20	0.125; 0.011	0.144; 0.011	0.220; 0.012	0.214; 0.012
0.25	0.111; 0.010	0.132; 0.012	0.189; 0.011	0.192; 0.011
0.30	0.98; 0.009	0.145; 0.011	0.171; 0.012	0.166; 0.011
0.35	0.91; 0.009	0.122; 0.011	0.119; 0.012	0.153; 0.011
0.40	0.85; 0.007	0.132; 0.011	0.115; 0.012	0.151; 0.011
0.45	0.33; 0.005	0.75; 0.009	0.91; 0.009	0.148; 0.011
0.55	0.10; 0.003	0.19; 0.005	0.25; 0.007	0.77; 0.009
0.60	0.07; 0.002	0.081; 0.004	0.091; 0.003	0.088; 0.008
0.65	0.017; 0.002	0.019; 0.002	0.019; 0.002	0.014; 0.002
0.70	0.026; 0.003	0.029; 0.002	0.068; 0.002	0.015; 0.002
0.75	0.095; 0.007	0.039; 0.002	0.094; 0.003	0.019; 0.002
0.80	0.156; 0.009	0.092; 0.002	0.112; 0.007	0.065; 0.006
0.85	0.141; 0.011	0.102; 0.008	0.113; 0.011	0.099; 0.007
0.90	0.122; 0.011	0.116; 0.011	0.121; 0.012	0.111; 0.011
0.95	0.158; 0.013	0.256; 0.011	0.129; 0.011	0.175; 0.012
1	0.233; 0.016	0.289; 0.012	0.199; 0.012	0.123; 0.011

Table 3: Effects of r_2 Parameter on the DBI Criterion Using the Four Cases (Average of 30 Runs).

$r_1=0.65, \alpha=0.6$	Case study			
r_2	HACS	OBS	LGS	AIS
0	0.171; 0.022	0.243; 0.016	0.263; 0.015	0.285; 0.015
0.05	0.143; 0.013	0.175; 0.013	0.222; 0.015	0.232; 0.014
0.10	0.148; 0.015	0.166; 0.010	0.215; 0.015	0.212; 0.014
0.15	0.126; 0.011	0.145; 0.013	0.205; 0.015	0.201; 0.015
0.20	0.115; 0.011	0.131; 0.012	0.201; 0.015	0.191; 0.014
0.25	0.102; 0.010	0.139; 0.013	0.195; 0.013	0.155; 0.014
0.30	0.91; 0.009	0.155; 0.011	0.166; 0.013	0.106; 0.012
0.35	0.89; 0.009	0.119; 0.01	0.152; 0.012	0.91; 0.011
0.40	0.81; 0.007	0.126; 0.01	0.121; 0.012	0.23; 0.009
0.45	0.46; 0.005	0.89; 0.01	0.105; 0.011	0.09; 0.005
0.55	0.23; 0.005	0.53; 0.005	0.99; 0.011	0.062; 0.005
0.60	0.05; 0.002	0.12; 0.002	0.43; 0.009	0.049; 0.003
0.65	0.03; 0.002	0.089; 0.002	0.091; 0.004	0.031; 0.002
0.70	0.03; 0.002	0.054; 0.002	0.045; 0.004	0.017; 0.002
0.75	0.017; 0.002	0.019; 0.002	0.019; 0.003	0.014; 0.002
0.80	0.08; 0.007	0.033; 0.002	0.053; 0.004	0.017; 0.002
0.85	0.09; 0.007	0.084; 0.009	0.089; 0.009	0.033; 0.002
0.90	0.11; 0.01	0.112; 0.01	0.909; 0.009	0.086; 0.008
0.95	0.123; 0.011	0.155; 0.011	0.111; 0.011	0.099; 0.009
1	0.189; 0.015	0.181; 0.012	0.159; 0.012	0.103; 0.009

We introduce a metric to evaluate the components identified by the proposed PCI-PSO method in comparison to the expert's ones. At first, the components identified by PCI-PSO must be mapped to the ones identified manually by experts. To perform this task, we use the Hungarian algorithm [32], which is a combinatorial optimization algorithm that solves the assignment problem between the components identified by PCI-PSO and the ones identified by experts in polynomial time. Then, for each component identified manually by experts, the true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) values depicted in Figure 3 are computed. Note that, in Figure 3, EC is the number of components identified by experts. There are a number of metrics to measure the accuracy of the predicted results such as Precision, Recall, and F-measure [38]. However, Shepperd [40] revealed that these metrics have been shown to be problematic in software engineering, because they focus only on positive predictions. For this reason, we use the Matthews correlation coefficient metric [40] called Sim2Manual metric to measure the similarity between the components identified manually by experts and the ones identified by other methods. Equation (12) illustrates the Sim2Manual metric. Note that the +1 and –1 values of Sim2Manual denote the perfect predictor and perfectly perverted predictor, respectively.

$$\text{Sim2Manual} = \frac{\text{TP}\times\text{TN}-\text{FP}\times\text{FN}}{\sqrt{(\text{TP}+\text{FP})\times(\text{TP}+\text{FN})\times(\text{TN}+\text{FP})\times(\text{TN}+\text{FN})}} \tag{12}$$

* Suppose that component j identified by PCI-PSO is mapped to component i identified manually by experts.

		Classes belonging to component i identified manually by experts	
		Yes	No
Classes belonging to component j identified by PCI-PSO	Yes	TP_j (True positive)	FP_j (False positive)
	No	FN_j (False negative)	TN_j (True negative)

$\text{TP} = \sum_{j=1}^{\lvert EC\rvert} \text{TP}_j$

$\text{FP} = \sum_{j=1}^{\lvert EC\rvert} \text{FP}_j$

$\text{FN} = \sum_{j=1}^{\lvert EC\rvert} \text{FN}_j$

$\text{TN} = \sum_{j=1}^{\lvert EC\rvert} \text{TN}_j$

Figure 3: Definitions of TP, FP, FN, and TN.

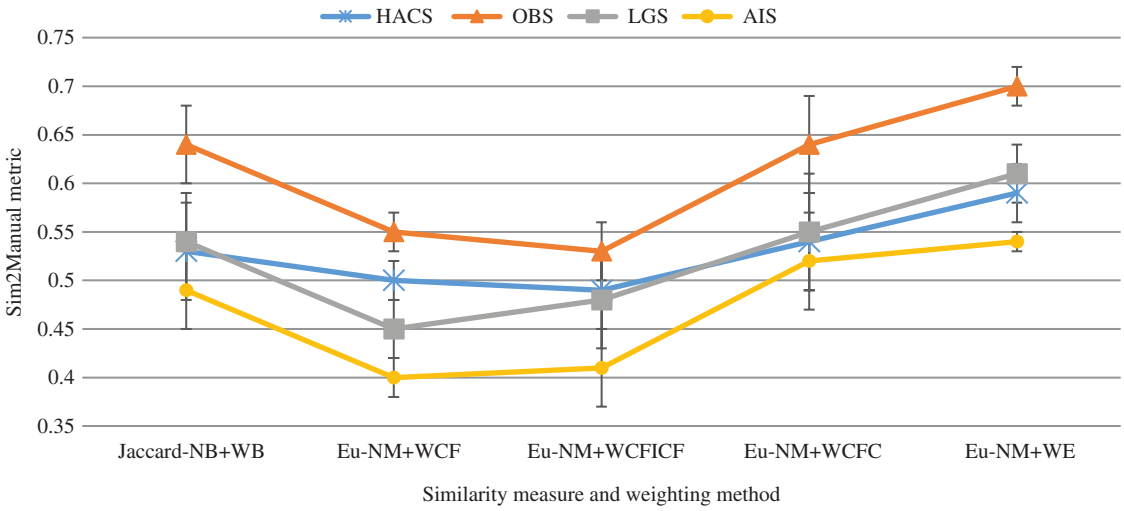

Figure 4: Results of PCI-PSO Without Interactive Iteration for the Four Cases (Average and Standard Deviation of 30 runs).

To find the best weighting method and similarity measure, 30 runs of the PCI-PSO without interactive iteration on the four real-world cases were performed. Note that PCI-PSO with all the weighting methods, including W_B, W_{CF}, W_{CFICF}, W_{CFC}, and W_E [21], and similarity measures, including Jaccard-NM [35] and E_u-NM [35], were separately applied on the four cases. Figure 4 shows the obtained results (i.e. average and standard deviation of 30 runs) of PCI-PSO with different used weighting methods and similarity measures for the four different case studies.

As shown in Figure 4, the $SimE_u$-NM similarity measure with the W_E weighting method identifies better solutions for all the four cases than other similarity measures and weighting methods with respect to the Sim2Manual metric. The reason for this is that the used entropy-weighting based (i.e. W_E) is able to present closer-to-reality connection strength between a pair of classes.

To compare PCI-PSO to other clustering-based component identification methods, we used SCI-GA [19], SBLCI [20], CCIC [21], neural network [23], and hybrid methods [1] as clustering-based component identification methods. It is worth mentioning that the SCI-GA [19], SBLCI [20], neural network [23], and hybrid methods [1] methods generally employ use cases as inputs, but to compare to PCI-PSO in this experiment they employ analysis classes as inputs. Furthermore, like PCI-PSO and [21], in this experiment, they employ W_E weighting method, $SimE_u$-NM similarity measure, and the DBI criterion as their fitness function. These methods also run 10 times and the best-obtained results are reported in Table 4.

As shown in Table 4, although the compared methods, especially neural network [23], hybrid methods [1], SCI-GA [19], and SBLCI [20], achieve the suitable values for the DBI criterion, the obtained solutions are different from the expert's solutions. In fact, optimizing only the DBI criterion for a case study cannot guarantee the identification of desirable components from the expert's viewpoints. The reason for this is the fact that the DBI criterion does not consider software architect's preferences; therefore, the obtained results may have significant differences from the components derived by experts.

Comparing the results of Table 4 reveals that, for all the four cases, the results of the proposed PCI-PSO method are better than other the compared clustering algorithms with respect to the Sim2Manual metric. Therefore, comparing the results of Table 4 reveals that PCI-PSO improves the Sim2Manual metric for all the four cases in comparison to SCI-GA [19], SBLCI [20], CCIC [21], neural network [23], and hybrid methods [1].

It should be noted that two professional developers with an average of 6 years of experience were invited to participate in trials as human experts, and for each case study, one of these experts participates in the interaction iteration. The human time (HT) and human interaction iteration (HI) for each case study are shown in Table 4. As shown in Table 4, the numbers of interactions during the PCI-PSO process for all the four cases were 4, 6, 13, and 12 iterations and each human interaction is approximately 7 min. Figure 5 shows the

Table 4: Results of Neural Network, Hybrid, SCI-GA, SBLCI, CCIC, and PCI-PSO Methods for the Four Cases.

Method	Metric	Case study			
		HACS	OBS	LGS	AIS
Neural network [23]	Time (s)	11.2	17.21	34.1	49.3
	DBI	0.019	0.023	0.028	0.029
	Sim2Manual	0.56	0.58	0.49	0.52
Hybrid clustering and PSO [1]	Time (s)	241.4	290.1	481.7	753.2
	DBI	0.017	0.019	0.021	0.023
	Sim2Manual	0.55	0.59	0.61	0.57
SCI-GA [19]	Time (s)	36.1	44.7	58.2	71.1
	DBI	0.021	0.025	0.027	0.031
	Sim2Manual	0.58	0.58	0.44	0.54
SBLCI [20]	Time (s)	49.1	58.3	78.2	112.1
	DBI	0.018	0.021	0.024	0.028
	Sim2Manual	0.58	0.61	0.53	0.55
CCIC [21]	Time (s)	1891.2	3055.6	11231.7	15782.1
	DBI	0.020	0.029	0.023	0.027
	Sim2Manual	0.86	0.96	0.84	0.86
PCI-PSO with interactive iteration	Time (s)	1855.1; HT = 1521, HI = 4	2756.2; HT = 2215, HI = 6	8242.6; HT = 5021, HI = 13	7012.1; HT = 5521, HI = 12
	DBI	0.021	0.026	0.021	0.022
	Sim2Manual	0.96	0.98	0.92	0.90
Experts have previously identified the components manually	DBI	0.033	0.039	0.044	0.049

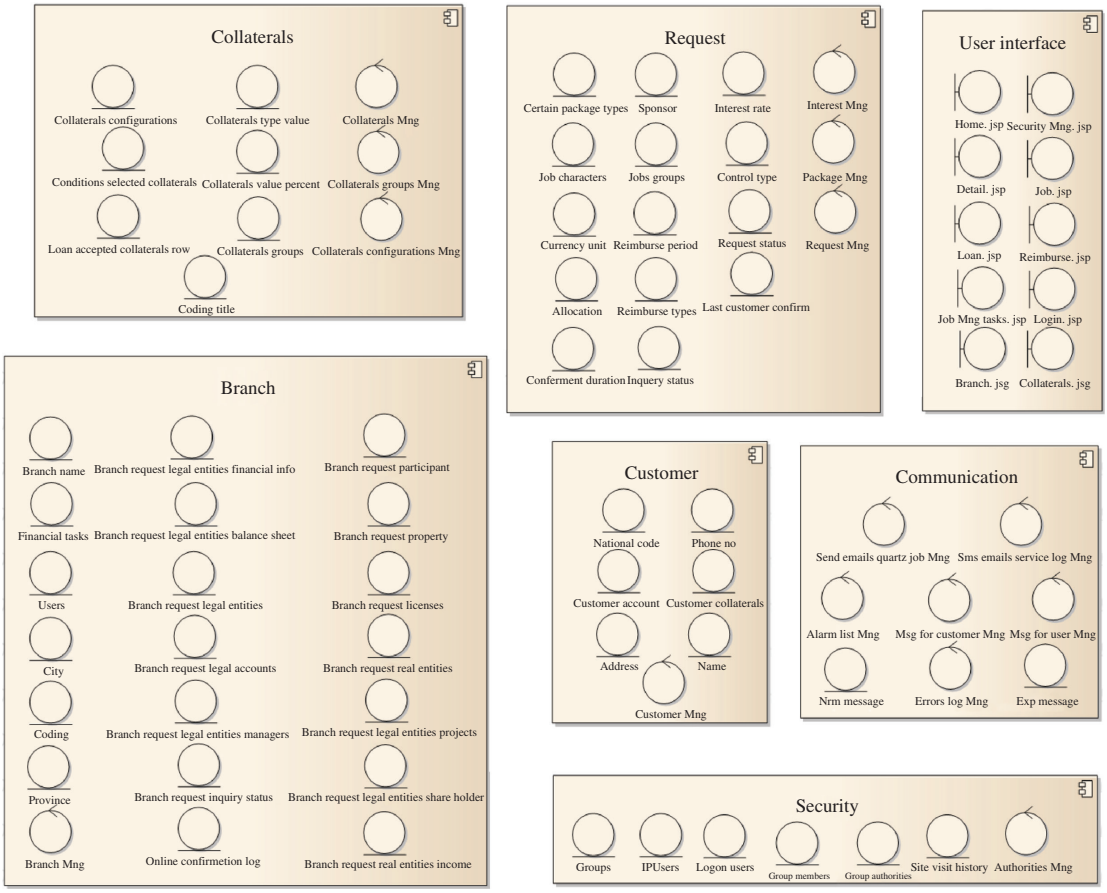

Figure 5: Components of the LGS Case Identified Automatically by PCI-PSO.

seven components identified automatically by PCI-PSO with human interactions for the LGS case, including User Interface, Collaterals, Branch, Request, Customer, Security, and Communication components.

Comparing the results of PCI-PSO and the results previously identified manually by experts shows that not only the results of PCI-PSO have the most similarity to the ones by experts but also its results have better DBI (i.e. high cohesion and low coupling) than the ones by experts. The reason for this is that there are other issues in the component identification process that the human experts considered them. Therefore, it is concluded that there are a number of aspects related to the component identification, which can hardly be considered during the identification process, such as the understandability of each obtained component. For this reason, PCI-PSO with the human interactions provides facilities for the software architects to explicitly interfere in the component identification process. Consequently, according to the obtained results presented in Table 4, when PCI-PSO employs human interactions, it identifies the most similar components to the ones identified manually by experts, and its obtained components can be better than the ones identified manually with respect to the DBI criterion.

Note that one of the limitations of PCI-PSO is that, when it employs human interactions, the run time of the proposed PCI-PSO method is greatly increased according to Table 4. To address this issue, we are going to propose an adaptive fitness penalty as a cheaper way to avoid the infeasible. However, it is worth remembering that, to identify logical components at an early stage of software design, it is not necessary to have a real-time method. Therefore, it seems likely that the run time of the proposed PCI-PSO method is tolerable.

6 Conclusions

In this paper, we proposed a preference-based method by the name of PCI-PSO, which is based on CPSOII and APSO, to identify logical components. To achieve the best values for PCI-PSO parameters, 30 runs of PCI-PSO parameters with different values for α, r_1, and r_2 were performed and the best values were chosen (according to Figure 2 and Tables 3 and 4). PCI-PSO provides the facilities for the software architects to play explicitly a role in the component identification process. In this situation, the software architects can interact with the CPSOII as human interactions. The experiments performed in Section 5 (see Table 4) revealed that, when PCI-PSO employs human interactions, it identifies the most similar components to the ones identified manually by experts, and its obtained components can be better than the ones identified manually with respect to the DBI criterion. We practically compared PCI-PSO to the clustering-based methods such as SCI-GA [19], SBLCI [20], CCIC [21], neural network [23], and hybrid methods [1] (see Table 4), and the results revealed that the clustering-based methods identify poor logical components that have significant differences from components derived by experts, because like other existing methods they do not take software architect or project preferences into account as opposed to the proposed PCI-PSO method.

For future work, after obtaining automatically cohesive subcomponents from PCI-PSO according to the responsibility of each subcomponent, we intend to use a suitable design pattern or library routine selection [17, 18]. Accordingly, an analysis model of a system can be mapped accurately to design pattern-based source codes [10].

Bibliography

[1] M. Ahmadzadeh, G. Shahmohammadi and M. Shayesteh, Identification of software systems components using a self-organizing map competitive artificial neural network based on cohesion and coupling, *ANDRIAS J.* **40** (2016), 60–71.
[2] AIS case, Available at: http://www.aiiri.gov.ir/HomePage.aspx?TabID=1&Site=aiiriPortal&Lang=en-US, Accessed 2016 Aug.
[3] A. Albani, S. Overhage and D. Birkmeier, Towards a systematic method for identifying business components, in: *International Symposium on Component-Based Software Engineering*, pp. 262–277, Springer, Berlin/Heidelberg, 2008.
[4] A. Alfi and M. M. Fateh, Intelligent identification and control using improved fuzzy particle swarm optimization, *Expert Syst. Appl.* **38** (2011), 12312–12317.

[5] A. Alfi and H. Modares, System identification and control using adaptive particle swarm optimization, *Appl. Math. Modell.* **35** (2011), 1210–1221.
[6] K. Ameli, A. Alfi and M. Aghaebrahimi, A fuzzy discrete harmony search algorithm applied to annual cost reduction in radial distribution systems, *Eng. Optim.* **48** (2016), 1529–1549.
[7] D. Birkmeier and S. Overhage, On component identification approaches – classification, state of the art, and comparison, in: *International Symposium on Component-Based Software Engineering*, pp. 1–18, Springer, Berlin/Heidelberg, 2009.
[8] D. Q. Birkmeier and S. Overhage, A method to support a reflective derivation of business components from conceptual models, *Inf. Syst. e-Bus. Manag.* **11** (2013), 403–435.
[9] Z.-g. Cai, X.-h. Yang, X.-y. Wang and A. Kavs, A fuzzy-based approach for business component identification, *J. Zhejiang Univ.-Sci. C* (Comput. Electron.) **12** (2011), 707–720.
[10] A. Chihada, S. Jalili, S. M. H. Hasheminejad and M. H. Zangooei, Source code and design conformance, design pattern detection from source code by classification approach, *Appl. Soft Comput.* **26** (2015), 357–367.
[11] J. F. Cui and H. S. Chae, Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems, *Inf. Softw. Technol.* **53** (2011), 601–614.
[12] D. L. Davies and D. W. Bouldin, A cluster separation measure, *IEEE Trans. Pattern Anal. Mach. Intell.* **2** (1979), 224–227.
[13] R. Ganesan and S. Sengupta, O2BC: a technique for the design of component-based applications, in: *Technology of Object-Oriented Languages and Systems, 2001, TOOLS 39, 39th International Conference and Exhibition on*, pp. 46–55, IEEE, Santa Barbara, CA, USA, 2001.
[14] HACS case, Available at: http://www.coursehero.com/file/3666313/HACSFINAL/, Accessed 2016 Aug.
[15] H. S. Hamza, A framework for identifying reusable software components using formal concept analysis, in: *Information Technology: New Generations, 2009, ITNG'09, 6th International Conference on*, pp. 813–818, IEEE, Las Vegas, NV, USA, 2009.
[16] H. Hashemi, G. Shahmohammadi and M. Shayesteh, Detection system software components using a hybrid algorithm, *ANDRIAS J.* **40** (2015), 57–63.
[17] S. H. A. Hasheminejad and S. M. H. Hasheminejad, Traffic accident severity prediction using a novel multi-objective genetic algorithm, *Int. J. Crashworthiness* **22** (2017), 425–440.
[18] S. M. H. Hasheminejad and S. Jalili, Selecting proper security patterns using text classification: in: *Computational Intelligence and Software Engineering, 2009, CiSE 2009, International Conference on*, pp. 1–5, IEEE, Wuhan, China, 2009.
[19] S. M. H. Hasheminejad and S. Jalili, Design patterns selection: an automatic two-phase method, *J. Syst. Softw.* **85** (2012), 408–424.
[20] S. M. H. Hasheminejad and S. Jalili, SCI-GA: software component identification using genetic algorithm, *J. Object Technol.* **12** (2013), 3-1.
[21] S. M. H. Hasheminejad and S. Jalili, An evolutionary approach to identify logical components. *J. Syst. Softw.* **96** (2014), 24–50.
[22] S. M. H. Hasheminejad and S. Jalili, CCIC: clustering analysis classes to identify software components, *Inf. Softw. Technol.* **57** (2015), 329–351.
[23] S. M. Hasheminejad and Z. Salimi, FDiBC: a novel fraud detection method in bank club based on sliding time and scores window, *J. AI Data Mining* (2017).
[24] S. H. A. Hasheminejad, M. Zahedi and S. M. H. Hasheminejad, A hybrid clustering and classification approach for predicting crash injury severity on rural roads, *Int. J. Injury Control Saf. Promot.* (2017), 1–17.
[25] H. Jain, N. Chalimeda, N. Ivaturi and B. Reddy, Business component identification – a formal approach, in: *Enterprise Distributed Object Computing Conference, 2001, EDOC'01, Proceedings, 5th IEEE International*, pp. 183–187, IEEE, Seattle, WA, USA, 2001.
[26] S. D. Kim and S. H. Chang, A systematic method to identify software components, in: *Software Engineering Conference, 2004, 11th Asia-Pacific*, pp. 538–545, IEEE, 2004.
[27] J. Kim, S. Park and V. Sugumaran, DRAMA: a framework for domain requirements analysis and modeling architectures in software product lines, *J. Syst. Softw.* **81** (2008), 37–55.
[28] S. D. Lee, Y. J. Yang, F. S. Cho, S. D. Kim and S. Y. Rhew, COMO: a UML-based component development methodology, in: *Software Engineering Conference, 1999, (APSEC'99) Proceedings, 6th Asia Pacific*, pp. 54–61, IEEE, Takamatsu, Japan, 1999.
[29] J. K. Lee, S. J. Jung, S. D. Kim, W. H. Jang and D. H. Ham, Component identification method with coupling and cohesion, in: *Software Engineering Conference, 2001, APSEC 2001, 8th Asia-Pacific*, pp. 79–86, IEEE, Macao, China, 2011.
[30] LGS case, Available at: https://taam.bankmellat.ir/TAAM/jsp/customer/main.jsp, Accessed 2016 Aug.
[31] H. Masoud, S. Jalili and S. M. H. Hasheminejad, Dynamic clustering using combinatorial particle swarm optimization, *Appl. Intell.* **38** (2013), 289–314.
[32] G. A. Mills-Tettey, A. Stentz and M. B. Dias, The dynamic Hungarian algorithm for the assignment problem with changing costs, in: *Tech. Rep. CMU-RI-TR-07-27*, Robotics Institute, Pittsburgh, 2007.
[33] R. Naseem, O. Maqbool and S. Muhammad, Improved similarity measures for software clustering, in: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 45–54, IEEE, Oldenburg, Germany, 2011.
[34] OBS case, Available at: http://www.isr.umd.edu/~austin/ense621.d/projects04.d/project_gouthami.html, Accessed 2016 Aug.

[35] S. M. A. Pahnehkolaei, A. Alfi, A. Sadollah and J. H. Kim, Gradient-based water cycle algorithm with evaporation rate applied to chaos suppression, *Appl. Soft Comput.* **53** (2017), 420–440.
[36] L. Peng, Z. Tong and Y. Zhang, Design of business component identification method with graph segmentation, in: *Intelligent System and Knowledge Engineering, 2008, ISKE 2008, 3rd International Conference on*, Vol. 1, pp. 296–301, IEEE, Xiamen, China, 2008.
[37] J. Rice, *Mathematical statistics and data analysis*, Nelson Education, Scarborough, 2006.
[38] F. Sebastiani, Machine learning in automated text categorization, *ACM Comput. Surv.* **34** (2002), 1–47.
[39] G. Shahmohammadi, S. Jalili and S. M. H. Hasheminejad, Identification of system software components using clustering approach, *J. Object Technol.* **9** (2010), 77–98.
[40] M. Shepperd, *Foundations of software measurement*, Prentice-Hall International (UK) Ltd., London, 1995.
[41] V. Tawosi, S. Jalili and S. M. H. Hasheminejad, Automated software design using ant colony optimization with semantic network support. *J. Syst. Softw.* **109** (2015), 1–17.
[42] R. Xu and D. C. Wunsch, Clustering algorithms in biomedical research: a review, *IEEE Rev. Biomed. Eng.* **3** (2010), 120–154.