CCIC: Clustering Analysis Classes to Identify Software Components

Article *in* Information and Software Technology · June 2014

DOI: 10.1016/j.infsof.2014.05.013

CITATIONS
19

READS
235

2 authors:

Seyed Mohammad Hossein Hasheminejad
Alzahra University
21 PUBLICATIONS **289** CITATIONS
SEE PROFILE

Saeed Jalili
Tarbiat Modares University
107 PUBLICATIONS **1,065** CITATIONS
SEE PROFILE

Information and Software Technology 57 (2015) 329–351

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

CCIC: Clustering analysis classes to identify software components

S.M.H. Hasheminejad, S. Jalili *

Department of Computer Engineering, Tarbiat Modares University, Tehran, Iran

ARTICLE INFO

Article history:
Received 19 June 2013
Received in revised form 19 May 2014
Accepted 19 May 2014
Available online 20 June 2014

Keywords:
Component identification
Software architect's preferences
Hierarchical evolutionary algorithm

ABSTRACT

Context: Component identification during software design phase denotes a process of partitioning the functionalities of a system into distinct components. Several component identification methods have been proposed that cannot be customized to software architect's preferences.
Objectives: In this paper, we propose a clustering-based method by the name of CCIC (*Clustering analysis Classes to Identify software Components*) to identify logical components from analysis classes according to software architect's preferences.
Method: CCIC uses a customized HEA (*Hierarchical Evolutionary Algorithm*) to automatically classify analysis classes into appropriate logical components and avoid the problem of searching for the proper number of components. Furthermore, it allows software architects to determine the constraints in their deployment and implementation framework.
Results: A series of experiments were conducted for four real-world case studies according to various proposed weighting schemes.
Conclusion: According to experimental results, it is concluded that CCIC can identify more cohesive and independent components with respect to software architect's preferences in comparison with the existing component identification methods such as *FCA*-based and *CRUD*-based methods.

1. Introduction

Component identification process is an important task to identify software architecture for a given system. There are two ways to achieve software components: (1) Component selection and (2) Component identification. In component selection methods, the goal is to provide the suitable existing components matching software requirements [1]. On the contrary, in the component identification during software design phase, the goal is to partition functionalities of a given system into non-intersecting logical components to provide the starting point for designing software architecture [2]. This paper has focused on the component identification process during software design phase. These components have two properties: (1) homogeneity within a component, i.e., the functionalities belonging to a component should be as similar as possible and (2) heterogeneity between the components, i.e., the functionalities of one component should be distinct from those belonging to other components.

A variety of methods have been proposed for identifying components during software design phase [3–15]. We summarize these methods into four categories: *Graph Partitioning*, *Clustering-based*, *CRUD-based* (Create, Read, Update, and Delete matrices) and *FCA-based* (*Formal Concept Analysis*) methods. At first, all these methods map functionalities of a system to a model, and then try to partition the functionalities into some components. To model functionalities of a given system, *Graph Partitioning*, *Clustering-based*, *CRUD-based* and *FCA-based* methods used graphs, feature vectors, matrixes and lattices, respectively. However, the existing methods have several drawbacks. First, most of them require manual threshold adjustment of their parameters, and highly depend on expert judgment to select the best solution. Second, in most of them, the number of components must be manually determined by experts in advance. Third, 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 [16]. Fourth, except for [4], they cannot use legacy components that are implemented in previous projects, and are reusable in the current project. Finally, they cannot identify components according to software architect's preferences (e.g., deployment and implementation framework constraints), and they only introduce completely automated procedures, which cannot be influenced by software architects.

In [15], we proposed a clustering-based method called SCI-GA, which is based on a genetic algorithm to identify software

* Corresponding author. Tel.: +98 (0)2182883374.
E-mail addresses: SMH.Hasheminejad@Modares.ac.ir (S.M.H. Hasheminejad), Sjalili@Modares.ac.ir (S. Jalili).

http://dx.doi.org/10.1016/j.infsof.2014.05.013
0950-5849/

components from use case models of a given system. The main drawback of SCI-GA [15] is that it cannot identify components according to software architect's preferences, and use legacy components. To address these drawbacks, in this paper, we propose a novel clustering-based method, called CCIC (*Clustering analysis Classes to Identify software Components*), to identify logical components from analysis classes according to software architect's preferences. The main contribution of CCIC is successful application of HEA (*Hierarchical Evolutionary Algorithm*) [17] in logical component identification with respect to software architect or project preferences, such as legacy components as well as deployment and implementation framework constraints of a given system. Additionally, in contrast to the majority of existing methods, CCIC does not require the determination of the number of software components in advance.

It should be noted that CCIC uses analysis classes in addition to use case models, which are inputs for SCI-GA [15]. In fact, using analysis classes of a given system in CCIC provides the capacity to meet software architect or project preferences in the logical component identification.

In CCIC, it is necessary to find a way to calculate the connection strength between a pair of classes. Therefore, we have proposed five weighing methods to calculate the connection strength between a pair of classes, and have performed a series of experiments to examine their effectiveness. For justification, we evaluated CCIC using four real-world cases, and the obtained results were analyzed and discussed by comparison with the other methods.

In the next section, an overview on HEA is provided as background. Then, Section 3 describes the inputs of logical component identification methods with more details. Section 4 illustrates the proposed weighting methods and similarity measures for computing similarity between a pair of classes. In Section 5, the proposed CCIC algorithm is described in detail, and in Section 6, we have evaluated CCIC using four real-world cases. After comparing CCIC with the current state of the art methods and describing its limitations in Section 7, Section 8 provides concluding remarks and future works.

2. Hierarchical evolutionary algorithm

The hierarchical evolutionary algorithm can be regarded as a variant of conventional genetic algorithm (GA). Genetic algorithms were invented by John Holland in 1970s [18]. In these algorithms, there are several terminologies, which are clarified below. A chromosome is a set of genes and shows a solution to the particular problem. Each gene presents a particular property of the solution. The set of chromosomes at a given time is called a population. An evolutionary process is iteratively executed, initialized by a randomly chosen population. Each iteration is called a generation. At each generation, such evolution is achieved by applying mutation and crossover operators between different chromosomes. The next generation of population includes the fittest parent or offspring chromosomes, and is done via a replacement scheme. Finally, the evolutionary process terminates when a population satisfies several predetermined conditions.

In HEA [17], each chromosome consists of two types of genes: control and parametric genes. Control genes determine which parametric gene should be utilized and which one can be disabled. In general, the value of each control gene is binary, while the value of each parametric gene is an integer or real number. In fact, the goal of control genes is to enable or disable the parametric genes. If the value of a control gene equals to 1, then, the associated parametric genes are enabled; otherwise, the other ones are disabled.

Fig. 1 shows two examples of HEA chromosome presentation. In Fig. 1(a), a two-level chromosome presentation is shown, in which the genes 12, 21, and 14 are enabled, and the others are disabled. In Fig. 1(b), a three-level chromosome presentation is shown, in which the second-level control genes manage the parametric genes and they are managed by the first-level control genes. Therefore, only the gene 12 is enabled, because the genes 5, 14, and 21 are disabled according to the gene 0 of the first-level control genes, and the genes 4 and 9 are disabled according to the genes 0 of the second-level control genes.

3. Inputs of logical component identification methods

The goal of any logical component identification method is to partition requirements of a system into meaningful units. In RUP methodology [19], requirements of a system are identified in "*Requirements Capture Workflow*", and are presented by use case model. Use case model consists of some use cases and actors. In RUP methodology, the identified use cases are described with more details in "*Analysis Workflow*", after capturing the use cases. One of the important artifacts in "*Analysis Workflow*" is analysis class diagram. In fact, for each use case, an analysis class diagram is created. Each analysis class diagram consists of three types of classes: boundary (interface), control, and entity (data) classes. In RUP methodology, components of a system are determined in "*Design Workflow*". Each component consists of a number of classes with their constraints and some interfaces. For example, one of these constraints is that each component must have at least one active class, i.e., control or boundary classes. Fig. 2 shows the RUP artifacts related to logical component identification along with their RUP workflow.

In the literature, two resources are used to identify logical components: use cases and analysis classes. In the existing components identification methods, analysis classes are used more than use cases. The reason for this is that use cases are always changed in generation of a system, but analysis classes, especially entity classes, are less variable than use cases.

In this paper, inputs of the proposed CCIC algorithm are analysis class diagrams and collaboration diagrams of a system. Note that the goal of CCIC is to partition analysis classes of a system into cohesive and independent units, called logical components, which are shown in Fig. 2.

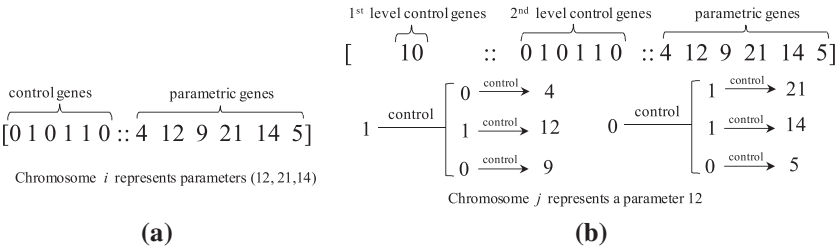


Fig. 1. Two examples for HEA chromosome presentation.

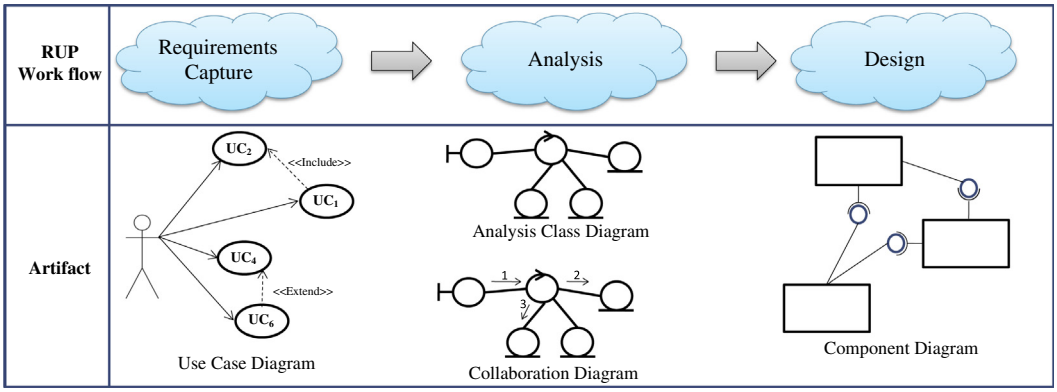


Fig. 2. RUP artifacts related to component identification methods.

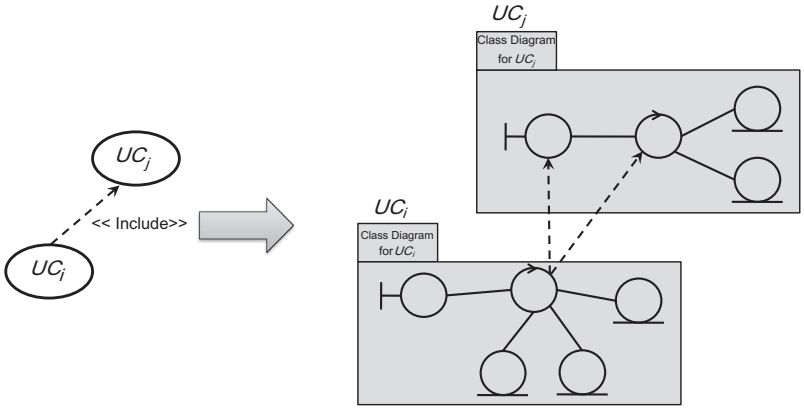


Fig. 3. An example of enhancing analysis class diagrams by use case diagrams.

It is worth mentioning that CCIC employs use case diagrams indirectly, in which ≪Include≫ and ≪Extend≫ relationships of each use case diagram are converted into two relationships between control and boundary classes in analysis class diagrams. For example, in Fig. 3, when UC_i includes UC_j in a part of the use case diagram, it means that the corresponding control class of UC_i calls either the corresponding control or boundary classes of UC_j. Therefore, in CCIC, for each relationship between two use cases, such as ≪Include≫ and ≪Extend≫, two relationships between their control and boundary classes are added in analysis class diagrams like Fig. 3.

4. Similarity measures for classes

As mentioned earlier, in CCIC, it is necessary to have a similarity measure, which determines how similar a pair of classes are. To apply each similarity measure, we can use weighting methods for finer tuning of similarity measures. Therefore, in this section, we propose five types of weighting methods for defining connections between classes, and then, illustrate the similarity measures.

4.1. Weighting methods

We propose five weighting methods to calculate the connection strength between a pair of classes. These weighting methods are inspired by weighting methods presented in text mining concepts [20], including *Binary weighting method* (W_B), *Term Frequency weighting method* (W_{TF}), *Term Frequency Inverse Document Frequency weighting method* (W_{TFIDF}), *Term Frequency Collection weighting method* (W_{TFC}), and *Entropy weighting method* (W_E). These weighting methods are described in detail below.

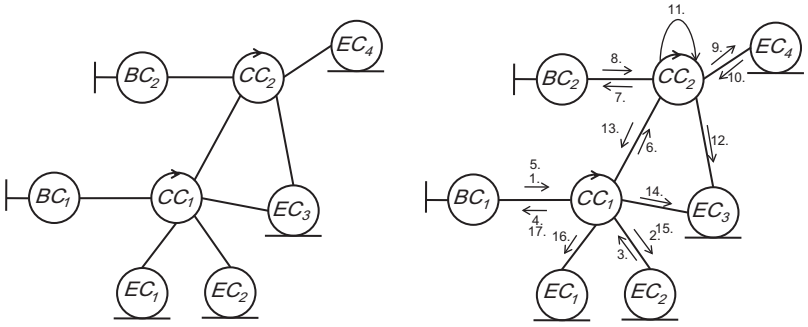
In this section, the following notations are used to describe weighting methods:

- Let SC denote the set of classes in a software system S; then, $|SC|$ is the number of classes of S.
- Let $connection(C_i, C_j)$ denote the set of connections between classes C_i and C_j, where C_i is the accessing class and C_j is the accessed class.
- Let n_{C_i} denote the number of classes connected to C_i.
- Let f_{C_i,C_j} denote the calling frequency of C_i in C_j, which is equal to $\frac{|connection(C_i,C_j)|}{|totalaccess(C_j)|}$, where $totalaccess(C_j)$ is the set of connections in which C_j is accessed (i.e., the number of methods calling C_j).

For example, if we take Fig. 4(b) as an example: $|connection(CC_1, EC_2)| = 2$, $|connection(EC_2, CC_1)| = 1$, $n_{CC_1} = 5$, $|totalaccess(CC_1)| = 3$, and $f_{EC_2,CC_1} = 1/3$. It should be noted that the relationship of a class to itself does not affect the proposed component identification; therefore, in all the proposed weighting methods, these relationships are not considered.

4.1.1. Binary weighting method (W_B)

The *Binary weighting method* is the simplest way to show the connection between a pair of classes. In this method, the value of 1 denotes connection between the two classes. In Eq. (1), W_B denotes the connection strength between both C_i and C_j classes. Fig. 5(a) shows all W_B values for the analysis model presented in Fig. 4.



(a) Analysis Class Diagram (b) Collaboration Diagram

Fig. 4. An example of analysis model.

W_B	BC_1	BC_2	CC_1	CC_2	EC_1	EC_2	EC_3	EC_4
BC_1		0.00	1.00	0.00	0.00	0.00	0.00	0.00
BC_2	0.00		0.00	1.00	0.00	0.00	0.00	0.00
CC_1	1.00	0.00		1.00	1.00	1.00	1.00	0.00
CC_2	0.00	1.00	1.00		0.00	0.00	1.00	1.00
EC_1	0.00	0.00	1.00	0.00		0.00	0.00	0.00
EC_2	0.00	0.00	1.00	0.00	0.00		0.00	0.00
EC_3	0.00	0.00	1.00	1.00	0.00	0.00		0.00
EC_4	0.00	0.00	0.00	1.00	0.00	0.00	0.00	

(a) W_B

W_{TF}	BC_1	BC_2	CC_1	CC_2	EC_1	EC_2	EC_3	EC_4
BC_1		0.00	1.50	0.00	0.00	0.00	0.00	0.00
BC_2	0.00		0.00	1.25	0.00	0.00	0.00	0.00
CC_1	1.50	0.00		0.50	1.00	1.25	0.50	0.00
CC_2	0.00	1.25	0.50		0.00	0.00	0.50	1.25
EC_1	0.00	0.00	1.00	0.00		0.00	0.00	0.00
EC_2	0.00	0.00	1.25	0.00	0.00		0.00	0.00
EC_3	0.00	0.00	0.50	0.50	0.00	0.00		0.00
EC_4	0.00	0.00	0.00	1.25	0.00	0.00	0.00	

(b) W_{TF}

W_{TFIDF}	BC_1	BC_2	CC_1	CC_2	EC_1	EC_2	EC_3	EC_4
BC_1		0.00	1.01	0.00	0.00	0.00	0.00	0.00
BC_2	0.00		0.00	0.98	0.00	0.00	0.00	0.00
CC_1	1.01	0.00		0.13	0.90	0.95	0.30	0.00
CC_2	0.00	0.98	0.13		0.00	0.00	0.30	0.98
EC_1	0.00	0.00	0.90	0.00		0.00	0.00	0.00
EC_2	0.00	0.00	0.95	0.00	0.00		0.00	0.00
EC_3	0.00	0.00	0.30	0.30	0.00	0.00		0.00
EC_4	0.00	0.00	0.00	0.98	0.00	0.00	0.00	

(c) W_{TFIDF}

W_{TFC}	BC_1	BC_2	CC_1	CC_2	EC_1	EC_2	EC_3	EC_4
BC_1		0.00	1.82	0.00	0.00	0.00	0.00	0.00
BC_2	0.00		0.00	1.50	0.00	0.00	0.00	0.00
CC_1	1.82	0.00		0.91	1.00	1.41	0.71	0.00
CC_2	0.00	1.50	0.91		0.00	0.00	0.71	1.50
EC_1	0.00	0.00	1.00	0.00		0.00	0.00	0.00
EC_2	0.00	0.00	1.41	0.00	0.00		0.00	0.00
EC_3	0.00	0.00	0.71	0.71	0.00	0.00		0.00
EC_4	0.00	0.00	0.00	1.50	0.00	0.00	0.00	

(d) W_{TFC}

W_E	BC_1	BC_2	CC_1	CC_2	EC_1	EC_2	EC_3	EC_4
BC_1		0.00	0.85	0.00	0.00	0.00	0.00	0.00
BC_2	0.00		0.00	0.76	0.00	0.00	0.00	0.00
CC_1	0.85	0.00		0.24	0.63	0.75	0.27	0.00
CC_2	0.00	0.76	0.24		0.00	0.00	0.27	0.76
EC_1	0.00	0.00	0.63	0.00		0.00	0.00	0.00
EC_2	0.00	0.00	0.75	0.00	0.00		0.00	0.00
EC_3	0.00	0.00	0.27	0.27	0.00	0.00		0.00
EC_4	0.00	0.00	0.00	0.76	0.00	0.00	0.00	

(e) W_E

Fig. 5. The values of different weighting methods for the analysis model presented in Fig. 4.

$$W_B(C_i,C_j)=\begin{cases}1 & \text{if } (|connection(C_i,C_j)|+|connection(C_j,C_i)|>0)\\ \text{Undefined} & \text{if } (i=j)\\ 0 & \text{otherwise}\end{cases} \quad (1)$$

4.1.2. Term Frequency weighting method (W_{TF})

The *Term Frequency weighting method* for a pair of classes is calculated according to summation of the calling frequency of both classes. Eq. (2) shows the formula to compute W_{TF}. Take Fig. 4(b) as an example: e.g., $W_{TF}(BC_1,CC_1)=f_{BC_1,CC_1}+f_{CC_1,BC_1}=\frac{|connection(BC_1,CC_1)|}{|totalaccess(CC_1)|}+\frac{|connection(CC_1,BC_1)|}{|totalaccess(BC_1)|}=\frac{2}{4}+\frac{2}{2}=1.5$. Fig. 5(b) shows all W_{TF} values for the analysis model presented in Fig. 4.

$$W_{TF}(C_i,C_j)=\begin{cases}\text{Undefined} & \text{if } (i=j)\\ f_{C_i,C_j}+f_{C_j,C_i} & \text{otherwise}\end{cases} \quad (2)$$

4.1.3. Term Frequency Inverse Document Frequency weighting method (W_{TFIDF})

The two weighting methods described do not take into account the number of classes statically connected to a class. Take Fig. 4(b) as an example. It is plausible that the connection between EC_2 and CC_1 is tighter than that between EC_4 and CC_2, but both W_B and W_{TF} cannot show these differences. W_{TFIDF} not only considers the use of calling frequency of a class, but also how frequently a class is connected to others in analysis class diagrams. Eq. (3) shows the formula to compute W_{TFIDF}. For example, $W_{TFIDF}(EC_2,CC_1)=f_{EC_2,CC_1}\times\log\frac{|SC|}{n_{CC_1}}+f_{CC_2,EC_2}\times\log\frac{|SC|}{n_{EC_2}}=0.95$ and $W_{TFIDF}(EC_4,CC_2)=f_{EC_4,CC_2}\times\log\frac{|SC|}{n_{CC_2}}+f_{CC_2,EC_4}\times\log\frac{|SC|}{n_{EC_4}}=0.98$. As can be seen, W_{TFIDF} is able to show this difference as opposed to both W_B and W_{TF}. Fig. 5(c) shows all W_{TFIDF} values for the analysis model presented in Fig. 4.

$$W_{TFIDF}(C_i,C_j)=\begin{cases}\text{Undefined} & \text{if } (i=j)\\ f_{C_i,C_j}\times\log\frac{|SC|}{n_{C_j}}+f_{C_j,C_i}\times\log\frac{|SC|}{n_{C_i}} & \text{otherwise}\end{cases} \quad (3)$$

4.1.4. Term Frequency Collection weighting method (W_{TFC})

W_{TFIDF} does not take into account the effect of calling frequencies of other classes on connection strength between a pair of classes. W_{TFC}, which is described in Eq. (4), is similar to the W_{TFIDF} except for the fact that the calling frequency normalization is used as part of the weighting method. Fig. 5(d) shows all W_{TFC} values for the analysis model presented in Fig. 4. Take Fig. 4 as an example. When classes CC_1, CC_2 and EC_3 are considered, it is plausible for class CC_1 to have a tighter relation with CC_2 than EC_3. However, none of W_B, W_{TF} and W_{TFIDF} according to Fig. 5(a), (b), and (c), respectively, presents this case. As shown in Fig. 5(d), W_{TFC} is able to show this tighter relation as opposed to the previous three weighting methods.

$$W_{TFC}(C_i,C_j)=\begin{cases}\text{Undefined} & \text{if } (i=j)\\ \frac{f_{C_i,C_j}\times\log\frac{|SC|}{n_{C_j}}}{\sqrt{\sum_{k=1}^{|SC|}\left(f_{C_k,C_j}\times\log\frac{|SC|}{n_{C_j}}\right)^2}}+\frac{f_{C_j,C_i}\times\log\frac{|SC|}{n_{C_i}}}{\sqrt{\sum_{k=1}^{|SC|}\left(f_{C_k,C_i}\times\log\frac{|SC|}{n_{C_i}}\right)^2}} & \text{otherwise}\end{cases} \quad (4)$$

4.1.5. Entropy weighting method (W_E)

Entropy-weighting is based on information theoretic ideas, and is the most effective weighting method in text mining [20]. In Eq. (5), the entropy-weighting method of text mining is customized to calculate the connection strength between a pair of classes. Fig. 5(e) shows all W_E values for the analysis model presented in Fig. 4.

$$W_E(C_i,C_j)=\begin{cases}\text{Undefined} & \text{if } (i=j)\\ \left(\log\left(f_{C_i,C_j}+1\right)\times\left(1+\frac{1}{\log|SC|}\times\sum_{k=1}^{|SC|}\frac{f_{C_k,C_j}}{n_{C_j}}\right)\right) & \\ +\left(\log(f_{C_j,C_i}+1)\times\left(1+\frac{1}{\log|SC|}\times\sum_{k=1}^{|SC|}\frac{f_{C_k,C_i}}{n_{C_i}}\right)\right) & \text{otherwise}\end{cases} \quad (5)$$

As shown in Fig. 5, the matrices of the proposed weighting methods are symmetric, and connections among analysis classes are considered as a directed graph. The reason for this is that if the asymmetric matrices are used, calculating similarity measures of a pair of classes becomes too complicated, a problem that is presented in clustering literature [21]. The weighting methods are believed to play an important role in component identification; therefore, to precisely evaluate the performance of those, some experiments will be performed in Section 6. Note that the summary of evaluations mentioned in Section 6.6 reveals that W_E weighting method outperforms the other ones in the identification of components.

4.2. Similarity measures

A feature vector can quantitatively represent an analysis class. In feature vector representation, each analysis class is represented by a number of properties. In this paper, each class is represented by a vector including its connection strengths with other classes. This connection strength can be calculated by either W_B, W_{TF}, W_{TFIDF}, W_{TFC} or W_E defined in Section 4.1. Take Fig. 5 as an example. For each class listed in the first column, the corresponding row presents the values of its feature vector. For example, the TFIDF feature vector of class CC_1 is (1.01, 0, 0.2, 0.13, 0.9, 0.95, 0.3, 0), which denotes that CC_1 has relationships with classes BC_1, EC_1, EC_2, EC_3 and CC_2 by the corresponding W_{TFIDF} values.

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 [16]. For this reason, seven popular similarity measures, including *Euclidean distance* [16], *Cosine similarity* [16], *Simple* [16], *Jaccard* [16], *Jaccard-NM* [21], *Unbiased Ellenberg* (E_u) [22], and *Unbiased Ellenberg-NM* (E_u-*NM*) [21] are used in this study, and in Section 6 their performance will be evaluated.

Before defining the similarity measures, it should be noted that according to each value type in feature vectors, a suitable similarity measure is chosen. Three are two value types in the proposed feature vectors: binary (W_B) and real (W_{TF}, W_{TFIDF}, W_{TFC} and W_E). In the following sections, *Simple*, *Jaccard* and *Jaccard-NM* similarity measures are defined for binary vectors, and *Euclidean distance*, *Cosine similarity*, E_u and E_u-*NM* similarity measures are defined for real vectors.

4.2.1. Similarity measures for binary vectors

For binary feature vectors, general similarity measures like *Euclidean distance*, should not be used, because their use leads to information loss [16]. Therefore, we use popular similarity measures, *Simple* and *Jaccard*, for binary vectors, i.e., W_B feature vectors for analysis classes. Eqs. (6) and (7) define *Simple* and *Jaccard* measures, respectively. Let n_{11} denote the number of features present in both classes C_i and C_j, n_{10} denote the number of features present in C_i but not C_j, n_{01} denote the number of features present in C_j but not C_i, and n_{00} denote the number of features not present in both C_i and C_j. Naseem et al. [21] proposed an extended *Jaccard* measure, called *Jaccard-NM* defined in Eq. (8), for software clustering. Their experiments revealed that *Jaccard-NM* measure is better than the other similarity measures for binary vectors in software domain. Therefore, we used it in this study to be compared with the other similarity measures. Note that the summary of evaluations

mentioned in Section 6.6 reveals that *Jaccard-NM* measure is better than the other similarity measures for binary vectors in the identification of components.

$$Sim_{Simple(C_i,C_j)} = \frac{n_{11} + n_{00}}{n_{11} + n_{10} + n_{01} + n_{00}} \quad (6)$$

$$Sim_{Jaccard(C_i,C_j)} = \frac{n_{11}}{n_{11} + n_{10} + n_{01}} \quad (7)$$

$$Sim_{Jaccard\text{-}NM(C_i,C_j)} = \frac{n_{11}}{2 \times (n_{11} + n_{10} + n_{01}) + n_{00}} \quad (8)$$

4.2.2. Similarity measures for real vectors

To calculate similarity between a pair of real vectors, we use two popular similarity measures: *Euclidean* and *Cosine Similarity*, which are defined in Eqs. (9) and (10), respectively, where x_k and y_k represent the values of C_i and C_j classes for the kth feature, respectively. It is valuable to point out that to compute similarity between a pair of classes according to *Euclidean distance*, after calculating *Euclidean distance* between them, the inverse value of *Euclidean distance* is considered as the similarity between them. The reason for this is that *Euclidean distance* has an opposite relation to similarity.

$$Sim_{Euclidean(C_i,C_j)} = 1/Distance_{Euclidean(C_i,C_j)} = 1/\sqrt{\sum_{k=1}^{|SC|} |x_k - y_k|^2} \quad (9)$$

$$Sim_{Cosine(C_i,C_j)} = \frac{\sum_{k=1}^{|SC|} x_k \times y_k}{\sqrt{\sum_{k=1}^{|SC|} x_k^2} \times \sqrt{\sum_{k=1}^{|SC|} y_k^2}} \quad (10)$$

Another similarity measure frequently used in clustering concepts is *Unbiased Ellenberg* (E_u) [22] defined in Eq. (11). Naseem et al. [21] also customized E_u for software clustering, which is called E_u-*NM*. Eq. (12) defines E_u-*NM* similarity measure, where Mn_{11} represents the sum of features present in both classes of C_1 and C_2. We used E_u and E_u-*NM* similarity measures in addition to *Euclidean* and *Cosine Similarity* to calculate similarity measures between a pair of real vectors, i.e., W_{TF}, W_{TFIDF}, W_{TFC} and W_E feature vectors. Note that the summary of evaluations mentioned in Section 6.6 reveals that E_u-*NM* and *Cosine* similarity measures are better than the other similarity measures for real vectors in the identification of components.

$$Sim_{E_u(C_i,C_j)} = \frac{0.5 \times Mn_{11}}{0.5 \times Mn_{11} + n_{10} + n_{01}} \quad (11)$$

$$Sim_{E_u\text{-}NM(C_i,C_j)} = \frac{0.5 \times Mn_{11}}{0.5 \times Mn_{11} + 2 \times (n_{10} + n_{01}) + n_{11} + n_{00}} \quad (12)$$

5. The CCIC algorithm

The logical component identification problem is a NP-complete problem [23]; therefore, the goal of this paper is to map the logical component identification problem to an optimization problem and solve this problem using the search-based approach. The reason is that search-based approaches are crucial alternatives to solve NP complete optimization problems that have achieved promising results [24]. Genetic algorithm is a popular search-based algorithm with robust and strong convergence. In this paper, we propose a customized HEA called CCIC to identify logical components. Our main motivation to use HEA instead of conventional GAs is the fact that the former is able to code parameters of the problem in a hierarchical and flexible structure. Therefore, it is more appropriate to solve the automatic logical component identification compared with the conventional GAs.

Fig. 6 shows the overall process of CCIC. The inputs of CCIC are an analysis model and its weighting matrices like Fig. 5 along with software architect or project preferences, and its outputs

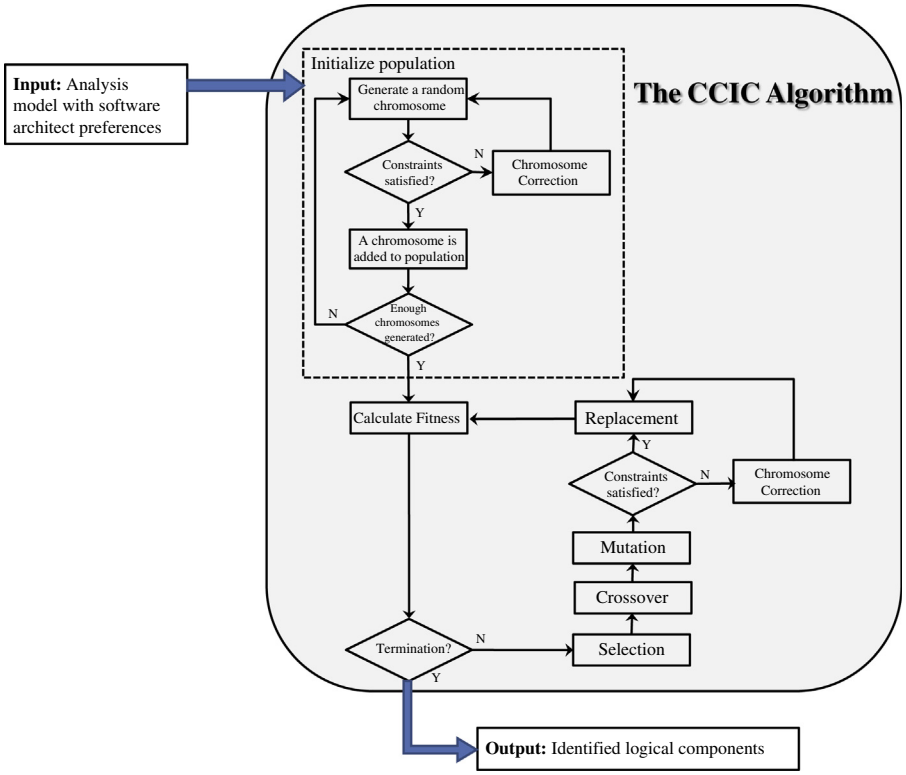


Fig. 6. Process of the CCIC algorithm.

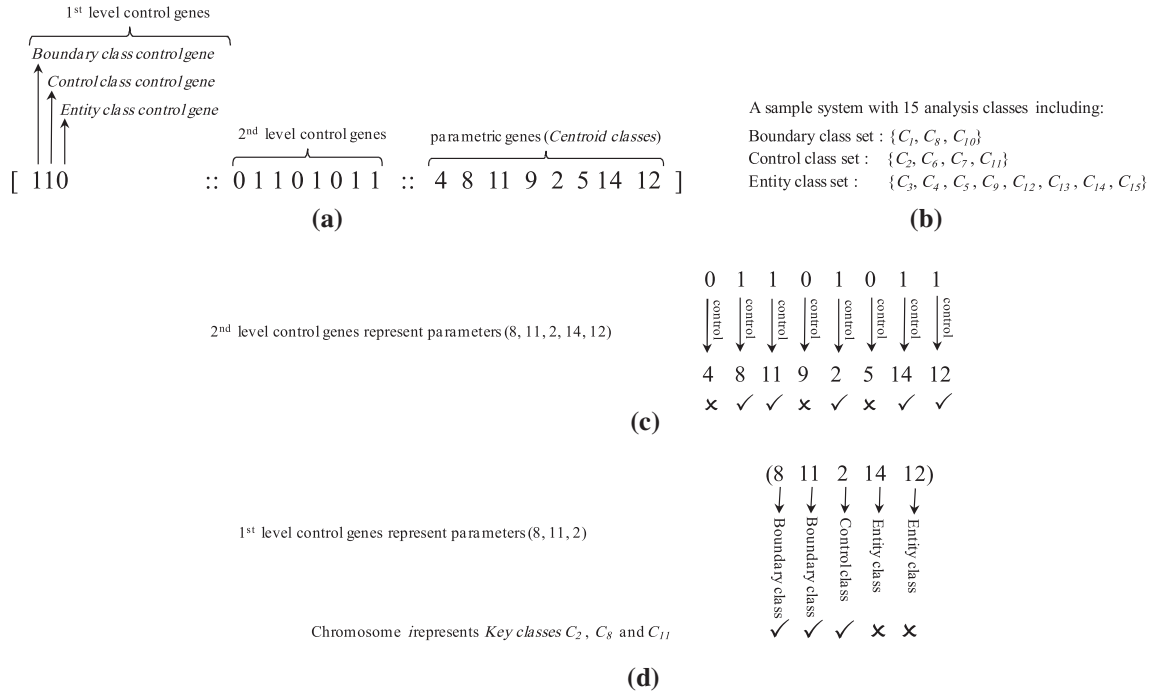


Fig. 7. An example of the hierarchical structure of chromosome in CCIC: (a) a sample chromosome, (b) a sample system as an input, (c) applying the 2nd level control genes on the parametric genes of the chromosome presented in (a), and (d) applying the 1st level control genes on the parametric genes along with key classes of the chromosome presented in (a).

are identified logical components. In CCIC, at first, the initial population is randomly generated, and if a generated chromosome does not satisfy constraints of the problem, it is corrected. After generating initial population, for each chromosome, the fitness is evaluated according to a fitness function, which will be defined in Section 5.2. Then, using the roulette wheel selection scheme, some chromosomes are selected for reproduction as parents [25]. After selecting some parent chromosomes, crossover operators are applied on all pairs of parents to generate two children. Then, the mutation operator is applied on each generated child. After applying the HEA operations, the consistency of each child is evaluated, and if it does not satisfy the constraints of the problem, the chromosome is corrected. Then, the least fit chromosomes in the existing population are replaced by the newly generated offspring. Now, the next generation of population is created; therefore, this process is repeated until the fittest chromosome satisfies some conditions or the maximum number of iterations is exceeded. The following sections describe the steps of CCIC.

5.1. Solution representation

To represent the output of CCIC, we must determine how to model logical components of a system. We represent the solution in two steps: *Chromosome Representation* and *Component Formation* steps. In the *Chromosome Representation* step, the structure of each chromosome is defined, and in the *Component Formation* step, the way to achieve the components presented by each chromosome is described. In the following sections, these steps are defined in detail.

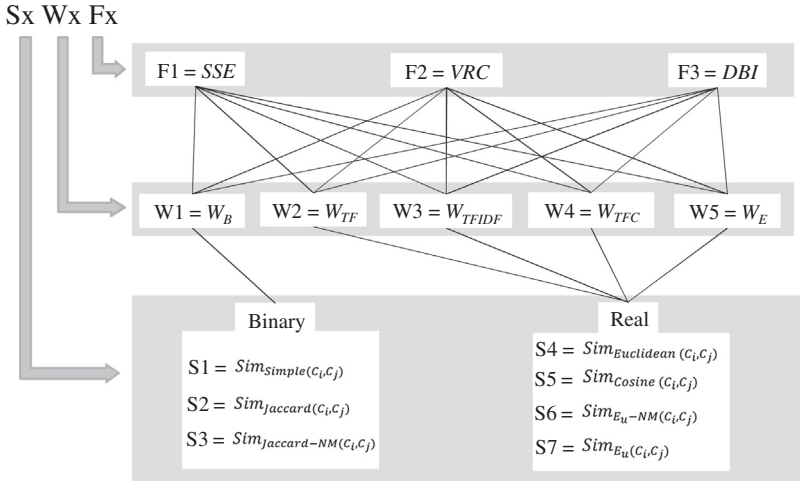


Fig. 8. Potential configurations of the CCIC algorithm.

* Suppose that Component j identified by CCIC is mapped to Component i identified by experts.

		Classes belonging to Component i identified by experts	
		Yes	No
Classes belonging to Component j identified by CCIC	Yes	TP (True Positive)	FP (False Positive)
	No	FN (False Negative)	TN (True Negative)

Fig. 9. Definitions of TP, FP, FN, and TN.

In [12], analysis classes of a system are clustered into logical components with high cohesion and low coupling. In this method, at first, key classes are selected as candidate components; then, other classes are assigned to components that have the most dependency with them. However, the main problem of the method presented in [12] is that identifying the key classes is a critical and difficult task, and is manually determined by experts. In this study, we use the key classes' idea and try to relax its difficulty. The goal of CCIC is to search suitable key classes among the analysis classes and automatically find the better ones to omit the role of experts in key class determination.

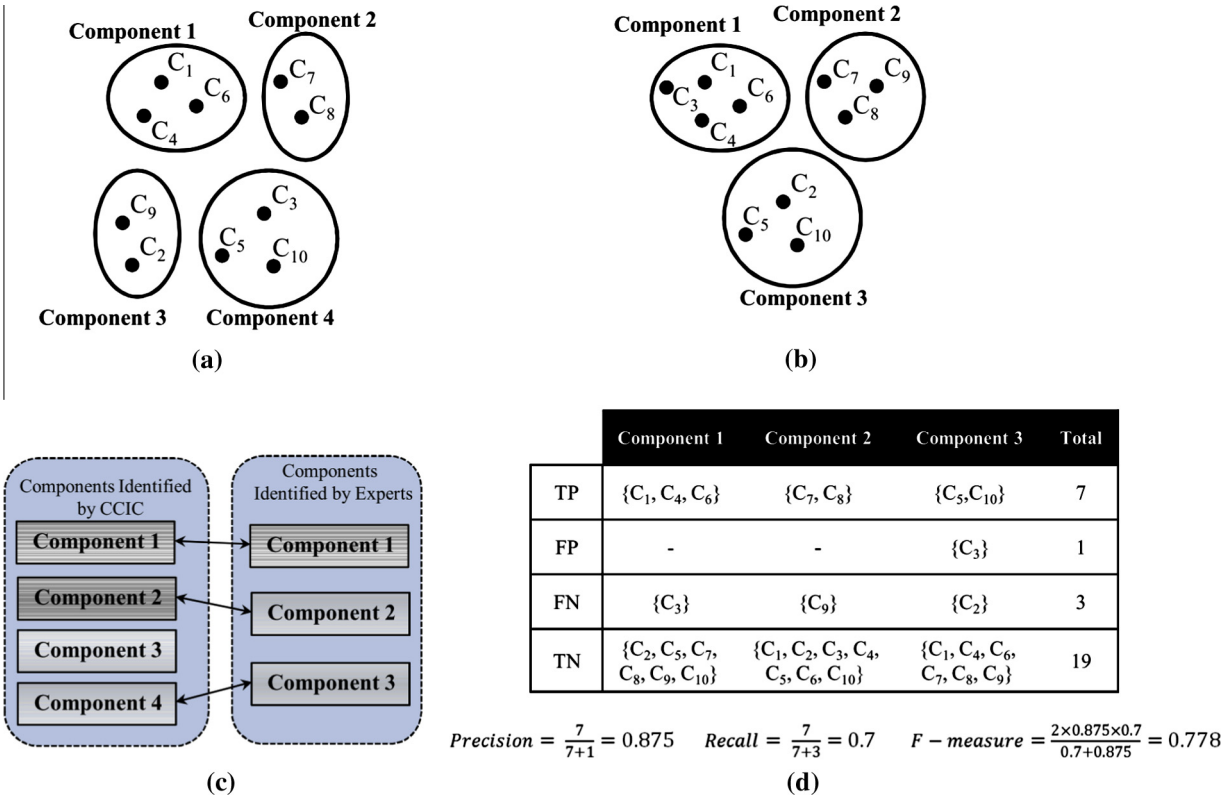
5.1.1. Chromosome representation

In CCIC, the chromosome is made up of two levels of control genes (representing in binary) as well as parametric genes (representing in integer). In this chromosome representation, each parametric gene denotes the analysis class number that can be a candidate key class. In addition, control genes are divided into two levels: the 1st level control genes denote three binary digits: the first digit is concerned to the boundary classes, the second is concerned to the control classes, and the third is concerned to the entity classes. Indeed, these three digits provide some constraints that can be used to determine which analysis class type should be considered as a key class. The 1 entry denotes that the candidate key classes presented by parametric genes can belong to that analysis class type and the 0 entry otherwise. The 2nd level control genes manage the parametric genes and have a similar role to the one for the common HEA introduced in Section 2. Note that the length of the 2nd level control genes and the length of the parametric genes are the same, and are equal to the maximum number of candidate key classes.

An example of the hierarchical chromosome structure in CCIC is illustrated in Fig. 7, where Fig. 7(a and b) represent a sample chromosome and a sample system with 15 analysis classes, respectively. In Fig. 7(a), it is assumed that the maximum number of candidate key classes for the sample system presented in Fig. 7(b) is equal to 8. Fig. 7(c) shows which parametric genes are omitted when the 2nd level control genes are considered. As shown in Fig. 7(c), only parametric genes with 2, 8, 11, 12 and 14 class numbers are considered as key classes; therefore, the 2nd level control genes show that the number of chromosome components presented in Fig. 7(a) is at most 5 components. Fig. 7(d) also shows which parametric genes are omitted when the 1st level control genes are considered. As shown in Fig. 7(a), only boundary and control classes can be considered as key classes; therefore, in Fig. 7(d), with respect to the 1st level control genes, the genes with 12 and 14 class numbers are omitted, because these are entity classes according to the input system presented in Fig. 7(b).

5.1.2. Component formation

As shown in Fig. 7, each chromosome shows some key classes, where each one presents one component. Therefore, to extract all classes of components of a system, it is necessary to associate other analysis classes to one of the identified key classes. To perform this



	Component 1	Component 2	Component 3	Total
TP	$\{C_1, C_4, C_6\}$	$\{C_7, C_8\}$	$\{C_5, C_{10}\}$	7
FP	-	-	$\{C_3\}$	1
FN	$\{C_3\}$	$\{C_9\}$	$\{C_2\}$	3
TN	$\{C_2, C_5, C_7, C_8, C_9, C_{10}\}$	$\{C_1, C_2, C_3, C_4, C_5, C_6, C_{10}\}$	$\{C_1, C_4, C_6, C_7, C_8, C_9\}$	19

$Precision = \frac{7}{7+1} = 0.875$ $Recall = \frac{7}{7+3} = 0.7$ $F-measure = \frac{2\times0.875\times0.7}{0.7+0.875} = 0.778$

(d)

Fig. 10. An example of components evaluation: (a) sample components identified by CCIC, (b) sample components identified by experts, (c) components mapping, and (d) metrics calculation.

task, a simple way is to associate each analysis class to the most similar key class. The disadvantage of this way is that CCIC cannot explore overall search space. Another way is to randomly associate each analysis class to a key class. The disadvantage of this way is that the similarities between classes do not have any effect on component formation. To extract components from a chromosome, in CCIC, each analysis class C_i is associated to $keyclass_c$ according to Eq. (13).

$$c = argmax\langle \forall keyclass_c (Sim(keyclass_c, C_i) + random(0, MaximumSimilarity))\rangle \quad (13)$$

where the *Sim* measurement is chosen among Eqs. (6)–(12) and *random* function denotes a random real number between 0 and the maximum value of similarity among all classes according to the measurement, which is chosen in the *Sim* measurement. It should be noted that due to the random function used in component formation step, different components can be extracted from a chromosome; therefore, in CCIC, since the component formation procedure is applied first, the extracted components with their members are stored in a link list data structure. In this link list data structure, each key class is the root of the link list and is linked to its members. For example, a sample component formation for Fig. 7 is {{**2**, 3, 4, 5, 10},{**8**, 13, 15},{**11**, 1, 6, 7, 9, 12, 14}}, where there are three components, in which analysis classes with 2, 8, and 11 class numbers are the key classes. It is worth mentioning that after performing the component formation step, the interface of each component consists of operations provided by its control and boundary classes.

5.2. Initial population

The HEA algorithm requires a population of potential solutions to be initialized at the beginning of the HEA algorithm. By default, the length of the 2nd level control and parametric genes are equal to the total number of input analysis classes, i.e., $|SC|$. However, CCIC allows software architects to determine the length of the 2nd level control, i.e., the maximum number of logical components, according to their preferences. Additionally, in CCIC, for initial chromosomes, the 1st level control genes are set according to software architect preferences (the default values of these three digits are 110, i.e., the entity classes cannot be key classes). Then, the number of 1's in the 2nd level control genes is uniformly distributed within $[1, k_{max}]$, where k_{max} is a user-defined maximum number of components (the default value of k_{max} is equal to $|SC|$), which can be determined by software architects. After setting the control genes for each chromosome, the values of the parametric genes are randomly selected within [1,|SC|], that each value shows the analysis class number of the input system.

5.3. Fitness function

There are a number of criteria to identify the suitable components. The coupling and cohesion aspects of software components are the fundamental quality attributes that heavily affect the software architecture, maintenance, evolution and reuse. Coupling criterion denotes heterogeneity between the components, i.e., the classes in a component show the least dependency with the other ones. Cohesion criterion denotes homogeneity within a component, i.e., the classes in a component should be as similar as possible. However, current studies [26,27] on component cohesion and coupling criteria show that there is no component cohesion measurement beginning with analysis class model. In this study, we use popular clustering criteria to evaluate each identified component solution, because similar to the component identification process, their goal is to identify condense and independent clusters, i.e., highly cohesive components and components with low coupling, respectively. 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 three popular clustering criteria: *Sum of Squared Error* (*SSE*) [16], *Variance Ratio Criterion* (*VRC*) [28] and *Davies–Bouldin Index* (*DBI*) [29], as fitness functions and examine them in Section 6.4. It should be noted that the definitions of these clustering criteria are reported in Appendix A with more details. Furthermore, summary of evaluations mentioned in Section 6.6 reveals that *DBI* metric is better than the other metrics in the identification of components.

5.4. Software architect or project preferences

It should be noted that the goal of automated logical component identification methods [3,6,7,11,12] like the three ones defined in Section 5.3 is to optimize the clustering fitness function in order to achieve the best possible solution. However, as mentioned earlier, software architect or project preferences have a strong impact on the performance of the component identification method, a fact not considered by current logical identification methods except for [4]. In CCIC, the software architect has a second possibility to vigorously affect the identification process, attempting to provide facilities to make use of the knowledge of software architects. In fact, CCIC addresses the three design concerns from software architect's viewpoint: deployment constraints, implementation framework constraints, and legacy components to be reused. Therefore, software architects have a powerful possibility to influence the automated identification outcome. In the following section, we propose some constraints to support software architect preferences. After that, we define a way to handle these constraints during identification process in the CCIC algorithm.

5.4.1. Proposed constraints

To take software architect preferences into account in the identification process, we propose four kinds of constraints: *Integratedclasses*(*x*), *Minimumclass*(*x*), *PredefinedComponent*(*LC*) and *Non-ZeroComponents* constraints, where *x* is one of the three analysis class types, i.e., *Boundary*(*B*), *Control*(*C*) and *Entity*(*E*) classes, and *LC* denotes a list of classes. In fact, these constraints help software architects to apply their preferences.

5.4.1.1. Integratedclasses(x) constraint. It denotes a constraint in which all analysis classes with *x* type are located into one component, e.g., *Integratedclasses*(*E*) constraint denotes that all entity classes of a system must be located in one component. One of the reasons for considering *Integratedclasses*(*E*) constraint may be implementation framework constraints. For example, when *Hibernate java framework* [30] is used to manage persistent data for developing a software project, all entity classes are located in a *Data Access Objects* (DAO) component (or package). In another example, if *JSF java framework* [31] is used for presentation layer of a software project; then, it is necessary that all boundary classes are located in one page bean component. To handle this case, the CCIC algorithm uses *Integratedclasses*(*B*) constraint. Furthermore, if the given project is a web-based software and all boundary classes have to partition into one server node (i.e., one component), the software architect can use *Integratedclasses*(*B*) constraint to handle this deployment model constraint. Therefore, this constraint can help software architects to apply their preferences on the identification process with respect to the use of common implementation frameworks and handle the deployment model constraints.

5.4.1.2. Minimumclass(x) constraint. It denotes a constraint in which all the identified components must have at least one analysis class with *x* type, e.g., *Minimumclass*(*B*) constraint shows that all components must have at least one boundary class. One of the reasons for

considering *Minimumclass*(*B*) constraint may be software deployment constraints. For example, when a software system is distributed in enterprise environments, each software component must have at least one boundary class; therefore, the CCIC algorithm uses *Minimumclass*(*B*) constraint to handle this constraint. Furthermore, application of *Minimumclass*(*C*) constraint in the identification process guarantees that each component has at least one key class, i.e. active class. Therefore, this constraint can help apply some deployment model and active class limitations on the identification process

5.4.1.3. PredefinedComponent(LC) constraint. It denotes a constraint in which all analysis classes belonging to *LC* are marked as a legacy component; then, the CCIC algorithm cannot alter this component. In fact, this constraint helps software architects to reuse some components produced in legacy systems or different projects. Additionally, when software architects have to use existing components due to technology or deployment model limitations, the use of this constraint seems likely to be useful.

5.4.1.4. Non-ZeroComponents constraint. It denotes a constraint in which the number of classes in each component has to be equal or higher than one. This constraint is used to avoid identification of empty components.

It should be noted that the goal of these corrections is to limit search spaces into valid software component spaces. For example, when we employ either *Minimumclass*(*C*) or *Minimumclass*(*B*) constraints, we mean that each component must have at least one active class. This preference cannot be provided by a general clustering method, because the goal of these methods only is achieving components with high cohesion and loose coupling. Therefore, giving these preferences to CCIC leads to search in a valid component space based on software architect's demands. After determining project constraints by software architects, CCIC guarantees to achieve a high quality solution with respect to minimizing coupling and maximizing cohesion to handle their constraints even for a large project. Note that in the software architecture viewpoint, an architectural style has certain constraints for certain components, then, applying these constraints on certain components of a system can lead to achieve a software architecture that conforms an architectural style. CCIC with these proposed constraints tries to identify software components that conform an architectural style such as layers, Client/Servers, and Pipe/Filter. For example, for the Pipe/Filter software architecture style, a constraint can be defined, in which control classes participated in Pipe/Filter style are given to CCIC, so it tries to identify some components, in which no two control classes are assigned into one component. Therefore, in obtained components, each Pipe/Filter's control class is assigned to a separate component.

5.4.2. Constraints handling with chromosome correction

In CCIC, after generating a chromosome, invalid component solutions may be produced, i.e., the solution presented in the chromosome does not satisfy some of the given constraints. To handle constraints in evolutionary algorithms, there are four common methods [32]: (1) omitting infeasible chromosomes and regeneration, (2) punishing infeasible chromosomes by fitness penalty, (3) defining operators to avoid generating infeasible chromosomes, and (4) converting infeasible chromosomes to feasible ones. The first method is extremely time consuming, and determining a good fitness penalty function for the second method is so difficult [33]. In addition, defining operators without generating infeasible chromosomes often directs the search algorithm to be limited to explore the entire search space. Therefore, in CCIC, the fourth method is preferred to use. In fact, when an infeasible chromosome is generated according to a violated constraint, the corresponding procedure defined below is applied on it.

Procedure Handling *Integratedclasses*(*x*) constraint	**Procedure** Handling *PredefinedComponent*(*LC*) constraint
1. *Component formation* step (defined in Section 5.1.2) is performed and all members of each component are determined.	
2. All analysis classes with type *x* are removed from all components and located into a new component.	2. All analysis classes belonging to *LC* are removed from all components and located into a new component.
3. For all components, the following conditions are separately evaluated:	
3.1. If the component has no key classes, a class with the highest similarity with other classes belonging to that component is chosen as a key class, and the corresponding parametric gene value is updated.	
3.2. If the component has no member, it is discarded, i.e., the corresponding 2nd level control gene value is set to zero.	

Procedure Handling *Minimumclass*(*x*) constraint

1. *Component formation* step (defined in Section 5.1.2) is performed, and all members of each component are determined.
2. If the number of chromosome components is higher than the number of analysis classes with type *x*, *components merging procedure* is repeated until the number of components of the chromosome is equal to the number of analysis classes with type *x*.
3. All the components are evaluated, and Step 3.1 is executed for each component Cmp_i that violates the constraint.

3.1. Similarities between Cmp_i and all analysis classes with type *x* are calculated according to Eq. (14); then, the most similar analysis class with type *x* belonging to a component that has more than one analysis class with type *x* is selected and added to Cmp_i.

$$Sim(C_i, Cmp_j) = \frac{\sum_{\forall C_j \in Cmp_j} Sim(C_i, C_j)}{TNC_j} \quad (14)$$

where TNC_j is the total number of analysis classes belonging to the component Cmp_j.

Procedure *components merging*

1. Similarities between all pairs of analysis classes are calculated.
2. For each component Cmp_i, similarities to other components are calculated according to Eq. (15).

$$Sim(Cmp_i, Cmp_j) = \frac{\sum_{\forall C_i \in Cmp_i} \sum_{\forall C_j \in Cmp_j} Sim(C_i, C_j)}{TNC_i \times TNC_j} \quad (15)$$

where TNC_i and TNC_j are the total number of analysis classes belonging to Cmp_i and Cmp_j, respectively.

3. Two components with the highest similarity are merged into one new component, and a class having the highest similarity with other classes belonging to that component is chosen as a key class, and the corresponding parametric gene values and 2nd level control gene values are updated.

It should be noted that if the *Non-ZeroComponents* constraint occurs for a component, that component is discarded and the corresponding 2nd level control gene value is updated. In fact, the goal of these

correction methods is to achieve some components that satisfy the system or architect constraints. When we do not employ these procedures in CCIC, the obtained results may have many violations of the constraints due to the randomized nature of the CCIC algorithm. Consequently, it is plausible that the components of CCIC without chromosome correction are different from expert's components. As expected, the results of applying CCIC on case studies shown in Table 2 reveal that the components of CCIC without chromosome correction are so different from expert's components. However, the software architects according to their preferences can lead to a situation, in which all chromosomes of genetic algorithm are equal. This situation appears when the software architects give all *Integratedclasses*(*B*), *Integratedclasses*(*C*), and *Integratedclasses*(*E*) constraints together to CCIC. Therefore, in this situation, applying the chromosomes correction of CCIC leads to generate the same chromosomes.

5.5. Selection and reproduction

Reproduction in CCIC means applying both crossover and mutation operators. In CCIC, two chromosomes are selected as parents for crossover using the roulette-wheel selection scheme [34], so that each parent's chance of selection is directly proportional to its fitness.

After selecting chromosomes for reproduction, some pairs of chromosomes are randomly selected to produce offspring chromosomes. The CCIC algorithm uses three standard crossover operators [34]: one-point, two-point, and uniform crossover. It should be noted that reproduction operators are separately applied on parametric genes and the 2nd level control genes. When the crossover operator is to be applied on two chromosomes, it is first applied on parametric genes and next, on the 2nd level control genes. Indeed, for each crossover, one of the three mentioned crossovers operators is randomly chosen and applied on both parts of chromosome pairs. After crossover, the produced children are mutated to avoid being trapped at local optima on the one hand and to ensure diversity on the other hand. The CCIC algorithm uses conventional *Random* mutation [25] as mutation operator. Like crossover, the mutation operator is separately applied on just two parts of a chromosome, i.e., parametric genes and the 2nd level control genes. Indeed, the mutation operator randomly changes the value of a 2nd level control gene from "0" to "1" or vice versa, and when it is applied on parametric genes, it randomly changes the value of a parametric gene into another analysis class number. It is worth mentioning that when crossover and mutation operators are applied on the 2nd level control genes, the number of components is changed dynamically as the evolutionary process progresses. Therefore, the number of components does not need to be specified by software architects in advance.

6. Experimental results

In this section, the CCIC algorithm is evaluated and the results are reported. At first, we introduce four real-world case studies as experimental data; then, some experiment configurations for CCIC are defined. After that, a metric is described to compare the components identified by CCIC with the ones identified by experts. Finally, the results obtained by CCIC are reported and compared to expert's opinions.

6.1. Experimental data

We used four real-world systems in this study, which were different in terms of the software architect and project preferences,

Table 1
Details of four real-world systems as our case studies.

System	Parameter														
	The number of use cases	The number of analysis classes				The number of connections among classes	Project constraints								Application domain
		Boundary	Control	Entity	Total		*PredefinedComponent(LC)*	*Integratedclasses(B)*	*Integratedclasses(C)*	*Integratedclasses(E)*	*Minimumclass(B)*	*Minimumclass(C)*	*Minimumclass(E)*		
Home Appliance Control System (HACS)	13	11	12	8	31	193		✓				✓		✓	Centralized (embedded)
Broker Online System (OBS)	30	4	10	6	20	227		✓				✓		✓	Centralized (web-based)
Custom System	28	17	11	26	54	1054	✓	✓				✓		✓	Distributed (web-based)
AgriInsurance System	68	43	36	154	233	6471			✓	✓		✓			Centralized (web-based)

Table 2
Results of CCIC with two modes, SCI-GA, and two classical clustering algorithms for the four cases.

Method	Case study											
	HACS			*OBS*			*Custom System*			*AgriInsurance System*		
	Measurement											
	Avg. DBI	Best *F*-measure	Avg. time (s)	Avg. DBI	Best *F*-measure	Avg. time (s)	Avg. DBI	Best *F*-measure	Avg. time (s)	Avg. DBI	Best *F*-measure	Avg. time (s)
RBR	0.0188	0.51	2	0.0164	0.61	1.6	0.0103	0.55	5.8	0.0118	0.58	23
Agglomerative hierarchical	0.0282	0.61	1.8	0.0163	0.64	1.3	0.0119	0.5	**4.2**	0.013	0.53	**14**
SCI-GA	0.0103	0.56	98.2	**0.0083**	0.63	112.9	0.0073	0.61	336.5	**0.0081**	0.51	967.8
CCIC without constraints handling	**0.0094**	0.58	138.6	**0.0083**	0.63	179.3	**0.0061**	0.58	592.3	**0.0081**	0.51	1631.2
CCIC with constraints handling	0.0297	**0.94**	1891.2	0.0221	1	3052.8	0.0134	**0.93**	12989	0.0182	**0.9**	14267

the number of analysis classes, and application domains. Table 1 provides details of the used systems.

As shown in Table 1, all the cases (except for *AgriInsurance System*) have both *Minimumclass*(*C*) and *Minimumclass*(*E*) constraints, since in all of them, it is supposed that each logical component needs at least one control class as a key class, and has own data, i.e., each logical component separately maintains its data like a distributed application because of deployment model constraints. In addition, all of these cases have the *Non-ZeroComponents* constraint. To illustrate these case studies, we introduce them as follows:

6.1.1. HACS [35]

This system is a centralized, remotely accessed, intelligent system to act as an intermediary between the user and his home appliances. In this system, all boundary classes must be compacted into one component, because it has one remote control to manage appliances. Therefore, the CCIC algorithm employs the *Integratedclasses*(*B*) constraint to handle this project preference.

6.1.2. OBS [36]

This system automates the traditional stock trading using internet and gives faster access to stock reports, current market trends and real-time stock prices. As it is a web-based application with different databases, this system needs that all boundary classes are compacted into one component. Therefore, like HACS, the CCIC algorithm employs the *Integratedclasses*(*B*) constraint to handle this project preference.

6.1.3. Custom System [37]

The United Nations Economic Commission for Europe (*UNECE*) administers the eTIR Convention, which provides an internationally recognized procedure to facilitate the cross border transportation of goods in transit using a standard. In this system, there are two international sub-systems: *International Guarantees* and *eTIR International Assurance*. These two sub-systems bear two distinctive logical components in the deployment viewpoint. For this reason, this system must have at least three components, including two predefined components and a component deployed in a main server. Additionally, due to the web-based nature of the system, all boundary classes except for those belonging to the predefined components must be compacted into one component. Therefore, to handle these project preferences, the CCIC algorithm employs *Integratedclasses*(*B*) and *PredefinedComponent*(*LC*) constraints, where *LC* represents two sets of analysis classes that must be marked as legacy components.

6.1.4. AgriInsurance System [38]

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, a famous software house company in Iran, for *Agriculture Bank*, one of the Iranian banks customized to the agriculture finance for Iranian farmers. Note that this project is developed by more than 10 professional developers with the average of 4 years of experience. In this system, Hibernate framework [30] is used to manage data, so the CCIC algorithm employs *Integratedclasses*(*E*) constraint to integrate all entity classes into one component. Additionally, in contrast to other introduced cases, according to project preferences, in this system each component must have at least one boundary class and at least one control class; therefore, the CCIC algorithm

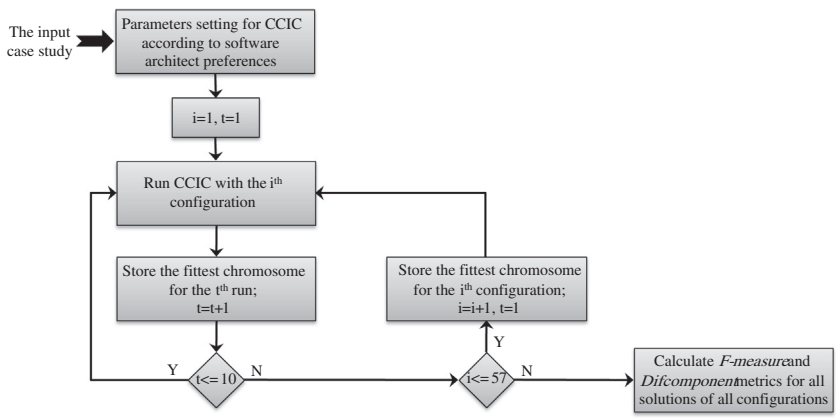
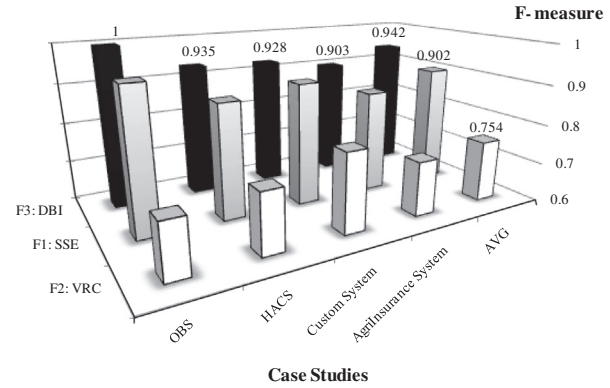
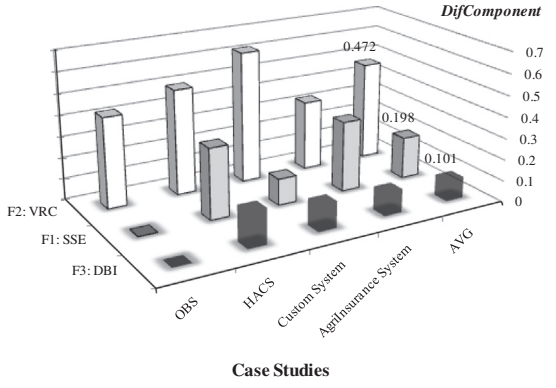


Fig. 11. The process of experiments for each case study.



(a) The best obtained *F-measure* of CCIC with *F1*-based, *F2*-based and *F3*-based configurations



(b) The best obtained *DifComponent* of CCIC with *F1*-based, *F2*-based and *F3*-based configurations

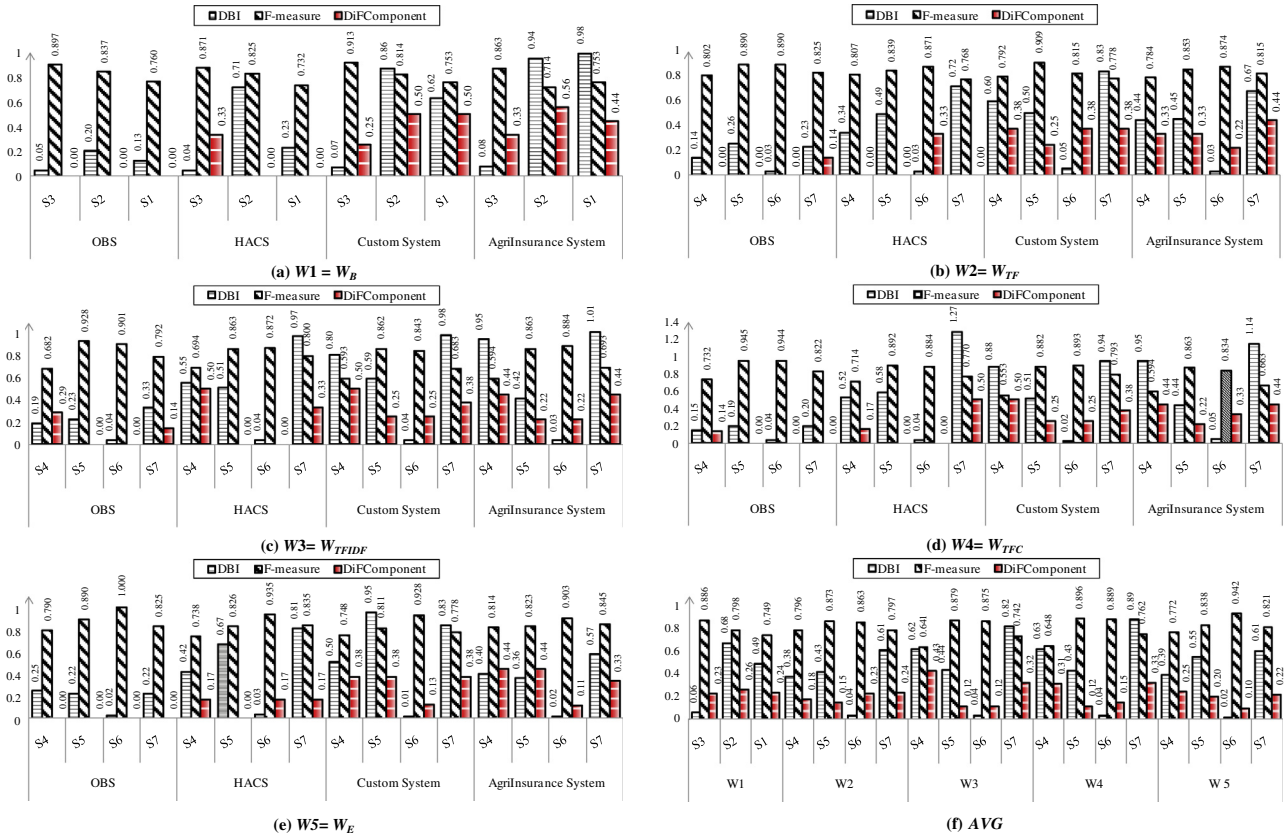
Fig. 12. The best obtained results of CCIC for all case studies.

employs the *Minimumclass*(*B*) and *Minimumclass*(*C*) constraints to handle these preferences.

Note that for OBS, HACS and *Custom System* cases, five professional developers with an average of 6 years of experience are considered as experts. For each case study, after identifying components by each expert, one of the five experts with higher experience than others combines all the expert's solutions to reach one solution. The components identified by experts for OBS, HACS and *Custom System* cases are reported in Appendix B. It is worth mentioning that the goal of all experts is to increase cohesion and decrease coupling.

6.2. Experiment configurations

To conduct a comparative study, we considered 57 configurations, which are composed of two typical similarity measures, i.e., similarity measures for binary and real vectors presented in Section 4.2 and five weighting methods, i.e., W_B, W_{TF}, W_{TFIDF}, W_{TFC} and W_E presented in Section 4.1 as well as three fitness functions, i.e., *SSE*, *VRC*, and *DBI* presented in Section 5.3. Fig. 8 provides details of these configurations. As shown in Fig. 8, the binary similarity measures can only be employed with W_B weighting method due to their characteristics defined in Section 4.2.1. In



(a) $W1 = W_B$ (b) $W2 = W_{TF}$ (c) $W3 = W_{TFIDF}$ (d) $W4 = W_{TFC}$ (e) $W5 = W_E$ (f) *AVG*

Fig. 13. Results of CCIC for various *F3*-based configurations (Fig. 8).

the following sections, we applied a series of experiments with these 57 configurations on four cases to evaluate the various configurations for identifying components with good quality.

6.3. Experiment metrics

It should be noted that to evaluate the components identified by the CCIC algorithm in comparison with expert opinions, we use the metrics defined below. At first, the components identified by CCIC must be mapped to the ones identified by experts. To perform this task, we use Hungarian algorithm [39], which solves these assignment problems. Then, for each component identified by experts, *True Positives* (*TP*), *False Positives* (*FP*), *True Negatives* (*TN*), and *False Negatives* (*FN*) values depicted in Fig. 9 are computed. After that, *Precision*, *Recall*, and *F-measure* [20] are computed according to Eqs. (16)–(18), where *EC* is the number of components identified by experts.

$$\text{Precision} = \sum_{i=1}^{|EC|} \frac{TP_i}{TP_i + FP_i} \tag{16}$$

$$\text{Recall} = \sum_{i=1}^{|EC|} \frac{TP_i}{TP_i + FN_i} \tag{17}$$

$$F_{measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{18}$$

Another metric we used to compare between the CCIC results and expert opinions was *DifComponent* metric defined in Eq. (19). In fact, *DifComponent* metric computes a relative difference between the number of components identified by CCIC (i.e., TNC_{CCIC}) and the number of components identified by experts (i.e., TNC_E). To clarify these metrics, Fig. 10 presents an example of their computation.

$$DifComponent = \frac{|TNC_{CCIC} - TNC_E|}{TNC_E} \tag{19}$$

As shown in Fig. 10, CCIC identifies four components in comparison with three components identified by experts; then, after mapping the similar components, the metrics defined in Eqs. (16)–(18) were calculated. It is worth mentioning that to evaluate components identified by CCIC, in addition to the use of *DifComponent* metric, we used *F-measure*, as this metric is considered as a suitable combination of *Precision* and *Recall* measurements [20].

6.4. Experimental results for various configurations

We implemented the CCIC algorithm and ran it with parameter settings determined by performing empirical studies [40] as follows: population size was 100, crossover and mutation rates were 70%, and 2%, respectively. Additionally, the CCIC algorithm was stopped when the generation number exceeded 1000 or the fitness value did not improve during the last 50 generations. To measure the performance of CCIC, we applied it on the four real-world cases introduced in Section 6.1. Note that CCIC with all 57 configurations defined in Section 6.2 was separately applied on the four cases. Moreover, in these experiments, for each configuration: (1) the CCIC algorithm runs 10 times and the best obtained result corresponding to the fittest chromosome is stored, (2) it is compared to the expert's solution, and (3) *F-measure* and *DifComponent* metrics are calculated. This process is shown in Fig. 11.

Fig. 12 shows the experimental results of the best obtained results of *F1*-based, *F2*-based and *F3*-based configurations, which are evaluated by the *F-measure* and *DifComponent* metrics for four different case studies. As shown in Fig. 12(a), the *F3*-based configurations achieved the highest values of *F-measure* for all four cases. Therefore, according to the averaged results of the four cases shown in Fig. 12(a), the solutions identified by *F3*-based configurations were the most similar to the ones identified by experts. The results shown in Fig. 12(b) also depicted that *F3*-based configurations achieve the lowest value of *DifComponent*; therefore, it is concluded that the number of components identified by *F3*-based configurations is the most similar to the ones identified by experts.

In our previous work on improving clustering techniques [41], it was concluded that when the number of clusters is unknown, the *SSE* measure cannot be used, as its value is reduced by increasing the number of clusters, and although the computational complexity of *DBI* is higher than *SSE* and *VRC*, it is an appropriate criterion, when the number of clusters is unknown. As expected, *DBI* fitness function is superior to *SSE* and *VRC* in software identification domain, and Fig. 12 addresses this issue.

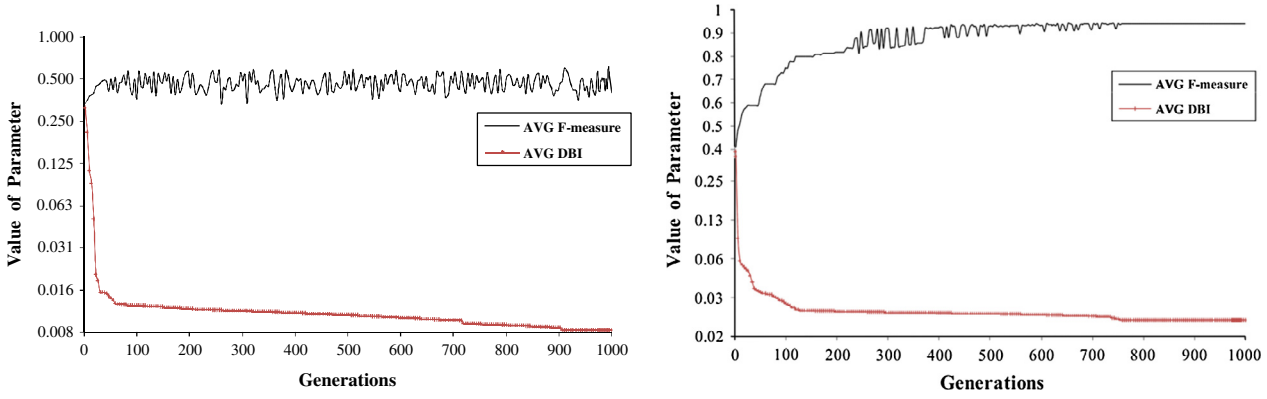
Therefore, we only report the results of *F3*-based configurations of CCIC. Fig. 13 shows the experimental results of the best obtained results of *F3*-based configurations, which are evaluated by *F-measure* and *DifComponent* metrics. As shown in Fig. 13(a), for the $W1 = W_B$ weighting method, the *S3 = Jaccard-NM* similarity measure generally identifies better solutions for all four cases than *S1* and *S2* similarity measures. As shown in Fig. 13(b), for the $W2 = W_{TF}$ weighting method, *S6* and *S5* similarity measures identify better solutions than the other similarity measures for real vectors. For OBS case, both *S6* and *S5* similarity measures achieve the same result, and for HACS case, although *S6* similarity measure achieves a better solution with respect to *F-measure*, the value of *DifComponent* metric for the obtained solution is worse than the one for the *S5* similarity measure. In addition, for *Custom* and *AgriInsurance Systems*, *S6* and *S5* similarity measures achieve better solutions, respectively. Comparing the results of *W1* and *W2* weighting methods shows that for moderate systems, like OBS, *W1* weighting method has a better performance, but for larger systems, like the *Custom* and *AgriInsurance Systems*, *W2* weighting method achieves a better performance.

As shown in Fig. 13(c), like *W2*, for *W3* weighting method, *S6* and *S5* similarity measures identify better solutions than the other similarity measures for real vectors. Comparing the results of *W1*, *W2* and *W3* weighting methods shows that although the results of *W3* weighting method for OBS and HACS cases are superior to those of *W2* weighting method, but *W2* weighting method has better performance for *Custom* and *AgriInsurance Systems*.

As shown in Fig. 13(d), like *W2* and *W3*, for *W4* weighting method, the *S6* and *S5* similarity measures identify better solutions than the other similarity measures for real vectors. Fig. 13(d) shows that *W4* weighting method with *S5* similarity measure outperforms *W1*, *W2* and *W3* weighting methods for both OBS and HACS cases, while its results for both *Custom* and *AgriInsurance Systems* are slightly worse than those for *W2* weighting method.

As shown in Fig. 13(e), for *W5* weighting method, *S6* similarity measure identifies better solutions than the other similarity measures for real vectors. In fact, Fig. 13(e) shows that *W5* weighting method with *S6* similarity measure produces the most similar components to the ones identified by experts with respect to *F-measure* and *DifComponent* metrics in comparison with the other weighting methods for all four cases.

To compare total results of *F3*-based configurations, we averaged the best obtained results of each weighting methods for all four cases, which is shown in Fig. 13(e). As shown in Fig. 13(e), for *W1*, *W2*, *W3*, *W4* and *W5* weighting methods, *S3*, *S5*, *S5*, *S5* and *S6* similarity measures achieve better performance, respectively. In Section 6.6, we will discuss the best weighting methods and similarity measures.



(a) CCIC without constraints handling

(b) CCIC with constraints handling

Fig. 14. The averaged effectiveness of CCIC for all four case studies.

6.5. Experimental results for constraint handling

As discussed earlier, one of the main contributions of this study is to handle software architect or project preferences. For this reason, in this section, we performed some experiments to show this capability of CCIC. We ran CCIC in two modes: (1) with chromosome correction and (2) without chromosome correction, i.e., without constraint handling, and the results were compared in Table 2. In both modes, CCIC is run 10 times, and the best obtained results for each mode are reported. According to the results of experiments performed in Section 6.4, *W5* weighting method, i.e., W_E, with *S6* similarity measure, i.e., $Sim_{E_u\text{-}NM}$, achieves the best performance; therefore, in this experiment, CCIC employs these weighting method and similarity measure for both modes.

To compare CCIC with the other clustering-based component identification methods, we used [6] and SCI-GA [15] as clustering-based component identification methods. The experimental results presented in [6] revealed that *RBR* (*Repeated Bisection k-means*) and *Agglomerative Hierarchical* clustering algorithms outperform other classical clustering algorithms including *k-means*, *Graph-based method*, and *Fuzzy C-means* algorithm. Therefore, we applied these clustering algorithms on the four cases introduces in Section 6.1. For applying these algorithms, we first calculated all the weighting methods defined in Section 4.1 for the four cases; then, all similarity measures introduced in Section 4.2, were applied on the weighting matrices. Finally, the obtained feature vectors are given to *RBR*, *Agglomerative Hierarchical*, and SCI-GA algorithms as inputs.

It is worth mentioning that SCI-GA generally employs use cases as inputs, but to compare it with CCIC in this experiment, SCI-GA employs analysis classes as inputs, and uses the key class concept instead of centroid use cases defined in [15]. Furthermore, to perform this experiment, SCI-GA uses *DBI* metric as its fitness function.

Table 2 shows the results of *RBR* and *Agglomerative Hierarchical* clustering algorithms, SCI-GA, and CCIC with two modes according to *DBI* and *F-measure* metrics and time. The key point in Table 2 is the fact that for both clustering algorithms, we determined the number of components in advance according to expert opinions, as opposed to CCIC and SCI-GA, which automatically identify the number of components. It is worth remembering that the results of *RBR* and *Agglomerative Hierarchical* clustering algorithms for different runs are the same due to their deterministic nature of these algorithms as opposed to the randomized nature of CCIC and SCI-GA. Note that the components identified by CCIC with constraint handling and RBR for OBS, HACS, and *Custom System* cases are reported in Appendix B.

As shown in Table 2, CCIC without constraint handling and SCI-GA are able to identify solutions far better than the ones identified by both *RBR* and *Agglomerative Hierarchical* clustering algorithms with respect to *DBI* metric, because they use powerful meta-heuristic search algorithms instead of simple heuristics like *k-means*. As shown in Table 2, comparing CCIC without constraint handling with SCI-GA according to *DBI* metric shows that both achieve approximately the same components, because both use the genetic algorithm. However, as shown in Table 2, SCI-GA is faster than CCIC since it uses the standard genetic algorithm as opposed to a complicated genetic algorithm like *HEA*, used in CCIC. Furthermore, comparing the run-times of both *RBR* and *Agglomerative Hierarchical* clustering algorithms with SCI-GA and CCIC shows that both clustering algorithms are so faster than CCIC and SCI-GA. The reason for this is that these classical clustering algorithms use deterministic and simple heuristics as opposed to CCIC and SCI-GA.

Although SCI-GA and CCIC without constraint handling are able to achieve much better solutions than both classical clustering algorithms according to *DBI* metric, the obtained solutions are not similar to the ones identified by experts. In fact, optimizing only *DBI* metric for a case study cannot guarantee the identification of desirable components from the expert's viewpoints. The reason for this is the fact that *DBI* metric does not consider software architect preferences; therefore, the obtained results may have significant differences from components derived by experts.

Considering the components identified by RBR as a clustering-based method reported in Appendix B shows two points: (1) there are very different between components identified by a clustering-based method and the ones identified by experts and (2) because clustering-based methods are not able to consider system constraints, these methods violate the constraints of the given systems. For example, in Appendix B, these violations are reported for three cases studies.

As opposed to both classical clustering algorithms, SCI-GA and CCIC without chromosome correction, when CCIC employs chromosome correction to handle project constraints, it is able to identify solutions quite similar to the ones derived by experts, because during its evolution process, it considers software architect preferences in addition to simultaneous consideration of cohesion and coupling. It should be noted that the results obtained by CCIC with constraints handling for OBS, HACS, and *Custom System* cases are reported in Appendix B.

To clearly show the CCIC effectiveness, Fig. 14 demonstrates the averaged values of *DBI* metric for all four case studies and the averaged values of the corresponding *F-measure* of the obtained solutions. As shown in Fig. 14, there is no relation between the change of *DBI* metric and *F-measure* during the CCIC evolution

process. However, as shown in Fig. 14(a), when the values of *DBI* metric are strictly decreasing as the number of generations grows, the values of *F-measure* for the results obtained by CCIC without constraint handling show very serious changes. On the contrary, as shown in Fig. 14(b), the results obtained by CCIC with constraint handling have fewer changes according to *F-measure*. This case emphasizes that the goal of CCIC without constraint handling is to optimize the fitness function just like other clustering algorithms, i.e., to minimize *DBI* metric, and it does not take software architect preferences into account. It should be noted that when we investigated the solutions obtained by CCIC without constraint handling, we found that in this mode, entity classes are often considered as key classes; therefore, it increases the total cohesion of components. However, this case leads to infeasible solutions from expert's viewpoints, as the entity classes cannot play the active class role of a component.

6.6. Summary of evaluations

According to the experiments presented in Sections 6.4 and 6.5, outline of the results is as follows:

- Regarding the fitness functions, it is concluded that among the three fitness functions used including *SSE, VRC* and *DBI* metrics, *DBI* metric outperforms the other metrics, and its application in CCIC leads to identification of the most similar components to the ones identified by experts.
- Regarding the weighting schemes, it is concluded that *W5* weighting method, i.e., W_E, produces the most similar components to ones identified by experts with respect to *F-measure* and *DifComponent* metrics in comparison with the other weighting methods for all four cases. The reason for this is that the proposed entropy-weighting based W_E is able to present closer to reality connection strength between a pair of classes, and this issue is addressed in Fig. 5(e) as a simple example.
- Regarding the similarity measures, it is concluded that $Sim_{E_u\text{-}NM}$ and Sim_{Cosine} similarity measures for real vectors and $Sim_{Jaccard\text{-}NM}$ similarity measure for binary vector outperform the other similarity measures for all four cases with respect to *F-measure* and *DifComponent* metrics. It seems likely that since both $Sim_{E_u\text{-}NM}$ and $Sim_{Jaccard\text{-}NM}$ similarity measures [21,22] are customized to software clustering, they are able to achieve the highest performance in these experiments. In addition, according to text mining results presented in [42], the cosine similarity is the best similarity to measure a pair of real vectors.
- Regarding the constraints handling, it is concluded that when CCIC employs the chromosome correction to handle project constraints (as opposed to classical clustering algorithms and its mode without chromosome correction), it is able to identify more cohesive and independent components with respect to software architect's preferences.

7. Related works and CCIC limitations

The attempts for automatic identification of logical or business components can be divided into four categories: *Graph Partitioning* [4,9,10], *Clustering-based* [3,6,11–13], *CRUD-based* [14,43], and *FCA-based* [5,7] methods, which are discussed in detail below. Additionally, we proposed a novel category, called *Evolutionary-based* methods.

7.1. Graph partitioning methods

Albani et al. [10] 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 preferences, the weights are assigned to edges. Finally, the graph is partitioned into components using a heuristic from graph theories. Peng et al. [9] transformed the relationship model among business elements to a weighted graph. Then, a graph segmentation method is applied on the graph to identify mutually disjoint sub-graphs as components. The authors claimed that the proposed method has achieved cohesive components with low coupling, but they did not evaluate their claim. In [4], the AHP (*Analytic Hierarchy Process*) is used to extract designer's preferences to compute required weights for each edge type of the graph. In fact, the authors proposed a systematic way to determine weights of CRUD relationships according to designer's preferences. It should be noted that the similarity between the work presented in [4] and CCIC is consideration of software architect or project preferences. However, the work presented in [4] aims at identifying business components, and its inputs are CRUD matrixes extracted from business models, in contrast to the CCIC algorithm that aims at identifying logical components, and its inputs are analysis class diagrams and collaboration diagrams. In addition, CCIC allows software architects to determine the main factors highly effective upon software architecture like deployment constraints, in contrast to the work presented in [4] that only allows them to determine the effects of CRUD relationships. However, like other graph partitioning methods, the work presented in [4] has two limitations: (1) weights are manually assigned to edges according to expert experiences and (2) it requires determining the number of components in advance.

7.2. Clustering-based methods

Lee et al. [12] proposed a method for clustering the 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 the components with the highest level of dependency with them. Identifying key classes is a critical problem, and is manually determined by experts. Jain et al. [13] 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 manually determined by experts. Kim and Chang [11] employed use case models, object models and collaboration diagrams to identify the components. For clustering related functions, functional dependencies of use cases are calculated, and related use cases are clustered. This work requires weighting, and does not give any guidelines in this regard. Shahmohammadi et al. [6] 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*, *Graph-based method*, and *Fuzzy C-means* were examined to achieve good software architecture. In [15], we proposed a clustering-based method called SCI-GA, which is based on a genetic algorithm to identify software components from use case models of a given system. In [3], at first, use cases of a system are extracted from the MDL (*Model Description Language*) file formed by default in Rational Rose. Then, these use cases are clustered by *k-means* to identify logical components. These *Clustering-Based* methods use classical clustering techniques (except for [15]), so they may achieve poor components due to their simple heuristics according to results presented in Table 2, and a disadvantage of them is the need to determine the best number of components in advance, as opposed to CCIC that does not require the number of components in advance. According to results presented in Table 2, although SCI-GA [15] outperforms classical clustering techniques in terms of *DBI* metric, but it cannot identify components according

to software architect's preferences, and it cannot use legacy components.

7.3. CRUD-based methods

Lee et al. [43] presented a tool called COMO, in which "use case/class matrix" is created with respect to use case diagrams and class diagrams; then, it is partitioned into blocks with tight cohesion as business components. Ganesan and Sengupta [14] presented a tool similar to COMO called O2BC, but it has several differences in the clustering technique, and uses business events and domain objects as input. However, these two *CRUD-Based* methods have a number of limitations similar to *Clustering-Based methods*, due to the use of classical clustering techniques.

7.4. FCA-based methods

Hamza [7] initially proposed a framework based on the theory of 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 as important metrics to identify components. Cai et al. [5] proposed a novel method based on Fuzzy FCA. They transformed business elements and their memberships into a lattice; then, they used a simple clustering technique to identify the components. They used dispersion and distance concepts to measure the cohesion and coupling, respectively. An important advantage of [5] in contrast to the other methods is that like CCIC, it considers cohesion and coupling simultaneously throughout the identification process. Note that this property leads to a good trade-off between these metrics. However, they used two dispersion and distance thresholds (i.e., *TD* and *Ts* 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.

7.5. Evolutionary-based methods

Recently, evolutionary algorithms have been widely applied to software problems. Therefore, a new scope of software engineering is appeared called *Search-Based Software Engineering* (SBSE) [44,45] to reformulate software problems as optimization problems. In [46], all works related to SBSE are categorized, and *Search-Based Software Design* (SBSD) works are especially surveyed in [47]. SBSD [47] applies meta-heuristic search techniques, such as genetic algorithms, to software design problems. Indeed, CCIC is classified as a SBSD method that uses the customized evolutionary-based method, i.e., HEA, to identify logical components. Due to the use of the customized HEA and with respect to software architect or project preferences, the CCIC algorithm has several advantages over the other methods. First, it searches components space much better than classical clustering techniques (according to Table 2), because it uses a powerful optimization search algorithm (i.e., HEA) instead of simple heuristics like *k-means.* Second, it does not require the number of components in advance in contrast to the other methods, so it automatically identifies the suitable number of logical components (according to the results presented in Fig. 12). Finally, it provides facilities for software architects to identify logical components according to their preferences including the deployment constraints, implementation framework constraints, and legacy components that have to be reused.

In this study, we theoretically compared the features of CCIC with those of the other available methods (see Sections 7 and 5.4), and practically compared CCIC with the *Clustering-based methods* [6] and SCI-GA [15] (according to Table 2). However, we do not practically compare CCIC with the other methods since they do not take into account software architect preferences considered in CCIC; therefore, they are expected to identify poor logical components with significant differences from the components derived by experts similar to the *Clustering-based methods* [6] (as shown in Figs. A5–A7 in Appendix B). The reason for this is that none of the available methods considers our design constraints, and their goal is only to maximize component cohesions and minimize coupling between components.

In the course of experiments during the evaluation, a number of limitations of CCIC became apparent. First, the use of an evolutionary search algorithm leads to an increase in complexity, particularly time complexity. In addition, when CCIC employs chromosome correction, the run-time of the CCIC algorithm is greatly increased according to Table 2. To address this issue, we are going to propose an adaptive fitness penalty as a cheaper way to avoid infeasible solutions instead of the chromosome correction defined in Section 5.4.2. 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 CCIC algorithm is tolerable.

The second limitation of CCIC is that although it provides several facilities mentioned in Section 5.4.1 for software architects to determine their preferences as opposed to the other existing methods, like the other existing methods, it does not take into account many factors, which are effective in component identification, such as performance, reliability, and component complexity. However, the CCIC algorithm attempts to map the component identification problem to an optimization problem with the predefined constraints; therefore, it is possible to extend it to consider the other factors, as opposed to the other existing methods introducing completely automated procedures, which cannot be influenced by the software architects. Note that each quality concern is achieved by architectural styles or patterns. As mentioned in Section 5.4.1, CCIC can identify components conforming an architectural style, because it can employ the constraints during the search process. Therefore, it will be able to handle different quality concerns by defining further constraints.

Finally, as the logical component identification problem is a type of NP-complete problem [23], therefore, like the other existing methods, the CCIC algorithm cannot guarantee to achieve an optimal solution. However, instead of the use of simple heuristics like *k-means*, CCIC uses a powerful search-based method, i.e., HEA, as a crucial alternative to solve NP complete optimization problems [24]. As shown in Table 2, CCIC has the ability to search components space much better than the other heuristics methods like *RBR* according to *DBI* metric.

8. Conclusions

Automatic identification of logical components is a crucial task during software design phase. In this paper, we proposed a search-based method called CCIC, based on a customized HEA to identify logical components from an analysis model. The CCIC algorithm provides several options, which support its customization to situational software architect or project preferences like the deployment constraints, implementation framework constraints, and legacy components that have to be reused as opposed to the other existing methods.

To evaluate the different aspects of CCIC, we examined CCIC with various configurations using four real-world case studies. According to the obtained results presented in Sections 6.4 and 6.5, we conclude several important consequences. First, among the three used fitness functions including *SSE, VRC* and *DBI* metrics, *DBI* metric outperforms the other metrics and identifies the most similar components to the ones identified by experts. Second, CCIC

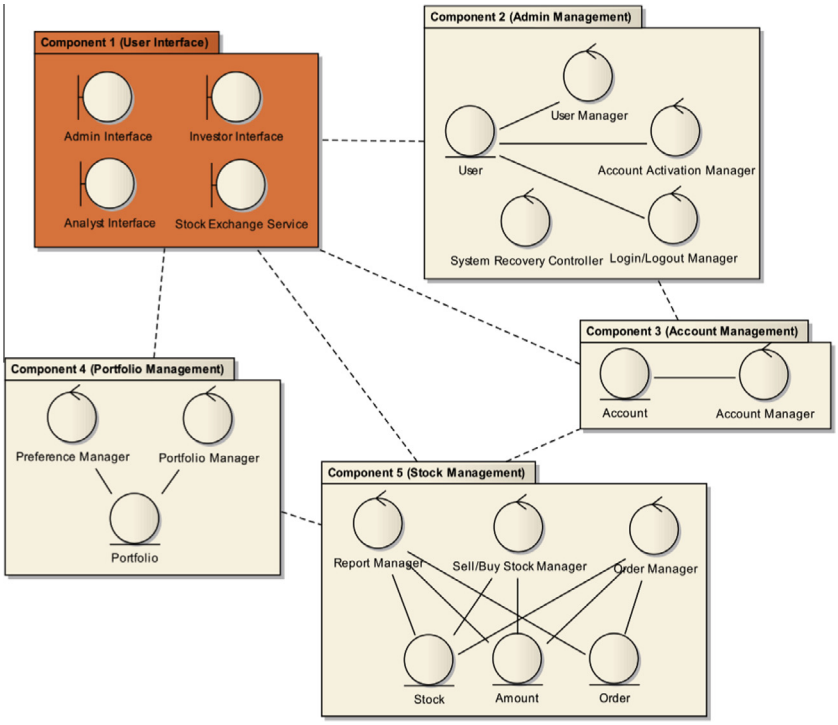


Fig. A1. Identified components of OBS case by experts and CCIC.

achieves its best results when it employs the proposed W_E weighting method with Sim_{E_u-NM} similarity measures. Third, when CCIC employs chromosome correction, it is able to identify more cohesive and independent components with respect to software architect or project preferences as opposed to the other existing methods.

We practically compared CCIC with the *Clustering-based methods* [6] and SCI-GA [15] (see Table 2), and the results revealed that the *Clustering-based methods* identify poor logical components with significant differences from components derived by experts, because like the other existing methods, they do not take software architect or project preferences into account as opposed to the CCIC algorithm that considers software architect preferences in addition to simultaneous consideration of cohesion and coupling.

For future work, we intend to use other optimization algorithms like PSO (*Particle Swarm Optimization*) [41] to examine the performance of the search. In addition, we are going to combine CCIC with the method presented in [42] to automatically select the right

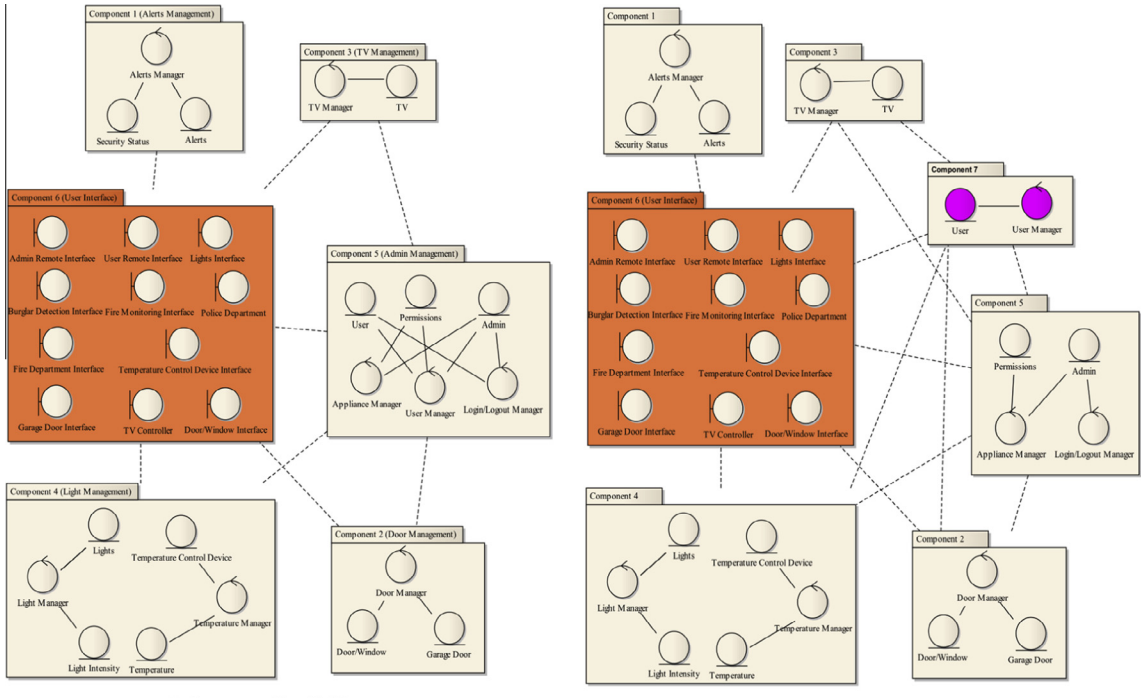


Fig. A2. Identified components of HACS case by experts and CCIC.

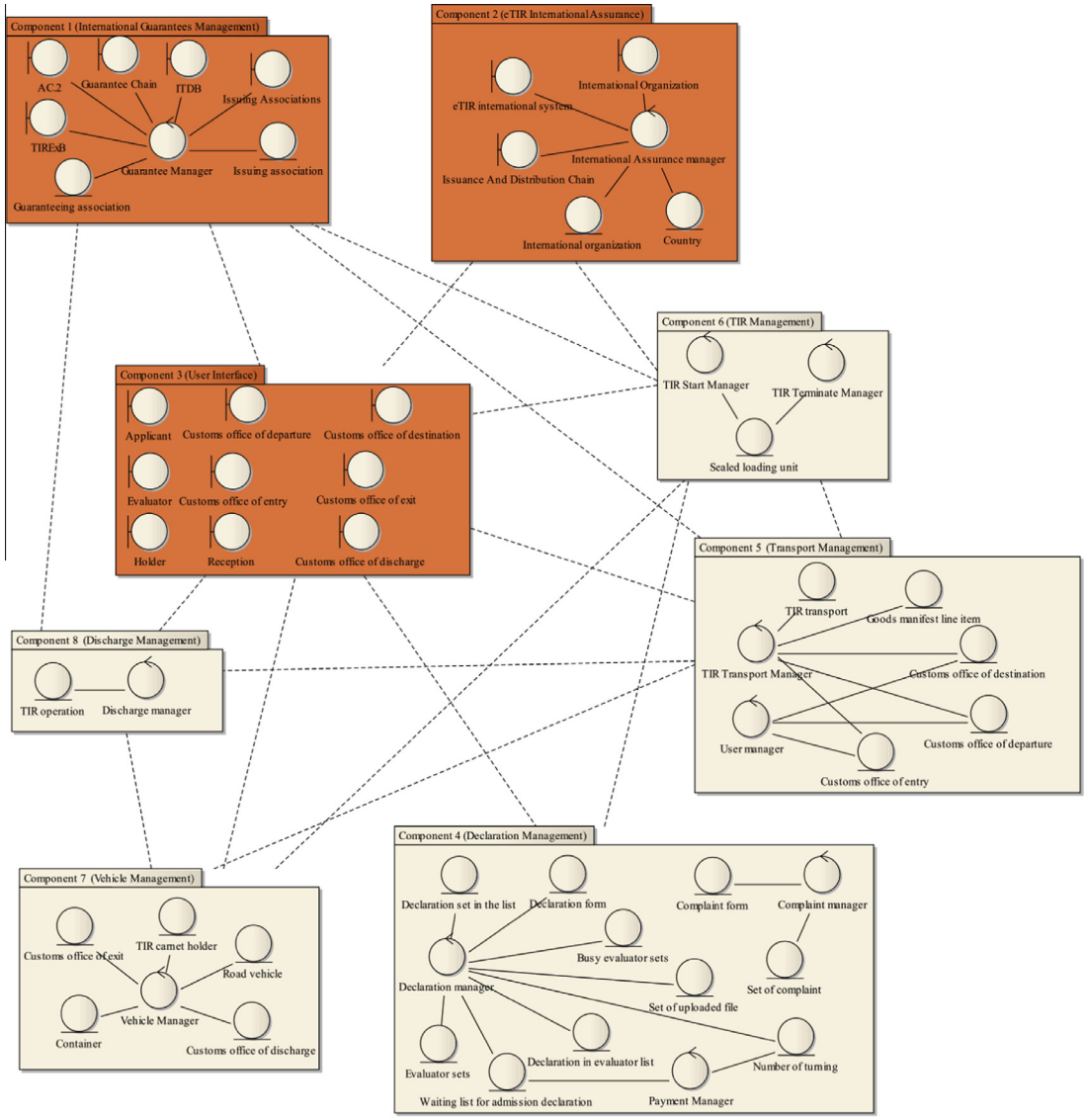


Fig. A3. Identified components of *Custom System* case by experts.

design pattern for a given design problem. Indeed, after automatically obtaining cohesive sub-components from CCIC according to the responsibility of each sub-component, a suitable design pattern or library routine will be automatically selected using [42,48]. Accordingly, an analysis model of a system can be accurately mapped to design pattern source codes.

Acknowledgment

This research was supported by ITRC, Iran Telecommunication Research Center.

Appendix A

The most common criterion in clustering is *SSE*, and the general objective is to obtain a cluster minimizing the squared error. This criterion is defined as follows:

$$SSE = \sum_{i=1}^{k} \sum_{O \in Cmp_i} (c_i - O)(c_i - O)^T \quad \text{(A1)}$$

where c_i, O and k are the key class of component Cmp_i, each class within component Cmp_i and the number of components, respectively. It is worth remembering that each class is represented by a feature vector defined in Section 4, and in *SSE*, the subtraction of a pair of classes denotes the subtraction between their corresponding feature vectors.

VRC is another criterion that can be used for cluster evaluation. This criterion considers both the within-component and between-component distances. The *VRC* is defined as follows:

$$VRC = \frac{Inter}{SSE} \times \frac{N-k}{k-1} \quad \text{(A2)}$$

where $(N-k)/(k-1)$ is the normalization term and prevents this ratio to monotonically increase with the number of components and N denotes the total number of analysis classes. *Inter* is the between-component distances defined as follows:

$$Inter = \sum_{i=1}^{k} |Cmp_i| \times (c_i - c)(c_i - c)^T \quad \text{(A3)}$$

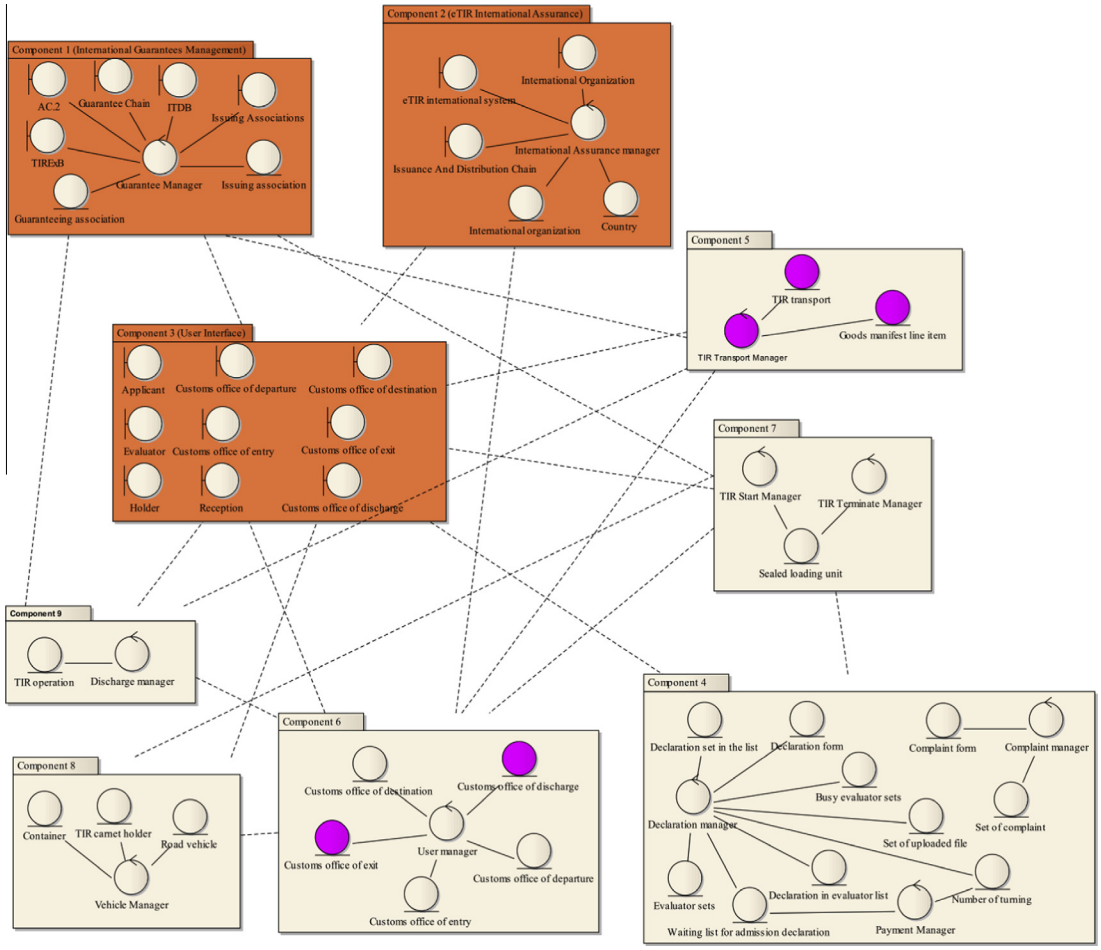


Fig. A4. Identified components of *Custom System* case by CCIC.

where c is the mean number of all classes obtained by the following equation.

$$c = \frac{1}{N}\sum_{i=1}^{N} O_i \tag{A4}$$

A popular criterion for the evaluation of clusters is *Davies–Bouldin Index* (*DBI*). The main objective of *DBI* criterion is to maximize between-component separation and minimize with O in-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:

$$S_{i,q} = \left(\frac{1}{|Cmp_i|}\sum_{O\in Cmp_i} \|O - c_i\|_2^q\right)^{\frac{1}{q}} \tag{A5}$$

where $S_{i,q}$ denotes the qth root of the qth moment of the analysis 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 and c_i. The distance between component Cmp_i and Cmp_j is defined as follows:

$$d_{ij,t} = c_i - c_{j_t} \tag{A6}$$

In Eqs. (A5) and (A7), q and t are integer numbers, where (q and $t \geqslant 1$) and they can be selected independently. In the following equation, $R_{i,qt}$ for component Cmp_i is defined as follows:

$$R_{i,qt} = \max_{j,j\neq i}\left\{\frac{S_{i,q} + S_{j,q}}{d_{ij,t}}\right\} \tag{A7}$$

Finally, the *DBI* is defined as follows:

$$DBI = \frac{1}{k}\sum_{i=1}^{K} R_{i,qt} \tag{A8}$$

A small value of the *DBI* and *SSE* criteria show better component solution in contrast to the *VRC* criterion.

Appendix B

Components identified by experts, CCIC with constraint handling and RBR for OBS, HACS, and *Custom System* cases are reported in this section. It should be noted that in Figs. A2(b), A4–A7, the dark use cases belonging to components identified by CCIC and RBR denote the ones whose components are different from components identified by experts.

Fig. A1 shows the components identified by experts and CCIC for OBS case including *User Interface*, *Admin Management*, *Portfolio Management*, *Account Management*, and *Stock Management* components. In this case, as mentioned in Table 1, all boundary classes are integrated in *User Interface component*. Furthermore, as shown in Table 2, CCIC with constraints handling, and W_E weighting method with $\boldsymbol{Sim}_{E_u-NM}$ similarity measure identifies the same as components derived by experts.

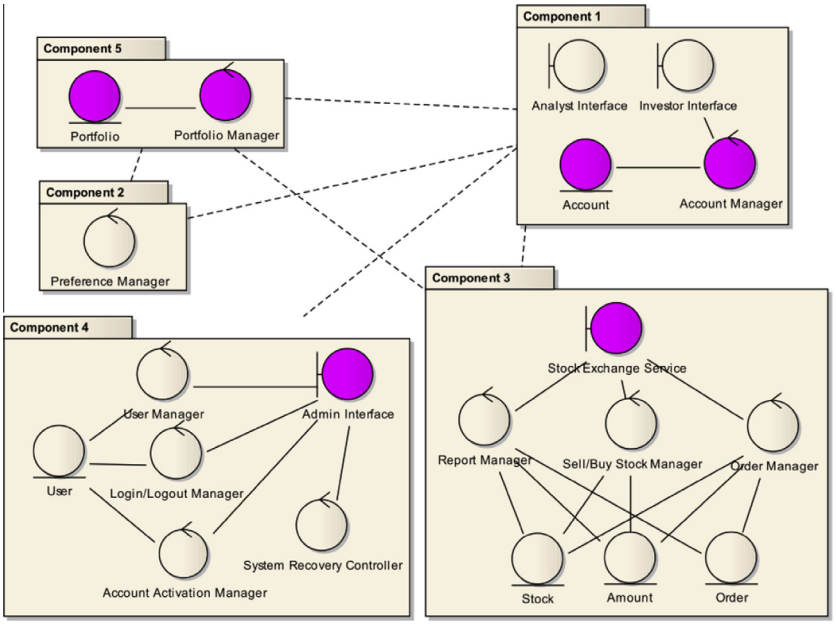


Fig. A5. Identified components of OBS case by RBR clustering algorithm.

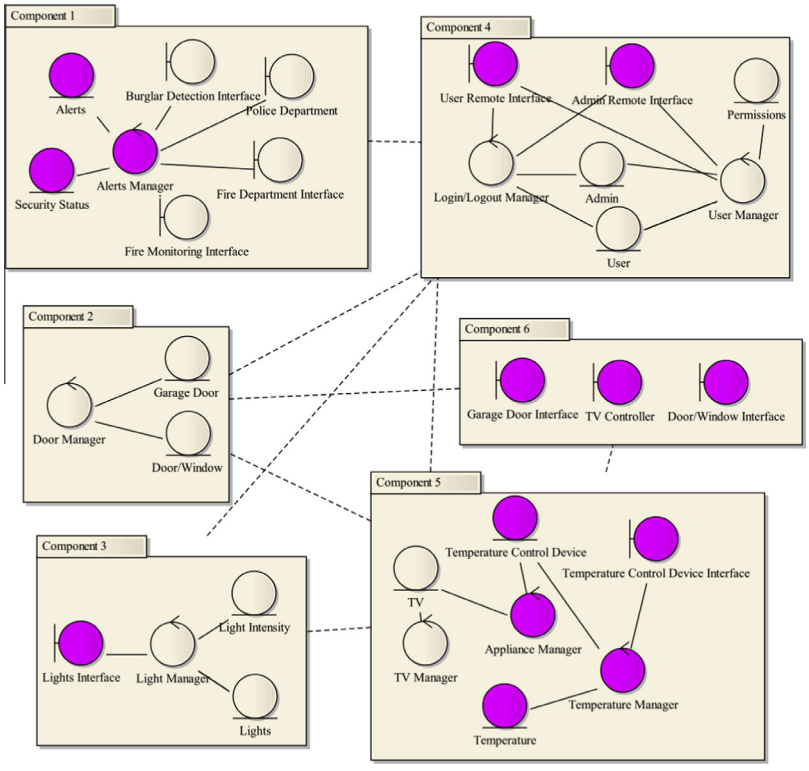


Fig. A6. Identified components of HACS case by RBR clustering algorithm.

Fig. A2(a) shows components identified by experts for HACS case including *User Interface*, *Alerts Management*, *TV Management*, *Admin Management*, *Light Management*, and *Door Management* components. In this case, as mentioned in Table 1, all boundary classes are integrated in *User Interface component*. Furthermore, Fig. A2 (b) shows the components identified by CCIC for HACS case with constraints handling, and W_E weighting method with Sim_{E_u-NM} similarity measure. Comparing the components derived by experts for HACS case (Fig. A2(b)) with the components identified by CCIC shows that there are two different analysis classes between expert's components and the components identified by CCIC.

Fig. A3 shows eight components identified by experts for Custom System case including User Interface, International Guarantees Management, eTIR International Assurance, Transport Management, TIR Management, Discharge Management, Declaration Management, and Vehicle Management components. In this case, as

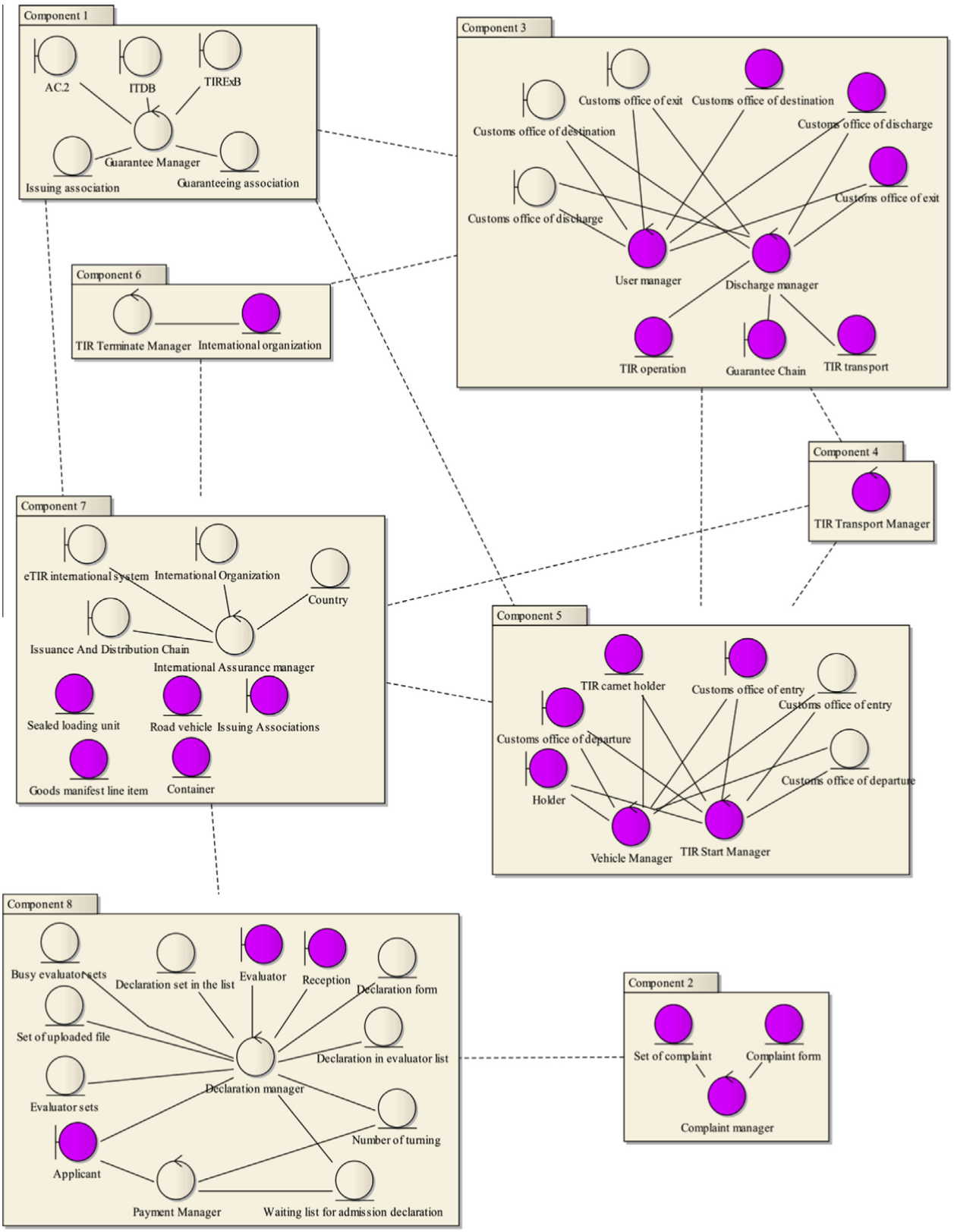


Fig. A7. Identified components of *Custom System* case by RBR clustering algorithm.

mentioned in Table 1, all boundary classes are integrated in User Interface component. Furthermore, as mentioned in Section 6.1, in this system, two international sub-systems: International Guarantees and eTIR International Assurance are considered as predefined components. Fig. A4 shows nine components identified by CCIC for Custom System case with constraints handling, and W_E weighting method with $\boldsymbol{Sim}_{E_u-NM}$ similarity measure. Comparing components derived by experts for Custom System case (Fig. A3)

with the components identified by CCIC (Fig. A4) shows that there are five different analysis classes between expert's components and the components identified by CCIC.

Figs. A5–A7 show components identified by RBR clustering algorithm for OBS, HACS, and *Custom System* cases, respectively. Comparing the components derived by experts for OBS, HACS, and *Custom System* cases (Figs. A1, A2(a), and A3) with the components identified by RBR (Figs. A5–A7) shows that there are 6, 14, and 27 different analysis classes between expert's components and the components identified by RBR, respectively. As shown in Fig. A5, the components identified by RBR for OBS case violate the *Integratedclasses*(B) constraint. Furthermore, Component 2 has only one control class which leads to violate the *Minimumclass*(E) constraint for OBS case. As shown in Fig. A6, like OBS case, the *Integratedclasses*(B) and *Minimumclass*(E) constraints are violated for HACS case. Note that Component 6 has only boundary classes, which leads to violation of the *Minimumclass*(E) constraint for HACS case. As shown in Fig. A7, for *Custom System* case like both OBS and HACS, the *Integratedclasses*(B) and *Minimumclass*(E) constraints are violated. Furthermore, for *Custom System* case, the components identified by RBR violate the *Predefinedcomponent* constraint.

References

[1] M.A. Khan, S. Mahmood, A graph based requirements clustering approach for component selection, Adv. Eng. Softw. 54 (2012) 1–16.
[2] J. Kim, S. Park, V. Sugumaran, DRAMA: a framework for domain requirements analysis and modeling architectures in software product lines, J. Syst. Softw. 81 (2008) 37–55.
[3] S. Kumar, R.K. Bhatia, R. Kumar, K-means clustering of use-cases using MDL, in: Proceeding of the Global Trends in Information Systems and Software Applications Conference, 2012, pp. 57–67.
[4] D.Q. Birkmeier, S. Overhage, A method to support a reflective derivation of business components from conceptual models, Inform. Syst. E-Bus. Manage. J. (2012) 1–33.
[5] Z.-g. Cai, X.-h. Yang, X.-y. Wang, A. Kavs, A fuzzy-based approach for business component identification, J. Zhejiang Univ.-Sci. C (Comput. Electron.) 12 (2011) 707–720.
[6] G.R. Shahmohammadi, S. Jalili, S.M.H. Hasheminejad, Identification of system software components using clustering approach, J. Object Technol. (JOT) 9 (2010) 77–98.
[7] H.S. Hamza, A framework for identifying reusable software components using formal concept analysis, in: Proceeding of the 6th International Conference on Information Technology: New Generations, 2009, pp. 813–818.
[8] D. Birkmeier, S. Overhage, On component identification approaches–classification, state of the art, and comparison, in: Proceeding of CBSE 2009, LNCS 5582, 2009, pp. 1–18.
[9] L. Peng, Z. Tong, Y. Zhang, Design of business component identification method with graph segmentation, in: Proceeding of the 3rd Int. Conf. on Intelligent System and Knowledge Engineering, 2008, pp. 296–301.
[10] A. Albani, S. Overhage, D. Birkmeier, Towards a systematic method for identifying business components, in: Proceeding of CBSE, LNCS 5282, 2008, pp. 262–277.
[11] S. Kim, S. Chang, A systematic method to identify software components, in: Proceeding of the 11th Software Engineering Conf., 2004, pp. 538–545.
[12] J.K. Lee, S.J. Jung, S.D. Kim, W.H. Jang, D.H. Ham, Component identification method with coupling and cohesion, in: Proceeding of the 8th Asia-Pacific Software Engineering Conference, 2001, pp. 79–86.
[13] H. Jain, N. Chalimeda, N. Ivaturi, B. Reddy, Business component identification, a formal approach, in: Proceeding of the 5th IEEE Int. Enterprise Distributed Object Computing Conf., 2001, pp. 183–187.
[14] R. Ganesan, S. Sengupta, O2BC: a technique for the design of component-based applications, in: Proceeding of the 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems, 2001, pp. 46–55.
[15] S.M.H. Hasheminejad, S. Jalili, SCI-GA: software component identification using genetic algorithm, J. Object Technol. (JOT) 12 (2013) 1–34.
[16] X. Rui, D.C. Wunsch, Clustering algorithms in biomedical research: a review, IEEE Rev. Biomed. Eng. 3 (2010) 120–154.
[17] K.S. Tang, K. Man, F.S. Kwong, Z.F. Liu, Design and optimization of IIR filter structure using hierarchical genetic algorithms, IEEE Trans. Ind. Electron. 45 (1998) 481–487.
[18] J.H. Holland, Adaption in Natural and Artificial Systems, MIT Press, Ann Arbor, 1975.
[19] P. Kruchten, The Rational Unified Process an Introduction, second ed., Addison Wesley, 2000.
[20] F. Sebastiani, Machine learning in automated text categorization, J. ACM Comput. Surv. (CSUR) 34 (2002) 1–47.
[21] R. Naseem, O. Maqbool, S. Muhammad, Improved similarity measures for software clustering, in: Proceeding of the 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011, pp. 45–54.
[22] O. Maqbool, H. Babri, Automated software clustering: an insight using cluster labels, J. Syst. Softw. 79 (2006) 1632–1648.
[23] J.F. Cui, H.S. Chae, Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems, Inform. Softw. Technol. 53 (2011) 601–614.
[24] G. Zäpfel, R. Braune, M. Bögl, Metaheuristic Search Concepts, Springer, Heidelberg Dordrecht London New York, 2010.
[25] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, New York, 1996.
[26] V.P. Venkatesan, M. Krishnamoorthy, A metrics suite for measuring software components, J. Convergence Inform. Technol. 4 (2009) 138–153.
[27] M. Choi, I.J. Kim, J. Hong, J. Kim, Component-based metrics applying the strength of dependency between classes, in: Proceeding of the ACM symposium on Applied Computing, 2009, pp. 530–536.
[28] T. Calinski, J. Harabasz, A dendrite method for cluster analysis, Commun. Stat. 3 (1974) 1–27.
[29] D.L. Davies, D.W. Bouldin, A cluster separation measure, IEEE Trans. Pattern Anal. Mach. Intell. 1 (1979) 224–227.
[30] Hibernate – JBoss Community. <www.hibernate.org/> (accessed April 2014).
[31] Java Server Faces. <www.seamframework.org> (accessed April 2014).
[32] S. Salcedo-Sanz, A survey of repair methods used as constraint handling techniques in evolutionary algorithms, Comput. Sci. Rev., Elsevier 3 (2009) 175–192.
[33] S. Parsa, O. Bushehrian, Genetic clustering with constraints, J. Res. Pract. Inform. Technol. 39 (2007) 47–60.
[34] K. Deb, Multi-Objective Optimization using Evolutionary Algorithms, John Wiley & Sons Ltd., Chichester, England, 2001.
[35] HACS Case. <http://www.coursehero.com/file/3666313/HACSFINAL/> (accessed April 2014).
[36] OBS Case. <http://www.isr.umd.edu/~austin/ense621.d/projects04.d/project_gouthami.html> (accessed April 2014).
[37] eTIR Project. <http://www.unece.org/trans/bcf/etir/welcome.html> (accessed April 2014).
[38] Agricultural Insurance Fund. <http://www.aiiri.gov.ir/HomePage.aspx?TabID=1&Site=aiiriPortal&Lang=en-US> (Accessed April 2014).
[39] G.A. Mills-Tettey, A. Stentz, M.B. Dias, The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs, Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-07-27, 2007.
[40] J.J. Grefenstette, Optimization of control parameters for genetic algorithms, IEEE Trans. Syst. Man. Cybern. 16 (1986) 122–128.
[41] H. Masoud, S. Jalili, S.M.H. Hasheminejad, Dynamic clustering using combinatorial particle swarm optimization, Appl. Intell. 38 (2013) 289–314.
[42] S.M.H. Hasheminejad, S. Jalili, Design patterns selection: an automatic two-phase method, J. Syst. Softw. 85 (2012) 408–424.
[43] S.D. Lee, Y.J. Yang, E.S. Cho, S.D. Kim, S.Y. Rhew, COMO: a UML-based component development methodology, in: Proceeding of the 6th Asia Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society, Los Alamitos, 1999, pp. 54–61.
[44] M. Harman, S.A. Mansouri, Y. Zhang, Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications, Department of Computer Science, King's College London, Tech. Rep. TR-09-03, 2009.
[45] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: trends, techniques and applications, ACM Comput. Surv. (CSUR) 45 (2012) 11.
[46] SBSE Publications. <www.sebase.org/sbse/publications> (accessed April 2014).
[47] O. Räihä, A survey on search-based software design, Comput. Sci. Rev. 4 (2010) 203–249.
[48] S.M.H. Hasheminejad, S. Jalili, Selecting proper security patterns using text classification, in: Proceeding of the International Conference on Computational Intelligence and Software Engineering, CiSE, 2009, pp. 1–5.