Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets © Jot, 2011
Online at http://www.jot.fm

SCI-GA: Software Component Identification using Genetic Algorithm

Seyed Mohammad Hossein Hasheminejad [a] Saeed Jalili [a]

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

Abstract Identifying software components is a crucial task in software development. There are a number of methods to identify components in the literature; however, the majority of these methods rely on clustering techniques with expert judgment. In contrast to the previous methods, which have used classical clustering techniques, this paper maps the components identification problem to an optimization problem. We propose a novel GA-based algorithm (*Genetic Algorithm*) as a powerful optimization search algorithm, called SCI-GA (*Software Component Identification using Genetic Algorithm*), to identify components from analysis models. SCI-GA uses software cohesion, coupling, and complexity measurements to define its fitness function. For performance evaluation, we evaluated SCI-GA using three real-world cases. The results reveal that SCI-GA can identify correct suboptimal software components, and performs far better than alternative heuristics like *k-means* and *FCA-Based* methods.

Keywords Software Component; Component Identification; Genetic Algorithm

1 Introduction

In Component-Based Software Development (CBSD) process, partitioning a software space to identify components is a crucial task. Several methods have been presented to identify software components, but they do not agree on what exactly is a component. Birkmeier and Overhage [BO09] divided components into three categories: *Business-oriented* components [AOB08, CYW11, GS01, JCI01, LYC99, PTZ08, WXZ05], *Architecture-focused* or *Logical* components [Ham09, KC04,

Seyed Mohammad Hossein Hasheminejad, Saeed Jalili. SCI-GA: Software Component Identification using Genetic Algorithm. In Journal of Object Technology, vol. 12, no. 2, 2013, pages 3:1–34. doi: 10.5381/jot.2013.12.2.a3

LJK01, SJH10], and *Technical* components [CC11]. *Business-oriented* components are associated with business components and require realizing business processes. *Architecture-focused* components concentrate on logical characteristics, i.e., a structuring is required in them. Finally, *Technical* components focus on deployment and implementation aspects.

This paper concentrates on a logical definition of component as follows [KPS08]: "*A logical component, in contrast to a physical component, is a component representing requirements except for technology, environments, and constraints. Nevertheless, it is meaningful that these logical components provide the starting point for designing the software architecture*".

Logical components are the primary abstractions of the entire design of a system. Partitioning a system into logical components has a key role in defining the system architecture [SJH10]. During the *CBSD* process, a software architect is responsible for decomposing a system into some logical components. However, because of heavy reliance on software architect experience, it is an extremely difficult and error prone task to identify logical components without any tool support [BO09]. To help overcome this difficulty, several works suggest automatic or semi-automatic methods to identify logical components.

Current attempts to identify logical components rely on classical clustering techniques with expert judgment and cluster either use cases or classes of a system into components. Lee et al. [LJK01] proposed a method for clustering classes into logical components with high cohesion and low coupling. Kim et al. [KC04] employed use case models, object models and collaboration diagrams to identify components. Hamza [Ham09] proposed a framework based on the theory of *Formal Concept Analysis* (FCA) to partition a class diagram into logical components with some heuristics similar to clustering techniques. Shahmohammdi et al. [SJH10] proposed a feature-based clustering method to identify logical components, in which several features like actors and entity classes were presented to measure the similarity between a pair of use cases. Therefore, several classical clustering techniques like *k-means*, *Hierarchal*, *Graph-based method*, and *Fuzzy C-means* were examined to achieve good software architecture.

These four methods use classical clustering techniques, and suffer from several common weaknesses. First, they need to manually adjust their thresholds, and highly depend on expert judgment to select the best solution. Second, the number of components must be manually determined by experts in advance. Third, they 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 [RW10]. Finally, the common objective of these methods is to optimize clustering criteria like *Sum of Squared Error* (*SSE*) and *Variance Ratio Criterion* (*VRC*) [HCF09] rather than software design measurements like software cohesion, coupling, and complexity.

Recently, evolutionary algorithms have been widely applied to software problems. Therefore, a new scope of software engineering by the name of *Search-Based Software Engineering* (SBSE) [HMZ12] has emerged to reformulate software problems as optimization problems. In [SP13], all works related to SBSE are categorized and *Search-Based Design* works are particularly surveyed in [Räi10].

The goal of this paper is to improve limitations of *Clustering-Based* methods. Therefore, we propose a search-based method called SCI-GA, which is based on an evolutionary approach (a GA-Based method), with the aim of mapping the logical component identification problem to an optimization problem. Compared with other *Clustering-Based* methods, especially [SJH10], SCI-GA has a number of distinguishing characteristics:

1. Component identification is an NP-complete problem [CYW11]. Therefore, SCI-GA uses a Meta-heuristic method (i.e., GA) as a powerful optimization search algorithm to identify components instead of a heuristic like *k-means*.
2. There is no requirement for the number of components to be determined in advance, and it can automatically identify the suitable number of components.
3. It uses a fitness function that measures software cohesion, coupling, and complexity metrics, so it automatically identifies suitable components.

For justification, we evaluated SCI-GA using three real-world cases, and the obtained results are analyzed and discussed in comparison with other methods.

The rest of this paper is organized as follows: Section 2 defines component identification problem, and Section 3 describes software design measurements used in SCI-GA. In Section 4, SCI-GA is described in detail, and in Section 5, we evaluate SCI-GA using three real-world cases. Finally, after describing related works in Section 6, Section 7 provides concluding remarks and future works.

2 Component Identification Problem

The goal of logical component identification is to partition requirements of a system into meaningful units. In RUP methodology [Kru00], requirements of a system are identified in the *Requirements Capture Workflow,* and are presented by use case model. Use case model consists of some use cases and actors. After capturing use cases, in RUP methodology, the identified use cases are described with more details at the *Analysis and Design Workflow*. One of the important artifacts in *Analysis and Design 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 this paper, SCI-GA inputs are a use case model and analysis class diagrams of a system. The goal of this paper is to partition use cases of a system

into cohesive and independent units called logical components. Figure 1(a) shows a representative use case model of a system. As shown in Figure 1(a), this system has two actors and seven use cases. In addition, Figure 1(b) shows the corresponding logical components which are identified for this system as an example. As shown in Figure 1(b), three logical components are identified for this sample system.

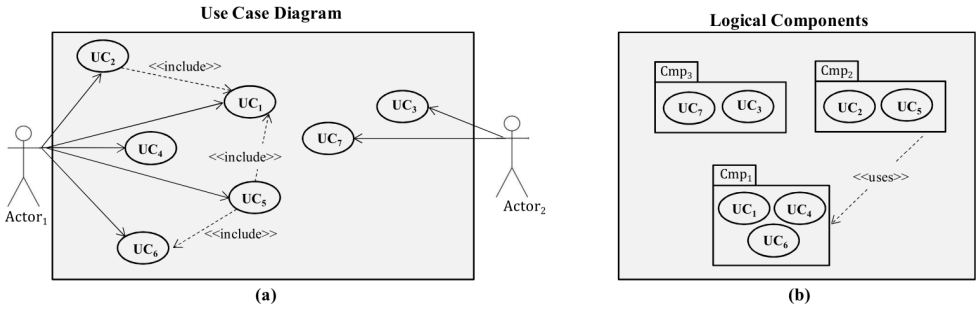


Figure 1 - (a) An example use case model and (b) the corresponding logical components of use cases of part (a) as an example

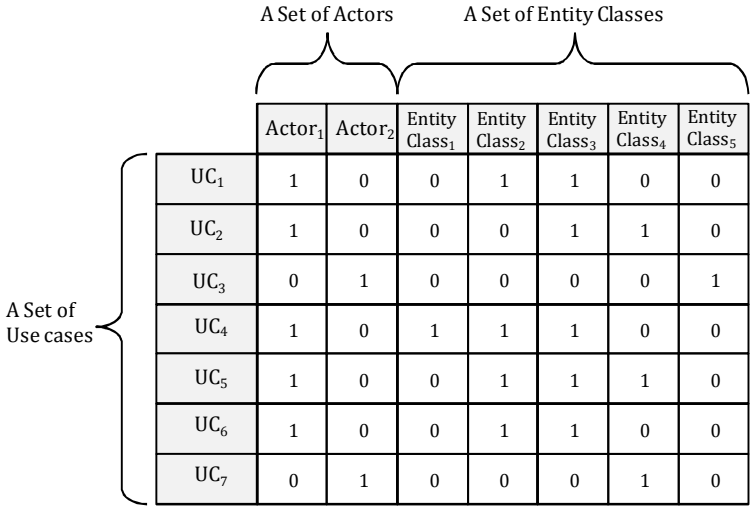
3 Software Design Measurements

Maintainability and reusability are two important factors in component identification. For this reason, SCI-GA employs software cohesion, coupling and complexity metrics, which will be defined in below.

3.1 Cohesion

To compute software cohesion, in this paper, we propose a metric based on similarity between a pair of use cases. In [SJH10], a use case is quantitatively represented by a feature vector. Among the properties of each use case presented in [SJH10], we only use two important property sets, i.e., actors and entity classes, because these property sets have major impacts according to the sensitivity analysis performed in [SJH10], and the preprocessing cost of making all properties available in [SJH10] is too high. In our feature-based representation, a "use case/property" matrix, called F, is created, where all use cases are listed in rows and all entity classes and actors in columns, respectively. In this matrix, each F_{ij} is either 0 or 1, i.e., 1 entry denotes the i^{th} use case (UC_i) has a relationship with property j ($EntityClass_j$ or $Actor_j$) and 0 otherwise. For example, Figure 2 shows a sample F matrix of the use cases introduced in Figure 1. As shown in Figure 2, this

system has seven use cases, two actors, and five entity classes; hence, F will be a binary matrix with 7 rows and 7 columns.



	$Actor_1$	$Actor_2$	Entity $Class_1$	Entity $Class_2$	Entity $Class_3$	Entity $Class_4$	Entity $Class_5$
UC_1	1	0	0	1	1	0	0
UC_2	1	0	0	0	1	1	0
UC_3	0	1	0	0	0	0	1
UC_4	1	0	1	1	1	0	0
UC_5	1	0	0	1	1	1	0
UC_6	1	0	0	1	1	0	0
UC_7	0	1	0	0	0	1	0

Figure 2 - a sample "use case/property" matrix for the use cases of Figure 1

There are many similarity measurements to measure the similarity between a pair of vectors, in terms of features [RW10]. Based on experiments in [SJH10], we choose *Simple coefficient* (see Equation 1) to compute the similarity between two use cases. Let n_{11} denote the number of features present in both UC_i and UC_j, n_{10} denote the number of features present in UC_i but not UC_j, n_{01} denote the number of features present in UC_j but not UC_i, and n_{00} denote the number of features not present in both UC_i and UC_j.

$$Sim\ (UC_i, UC_j) = (n_{11} + n_{00})/(\ n_{11} + n_{00} + n_{10} + n_{01}) \quad (1)$$

Table 1 shows a matrix in which all similarities among use cases of Figure 2 are computed in terms of Equation (1). For example, the similarity between UC_4 and UC_5 is 0.714, because according to Figure 2, n_{11}, n_{00}, n_{10}, and n_{01} values for these use cases are 3, 2, 1, and 1, respectively.

There is no component cohesion measurement based on use case model. Consequently, according to *Cohesion Ratio* (*CR*) idea [Bal96]: Q/(P+Q), where the number of pairs with similarity (*Q*) is divided by the total number of pairs, we propose a new *Component Cohesion* (*CC*) measurement based on a use case model defined as Equation (2).

Table 1 - The similarity matrix of the "use case/property" matrix presented in Figure 2

	UC_1	UC_2	UC_3	UC_4	UC_5	UC_6	UC_7
UC_1	1	0.714	0.286	0.857	0.857	1	0.286
UC_2	0.714	1	0.286	0.571	0.857	0.714	0.571
UC_3	0.286	0.286	1	0.143	0.143	0.286	0.714
UC_4	0.857	0.571	0.143	1	0.714	0.857	0.143
UC_5	0.857	0.857	0.143	0.714	1	0.857	0.429
UC_6	1	0.714	0.286	0.857	0.857	1	0.286
UC_7	0.286	0.571	0.714	0.143	0.429	0.286	1

$$CC\ (cmp_c) = \begin{cases} 1 & if\ \ m_c = 1 \\ \dfrac{\sum_{\forall\, UC_i \in cmp_c} \ \sum_{\forall\, UC_j \in cmp_c,\ UC_j \neq UC_i} Sim\ (UC_i, UC_j)}{\binom{m_c}{2}} & otherwise \end{cases} \quad (2)$$

Where $CC(cmp_c)$ is the cohesion of a component cmp_c and m_c is the number of use cases in cmp_c. In other words, to compute the cohesion of component cmp_c, the summation of similarities between all pairs of its use cases is divided into maximum interactions between them ($\binom{m_c}{2}$). An example of this metric is presented below.

Take Figure 1(b) and Table 1 as an example. The *CC* values of components cmp_1, cmp_2, and cmp_3 are 0.905, 0.857, and 0.714, respectively. For example,

$$CC\ (cmp_1) = \frac{Sim\ (UC_1,UC_4) + Sim\ (UC_1,UC_6) + Sim(UC_4,UC_6)}{\binom{3}{2}} = \frac{0.857+1+0.857}{3} = 0.905.$$

The *CC* value of a component lies in the range [0,1] and if a component has only one use case, its *CC* value equals to 1. A component with a higher *CC* value is better than one with a smaller *CC* value. For evaluating the overall software cohesion, we use Equation (3), where n is the number of components.

$$SoftwareCohesion = \sum_{c=1}^{n} \frac{CC\ (cmp_c)}{n} \quad (3)$$

For example, the *SoftwareCohesion* value of identified components of Figure 1(b) is equal to 0.825.

3.2 Coupling

Coupling represents how tightly one component interacts with others. There have been several studies on component coupling metrics [CL06, WYF03]. However, most of them are not applicable at an early stage of the software design, because they usually need factors extracted from source codes. We use *CCR* [CC11] defined in Equation (4) to evaluate coupling of a component with others, because it is applicable at use case model and is also more accurate.

$$CCR(cmp_c) = \frac{|CP(cmp_c)|}{|UCMP| - 1} \tag{4}$$

Where $CCR(cmp_c)$ is the coupling of a component cmp_c, $CP(cmp_c)$ and $UCMP$ denote a set of components coupled to cmp_c and a set of components that compose a software system, respectively. Note that in *CCR*, three types of relationships among use cases including <<include>>, <<extend>>, and <<generalization>> are considered. In *CCR*, two components are coupled if there is a relationship between their use cases. The *CCR* value of a component lies in the range [0,1]: one and zero mean that component cmp_c is accessed by all the other components and is entirely independent, respectively. For example, in Figure 1 (b), the *CCR* values of components cmp_1, cmp_2, and cmp_3 are 0, 0.5, and 0, respectively. As shown in Figure 1(b), $|CP(cmp_1)| = 0$, $|CP(cmp_2)| = 1$, $|CP(cmp_3)| = 0$, and $|UCMP| = 3$.

AlSharif et al. [ABA04] have shown that to compute total software coupling, *Euclidean norm distance* formula outperforms *average norm* formula, so we use Equation (5) to compute overall software coupling, which is in the range [0,1].

$$SoftwareCoupling = \sqrt{\frac{\sum_{c=1}^{n}(CCR(cmp_c))^2}{No.\,of\ Use\ cases}} \tag{5}$$

For example, the *SoftwareCoupling* value of identified components of Figure 1(b) is equal to 0.189. It is worth mentioning that softwares with lower *SoftwareCoupling* are more maintainable and reusable.

3.3 Complexity

Although, there are many metrics to compute software complexity, the majority of them such as conventional OO complexity metrics including Chidamber and Kemerer's metrics [CK94] do not incorporate a procedure to account for characteristics of CBSD such as component complexity. However, a few metrics are presented in literature to compute component complexity and most of them are not applicable at an early stage of software design because of the lack of

information. *UCP* (*Use Case Point*) is one of the famous metrics applicable at use case model [Kar93].

UCP is a software complexity measurement and is accomplished in two steps as shown in Equation 6. First, the *Unadjusted UCP* (*UUCP*) count is calculated based on the unadjusted weighted actors and use cases as illustrated in Equations 7, 8, and 9. Second, the *Adjusted UCP* (*AUCP*) count is calculated using technical complexity. Note that we neglect *AUCP* in complexity measurement, because determining this technical complexity metric is an extremely difficult task.

$$UCP = UUCP + AUCP \tag{6}$$

$$UUCP = UAW + UUCW \tag{7}$$

$$UAW = \sum (\text{Complexity Weight}) \times (\#\text{ of Actors associated with Complexity}) \tag{8}$$

$$UUCW = \sum (\text{Complexity Weight}) \times (\#\text{ of Use cases associated with Complexity}) \tag{9}$$

Table 1 - UCP actor types and complexity weights

Actor Type	Description	Weight
Simple	Program interface	1
Average	Interactive, or protocol-driven interface	2
Complex	Graphical interface	3

Table 2 - UCP use case types and complexity weights

Use Case Type	Description	Weight
Simple	Fewer than 5 analysis classes	5
Average	5 to 10 analysis classes	10
Complex	More than 10 analysis classes	15

Unadjusted Actor Weight (*UAW*): An actor in a use case can be a person, a software program or a hardware device. Then, *UAW* is computed based on three actor types shown in Table 2 with complexity weights.

Unadjusted Use Case Weight (*UUCW*): The complexity level of the use cases is primarily derived from the number of analysis classes. Then, *UUCW* is computed based on three use case types shown in Table 3 with complexity weights.

For example, if we take Figure 2 as an example, Table 4 shows the *UCP* value of each of the use cases. We suppose that both actors in Figure 2 are graphical interfaces and each use case has one boundary class and one control class except for its entity classes.

Table 3 - The UCP value of each use case of Figure 2

	UAW	***UUCW***	***UCP***
UC_1	3	5	8
UC_2	3	5	8
UC_3	3	5	8
UC_4	3	10	13
UC_5	3	10	13
UC_6	3	5	8
UC_7	3	5	8

We propose a new component complexity measurement based on *UCP* metric defined as:

$$ComponentComplexity\ (cmp_c) = \sum_{\forall\ UC_i \in cmp_c} \frac{UCP_i}{m_c \times TotalUCP} \tag{10}$$

Where $ComponentComplexity(cmp_c)$ and m_c are the complexity of a component cmp_c and the number of use cases in cmp_c, respectively. In addition, UCP_i and $TotalUCP$ denote the complexity of the i^{th} use case and the summation of all UCP_i, respectively. Moreover, to compute overall software complexity, we employ Equation (11) according to AlSharif et al. [ABA04] idea about *Euclidean norm distance* formula, which is in the range [0,1].

$$SoftwareComplexity = \sqrt{\frac{\sum_{c=1}^{n}(ComponentComplexity\ (cmp_c))^2}{No.of\ Use\ cases}} \tag{11}$$

Where n denotes the number of components in the system. Take Figure 1(b) and Table 4 as an example. The *ComponentComplexity* values of components cmp_1, cmp_2, and cmp_3 are 0.146, 0.159, and 0.121, respectively. For example, $ComponentComplexity(cmp_1) = \frac{UCP_1 + UCP_4 + UCP_6}{m_1 \times TotalUCP} = \frac{8+13+8}{3 \times 66} = 0.146$. In addition, the *SoftwareComplexity* value of identified components of Figure 1(b) is equal to 0.094. It is obvious that the value of *SoftwareComplexity* for each system is over zero; therefore, logical components with lower complexity are more desirable and maintainable.

4 The SCI-GA Algorithm

To discover the best use cases grouping, i.e., logical components identification, we have to consider

$$n(N,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i \binom{k}{i} (k-i)^N \quad (12)$$

possibilities, where N is the total number of use cases and k is the number of components [HE03]. For example, there are $n(25,5) = 2 \times 10^{15}$ different ways of grouping 25 use cases into 5 logical components. Therefore, when k is known, it is not easy to identify the best components. If k is unknown; then, we face to $\sum_{k=1}^{N} n\,(N,k)$ possibilities. For example, considering 25 use cases, this number represents approximately 4×10^{18} different component identifications [HE03]. Thus, the component identification problem is a NP-complete problem [CYW11], because the number of different ways of grouping N use cases into k components increases approximately as $k^N/k!$ [HE03].

In [SJH10], we identified logical components using classical clustering techniques, but in this paper, the SCI-GA algorithm is proposed to extend [SJH10]. Our contributions in this paper in comparison with [SJH10] are mentioned as follows. First, SCI-GA uses a GA-Based algorithm as a powerful optimization search algorithm to identify components instead of a heuristic like *k-means*, so that GA has been used successfully for tackling large and complex search spaces like NP-complete problems [HCF09]. Second, SCI-GA aims at automatically finding a near-optimal number of logical components. Finally, SCI-GA proposes a novel fitness function defined in Section 4.2 to make trade-off among software cohesion, coupling, and complexity of system components.

Figure 3 shows the SCI-GA algorithm to identify logical components. The inputs of SCI-GA are a use case model with similarity matrix like Table 1 and *UCP* matrix like Table 4. The outputs of SCI-GA are the proper number of components and identified logical components. In SCI-GA, the initial chromosomes of population are first generated randomly. Then, for each chromosome, the fitness is evaluated according to a fitness function defined in Section 4.2. Then, some chromosomes for reproduction are selected as parents using the roulette wheel selection scheme [Mic96]. After selecting some parent chromosomes, one of the three crossover operators defined in Section 4.3.1 is randomly applied on all pairs of parents to generate two children. Then, one of the two mutation operators defined in Section 4.3.2 is randomly applied on each generated offspring. After applying SCI-GA operators, the consistency of each offspring is evaluated and 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.

4.1 Use Case Encoding in Chromosomes and Population Initialization

To encode the entire logical components of a system in a chromosome, we propose a novel encoding scheme, in which for each component cmp_c, a use case, called *centroid use case* (CUC_c), is considered as a representative of other use cases belonging to that component. In the proposed encoding scheme, each component identification solution is represented as a binary string of N length, where N is the total number of use cases in the system. Each position of the binary string corresponds to a particular use case, i.e., the i^{th} position (gene) represents the UC_i. The value of the i^{th} gene is 1 if the UC_i is a *centroid use case* and zero otherwise. Therefore, the number of "1" in the binary string of a chromosome shows the number of components. For example, the components depicted in Figure 4 can be encoded by means of the string [0111000], in which UC_2, UC_3, and UC_4 are *centroid use cases* of components cmp_2, cmp_3, and cmp_1, respectively. Each UC_i that is not a *centroid use case*, is assigned to cmp_c that UC_i has the highest similarity to CUC_c in comparison with other *centroid use cases* according to Equation (13). Take Figure 4 and Table 1 as an example. According to Figure 4, UC_1, UC_5, UC_6, and UC_7 are not *centroid use cases* and must be assigned to one of the three components with UC_2, UC_3, and UC_4 as *centroid use cases*. For example, for UC_1, among three similarities between UC_1 and each of three *centroid use cases,* i.e., $Sim(UC_1, UC_2)$, $Sim(UC_1, UC_3)$, and $Sim(UC_1, UC_4)$, the value of $Sim(UC_1, UC_4)$ is the highest, so UC_1 is assigned to cmp_1.

$$c = argmax\ (\forall\ CUC_c\ \ Sim(CUC_c, UC_i)) \tag{13}$$

For the initial population, each chromosome is randomly generated by SCI-GA in such a way that the number of 1's in each chromosome is uniformly distributed within [1, $k_{\max}$], where k_{max} is a user-defined maximum number of components that can be determined by software architects to apply their preferences. It should be noted that the default value of k_{max} is the total number of use cases.

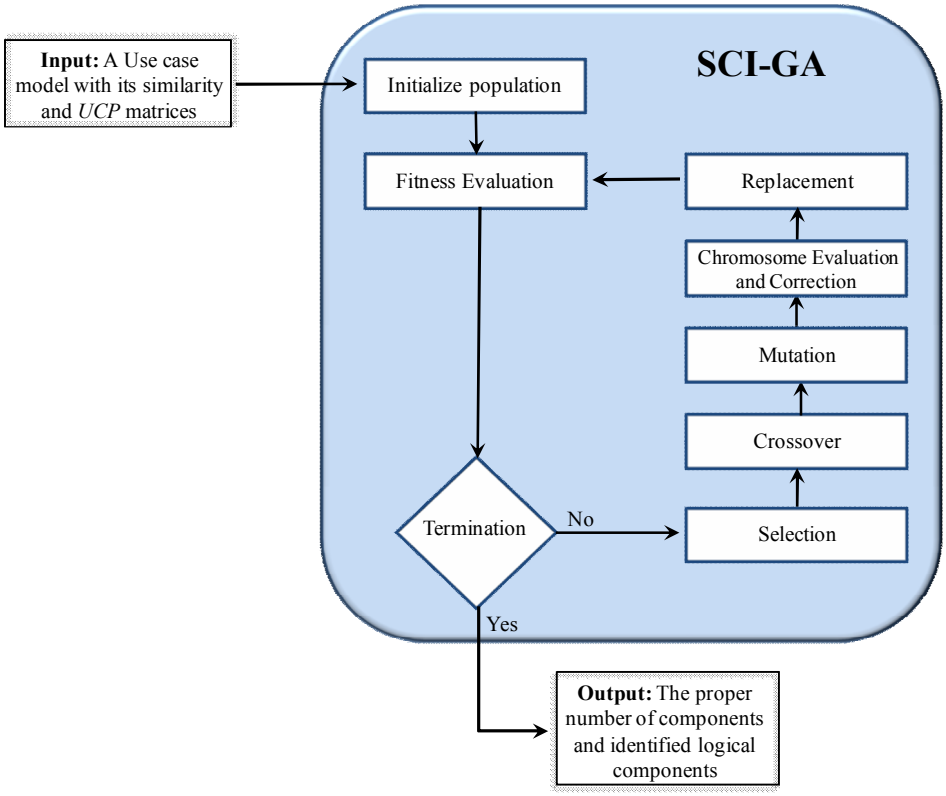


Figure 3 - The flowchart of the SCI-GA algorithm

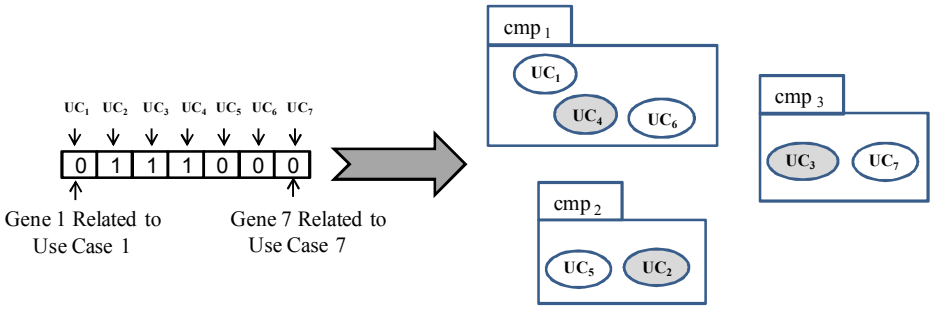


Figure 4 - SCI-GA encoding

4.2 Objective Function

The input of SCI-GA includes the use case model used in chromosome encoding and analysis class diagrams used in computing the SCI-GA fitness function. In the SCI-GA algorithm, we simultaneously employ the *SoftwareCohesion*, *SoftwareCoupling*, and *SoftwareComplexity* metrics (defined in Section 3) as a

fitness function. However, in the literature, few works propose a single function to evaluate the quality of logical components. Making logical components, Choi et al. [CKH09] proposed a single function called *Independence Degree of a System* (*IDS*) that is computed by the expression (*SoftwareCohesion* - *SoftwareCoupling*). When *IDS* is high, it consists of more independent components. Based on *IDS* idea and with respect to the *SoftwareComplexity* metric, we propose a new *Fitness Function* (*FF*) defined in Equation (14) to maximize the overall software cohesion and minimize the overall software coupling and complexity.

$$FF = SoftwareCohesion - (SoftwareCoupling + SoftwareComplexity) \quad (14)$$

The maximum value of *FF* denotes that highly cohesive and loosely coupled components with the least complexity have been obtained.

4.3 Selection and Reproduction

Reproduction in SCI-GA consists of applying both crossover and mutation operators. In SCI-GA, two chromosomes are selected as parents for crossover, using the roulette-wheel selection scheme [Deb01], so that each parent's chance of selection is directly proportional to its fitness.

4.3.1 Crossover

After selecting some chromosome for reproduction, some pairs of them are randomly selected to produce offspring chromosomes. SCI-GA uses three standard crossover operators [Deb01]: one-point, two-point, and uniform crossover operators. In one-point crossover, a position in the chromosome is randomly selected as parts of two parents after the crossover position are exchanged. In two-point crossover, two positions are randomly chosen and the parts between them are exchanged.

In uniform crossover, a mask binary vector is first generated at random. Suppose that a mask such as [0110001] is generated; then, the values of the 2nd, 3rd and 7th genes are taken from the second parent to copy in the corresponding genes of the first offspring, and the others are taken from the first parent. For the sake of illustration, an example of these operators is shown in Figure 5. It should be noted that SCI-GA uses one of these three crossover operators at random for each pair of chromosomes.

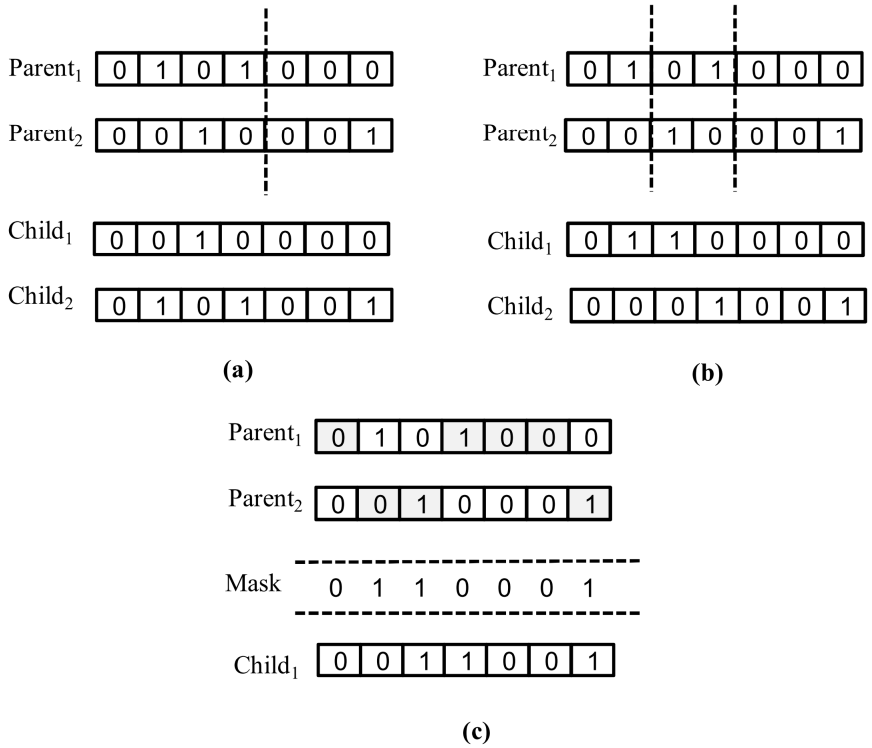


Figure 5 - Example of crossover operators: (a) one-point, (b) two-point, and (c) uniform crossover

4.3.2 Mutation

After crossover, the offspring are mutated to avoid getting trapped at local optima on one hand and to ensure diversity on the other hand. SCI-GA uses two novel mutation operators including *Eliminate Mutation* and *Add Mutation* operators. SCI-GA uses *Eliminate Mutation* and *Add Mutation* operators to allow for the number of components to be changed dynamically as the evolutionary process progresses. Therefore, the number of components does not need to be specified by software architects in advance.

In *Add Mutation*, one chromosome is selected; then, it randomly changes value of a gene, i.e., its value is changed from "0" to "1". By applying *Add Mutation* to a chromosome, the number of components is increased by one.

In *Eliminate Mutation*, a component is randomly chosen and eliminated, i.e., the value of its *centroid use case* is set to zero. Algorithm 1 presents the pseudo code of the *Eliminate Mutation* operator. As shown in Algorithm 1, a candidate component can be a component, which has the highest *CCR* or *ComponentComplexity* values or the one with the lowest *CC* value. By applying

Eliminate Mutation to a chromosome, the number of components is decreased by one. It is expected that when a component with the highest *ComponentComplexity* value is eliminated and the use cases belonging to it are assigned to other components, the value of *SystemComplexity* is decreased. However, this decrease is not an inclusive event. It should be noted that SCI-GA uses one of these two mutation operators at random for each chromosome.

Algorithm 1 - The *Eliminate Mutation* description

1)	Cmp_{random} : Select randomly a component belonging to the parent chromosome.
2)	$Cmp_{complex}$: Select a component with the highest *ComponentComplexity* value belonging to the parent chromosome.
3)	$Cmp_{not\text{-}cohesive}$: Select a component with the lowest *CC* value belonging to the parent chromosome.
4)	$Cmp_{dependent}$: Select a component with the highest *CCR* value belonging to the parent chromosome.
5)	$Cmp_{candidate}$ *:* Choose randomly a component among Cmp_{random}, $Cmp_{complex}$, $Cmp_{not\text{-}cohesive}$, and $Cmp_{dependent}$.
6)	The corresponding gene value of *centroid use case* of $Cmp_{candidate}$ is set to zero.

4.4 Component Evaluation and Correction

The main challenge of applying genetic operators to chromosomes is that invalid component solutions may be produced. An identified component solution is invalid if the number of components is less than 1 or more than k_{max}. To illustrate this point, let us apply the one-point crossover to both chromosomes [0100011] and [1000000], as displayed in Figure 6 (bold type refers to the exchanged genetic information). It is not difficult to see that similar problems may occur under the other used crossover operators and introduced mutation operators.

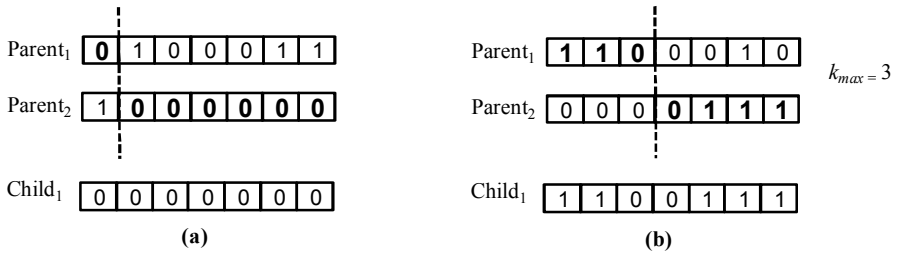


Figure 6 - Examples of invalid identified components after applying one-point crossover: (a) the number of components of $Child_1$ is 0 and (b) the number of components of $Child_1$ is more than k_{max}

To solve this problem, we propose four methods as follows:

a. ***Omission***: In this method, after applying each operator, if an invalid offspring is generated, it is omitted from the population, and does not participate in the next generation.

b. ***Applying Again***: In this method, if an invalid offspring is produced, because of applying crossovers on a pair of parents or mutations on a parent, this offspring is omitted and genetic operators are applied again on those parents.

c. ***Punishment***: In this method, if the produced offspring is invalid, its fitness is decreased. The motivation of this method is that when the fitness of an invalid offspring is decreased, the chance of its presence in the next generation is extremely decreased, but in contrast to *omission method*, participation in the next generation is possible. To decrease the fitness of an invalid offspring, we need a penalty function, which is described in Equation (15).

$$FF_{new} = \frac{FF_{old}}{P+1} \qquad P = \begin{cases} 1 & NIC = 0 \\ NIC - k_{max} & NIC > k_{max} \end{cases} \tag{15}$$

Where FF_{old} and FF_{new} are the initial fitness of an invalid offspring and its fitness after punishing, respectively. Moreover, NIC denotes the number of "1" of the invalid offspring. For example, in Figures 6(a) and 6(b), the values of P are 1 and 2 (i.e., 5-3), respectively.

d. ***Correction***: The goal of this method is to correct an invalid offspring; therefore, it uses *Eliminate* and *Add mutations* described in Section 4.3.2 for this correction. If the number of components of the invalid offspring is zero, the *Add mutation* operator is applied on it. Moreover, if the number of components of the invalid offspring is higher than k_{max}, the *Eliminate mutation* operator is applied P (according to Equation 15) times on it.

We evaluate these methods in Section 5.2 and the efficient method is derived from experiments.

4.5 Replacement and Termination Conditions

SCI-GA uses the roulette-wheel replacement scheme [Deb01] to replace new members of the current population to the old ones, in which a chromosome with the higher fitness has more chance to survive in the next generation.

For the sake of simplicity, the generation step is stopped when the number of iterations exceeds the limit or when the best fitness value does not improve during some generations.

5 Experimental Results

We implemented SCI-GA and set its parameters by performing empirical studies [Gre86] as follows: population size was 100, crossover and mutation rates were 70%, and 2%, respectively. In addition, the SCI-GA algorithm was stopped when the generation number reached 1000 or the fitness value did not improve during the last 50 generations.

For measuring the performance of SCI-GA, we have applied it on an *Online Broker System* (OBS), derived from a number of established Internet-based broker systems [CSC13], a *Restaurant Automation System* (RAS) [CSC13] and *AgriInsurance System* [AIF13]. The analysis models of both OBS and RAS cases are reported in [CSC13], and are comprised of 30 and 32 use cases, 4 and 6 actors, 22 and 25 analysis classes, and 6 and 10 entity classes, respectively.

Note that *AgriInsurance 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, one of the famous software house companies in Iran, for *Agriculture Bank*, the Iranian bank customized to agriculture finance for Iranian farmers. This system is developed by more than 10 professional developers with the average of 4 years of experience and comprises 68 use cases, 27 actors, 233 analysis classes, and 154 entity classes. According to Equation (12), to identify the best logical components, we have to consider $\sum_{k=1}^{30} n\,(30,k)$, $\sum_{k=1}^{32} n\,(32,k)$, and $\sum_{k=1}^{68} n\,(68,k)$ possibilities for OBS, RAS and *AgriInsurance System* cases, respectively. It seems that these problem spaces are enough to show the SCI-GA effectiveness.

It should be noted that for OBS and RAS cases, 3 professional developers with the average of 5 years of experience, are considered as experts, and one of them with more experience than others combines solutions of all developers to reach one solution. It is worth mentioning that the goal of all experts is to increase software cohesion and decrease software coupling and complexity.

For comparison, *CRUD-Based* [LYC99], *Clustering-Based* [SJH10], and *FCA-Based* [CYW11] methods are considered. We implemented these methods and applied them on OBS, RAS and *AgriInsurance System* cases, but due to the space limitation, we only demonstrate the final results of them, and the details of evaluations are reported in [CSC13].

Figures 7 and 8 illustrate the components obtained by SCI-GA for OBS and RAS cases, respectively. Note that in Figures 7 and 8, dark use cases refer to the centroid use cases. For OBS case, comparing the components identified by experts (reported in [CSC13]) with the components identified by SCI-GA (Figure 7) shows that SCI-GA automatically obtains components which are approximately the same as the ones identified by experts. Indeed, there is only one difference between components identified by experts and SCI-GA: the UC28-Compute Benefit in components identified by experts belongs to Component 6, but to Component 3 in SCI-GA components (see Figure 7). However, the values of *SoftwareCohesion* and *SoftwareComplexity* metrics, i.e., 0.969 and 0.0771, for SCI-GA components are slightly better than the values identified by experts for components (i.e., 0.966 and 0.0779).

For RAS case, comparing the components identified by experts (reported in [CSC13]) with the components identified by SCI-GA (Figure 8) shows that SCI-GA automatically obtains 8 components, as opposed to 7 components determined by experts. In fact, all components identified by both SCI-GA and experts are the same except for Components 2 and 3. However, SCI-GA divides the expert's *Administrator component* into two cohesive components (Components 2 and 3 in Figure 8), because this component is too complex. Indeed, dividing the complex *Administrator component* into two simpler components in the SCI-GA results leads to improve the values of *SoftwareCohesion, SoftwareCoupling*, and *SoftwareComplexity* metrics, i.e., 0.907, 0.138 and 0.0153, respectively, in contrast to the ones identified by experts for the components (i.e. 0.886, 0.144 and 0.0159, respectively).

For *AgriInsurance System* case, comparing the components identified by experts with the components identified by SCI-GA shows that SCI-GA automatically obtains 11 components, as opposed to 9 components determined by experts and including *Administrator, Policy Management, Commissions, Billing, General Ledger, Claims, Reporting, Calculations* and *User Managements components.* However, SCI-GA divides each of the expert's *Administrator* and *Policy Management components* into two simpler components, because these components are too complex. Indeed, dividing each of the complex *Administrator* and *Policy Management components* into two simpler components in the SCI-GA results leads to improve the values of *SoftwareCohesion, SoftwareCoupling*, and *SoftwareComplexity* metrics, i.e., 0.867, 0.111 and 0.0599, respectively, in contrast to the ones identified by experts for the components (i.e., 0.842, 0.112 and 0.0623, respectively).

It should be noted that to evaluate components identified by SCI-GA according to expert opinions, we employ a metric, called *Quality metric* [TH99], which is presented in Equation (16).

$$Q(A, ExpertComponents) = (1 - \frac{MoJo(A, ExpertComponents)}{n}) \times 100 \quad (16)$$

Where $MoJo(A, ExpertComponents)$ computes the minimal number of *Move* and *Join* operations needed to transform solution A into the expert's solution, and n is the total number of use cases of the system. Note that a solution with higher *Quality Metric* value has higher similarity with the expert's solution than that with a lower value.

Table 5 compares the final results of applying various methods on OBS, RAS and *AgriInsurance System* cases in terms of the number of components identified by each method, the number of different use cases in components identified by each method compared with experts and the value of *Quality metric* for each solution. As shown in Table 5, SCI-GA outperforms the other methods, and has the closet results to the ones identified by experts.

For RAS case, although the number of different use cases of components identified by SCI-GA compared with experts is 6, but the reason of this case is that SCI-GA identifies 8 simpler and cohesive components for RAS case (see Figure 8) in comparison with 7 components identified by experts [CSC13]. However, when we use expert opinions in SCI-GA to determine the number of components for RAS case in advance (i.e., $k_{\max}$ is set to 7), SCI-GA identifies a solution with 7 components, which has only 4 different use cases compared with experts. Moreover, for *AgriInsurance System,* SCI-GA automatically identifies 11 components and when we use expert opinions to determine the number of components in advance (i.e., $k_{\max}$ is set to 9), it identifies a solution with only 6 different use cases compared with the expert's solution. The key point in Table 5 is that SCI-GA automatically identifies the number of components, which is approximately the same as expert opinions, as opposed to other methods, which must be given in advance.

Table 5 - Results of different methods applied on OBS, RAS and *AgriInsurance System* case studies according to the number of components, the number of different use cases compared with experts and *Quality metric*

Case Study	Method	No. of Components	No. of different use cases compared with experts	*Quality metric*
OBS	COMO (CRUD-Based) [LYC99]	6	12 from 30 use cases	60
	Shahmohammadi et al. (Clustering-based)[SJH10]	6	5 from 30 use cases	83
	Cai et al. (FCA-Based) [CYW11]	5	4 from 30 use cases	87
	SCI-GA (Evolutionary)	**6**	**1** from 30 use cases	**97**
	Expert	**6**	0	100
RAS	COMO (CRUD-Based) [LYC99]	7	17 from 32 use cases	47
	Shahmohammadi et al. (Clustering-based)[SJH10]	7	10 from 32 use cases	69
	Cai et al. (FCA-Based) [CYW11]	7	7 from 32 use cases	78
	SCI-GA (Evolutionary) with 7 components determining by experts in advance	**7**	**4** from 32 use cases	**88**
	SCI-GA (Evolutionary)	**8**	**6** from 32 use cases	**81**
	Expert	**7**	0	100
AgriInsurance System	COMO (CRUD-Based) [LYC99]	9	28 from 68 use cases	59
	Shahmohammadi et al. (Clustering-based)[SJH10]	9	19 from 68 use cases	72
	Cai et al. (FCA-Based) [CYW11]	9	16 from 68 use cases	76
	SCI-GA (Evolutionary) with 9 components determining by experts in advance	**9**	**6** from 68 use cases	**91**
	SCI-GA (Evolutionary)	**11**	**9** from 68 use cases	**87**
	Expert	**9**	0	100

Table 6 shows the values of *SoftwareCohesion*, *SoftwareCoupling*, *SoftwareComplexity*, and *FF* metrics for the final results of applying various methods on OBS and RAS cases. As shown in Table 6, for three metrics introduced in Section 3 and *FF* metric, SCI-GA outperforms the other methods. Accordingly, it is concluded that SCI-GA has far better performance in searching component space than classical clustering techniques used in [SJH10], and is able to achieve

near-optimal logical components, which are better than the components identified by experts in terms of all the four metrics used. An important point in Table 6 is that SCI-GA with the highest values for *FF* metric in all three cases has the closest results to components identified by experts in comparison with the other methods. Moreover, comparing results of the other methods [CSC13] with the components identified by experts reveals that when a solution for logical components has a higher *FF* value, it is more desirable and closer to expert opinions. Consequently, it is concluded that *FF* metric is a suitable metric to evaluate logical components, and is a metric close to expert opinions.

It is worth mentioning that Cai et al. work [CYW11] on OBS case has a main limitation in choosing the number of components, i.e., this work cannot identify more than five components for OBS case [CSC13].

Table 6 - Results of different methods applied on OBS, RAS and *AgriInsurance System* case studies according to *SoftwareCohesion*, *SoftwareCoupling*, *SoftwareComplexity,* and *FF* metrics

Case Study	Method	*SoftwareCohesion* Metric	*SoftwareCoupling* Metric	*SoftwareComplexity* Metric	*FF* Metric
OBS	COMO (CRUD-Based) [LYC99]	0.825	0.171	0.079	0.575
	Shahmohammadi et al. (Clustering-based) [SJH10]	0.927	0.163	0.083	0.681
	Cai et al.(FCA-Based) [CYW11]	0.919	0.183	0.087	0.649
	SCI-GA (Evolutionary)	**0.969**	**0.159**	**0.0771**	**0.7329**
	Expert	**0.966**	**0.159**	**0.0779**	**0.7291**
RAS	COMO (CRUD-Based) [LYC99]	0.846	0.169	0.0145	0.663
	Shahmohammadi et al. (Clustering-based) [SJH10]	0.859	0.154	0.0164	0.688
	Cai et al.(FCA-Based) [CYW11]	0.864	0.156	0.0140	0.694
	SCI-GA (Evolutionary)	**0.907**	**0.138**	**0.0153**	**0.7537**
	Expert	**0.886**	**0.144**	**0.0159**	**0.7261**
AgriInsurance System	COMO (CRUD-Based) [LYC99]	0.764	0.187	0.0785	0.449
	Shahmohammadi et al. (Clustering-based) [SJH10]	0.821	0.167	0.0873	0.567
	Cai et al.(FCA-Based) [CYW11]	0.792	0.145	0.0781	0.569
	SCI-GA (Evolutionary)	**0.867**	**0.111**	**0.0599**	**0.6961**
	Expert	**0.842**	**0.112**	**0.0623**	**0.6677**

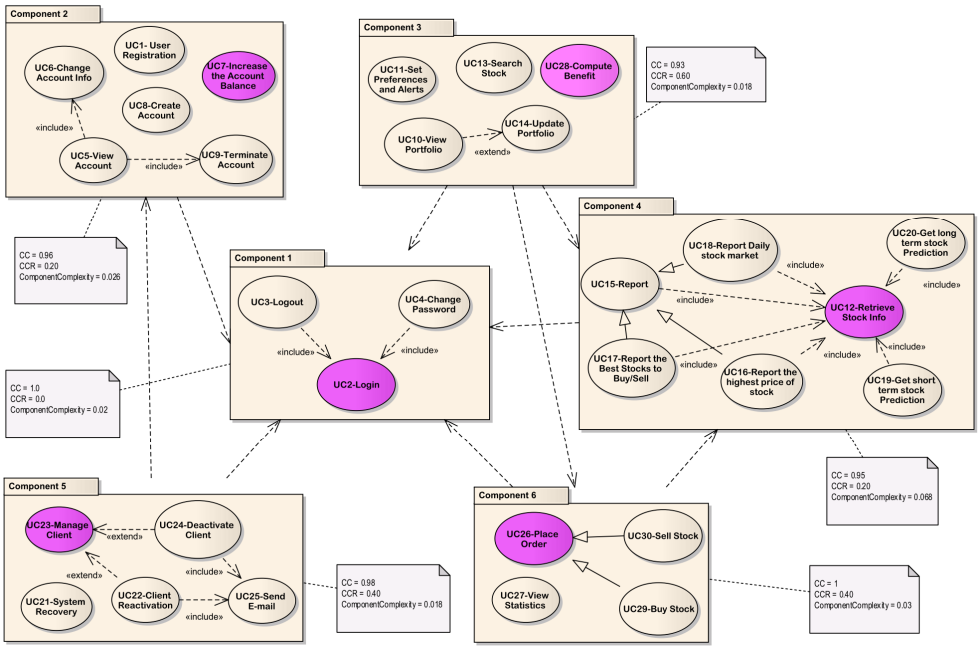


Figure 7 – Obtained components for OBS case by SCI-GA

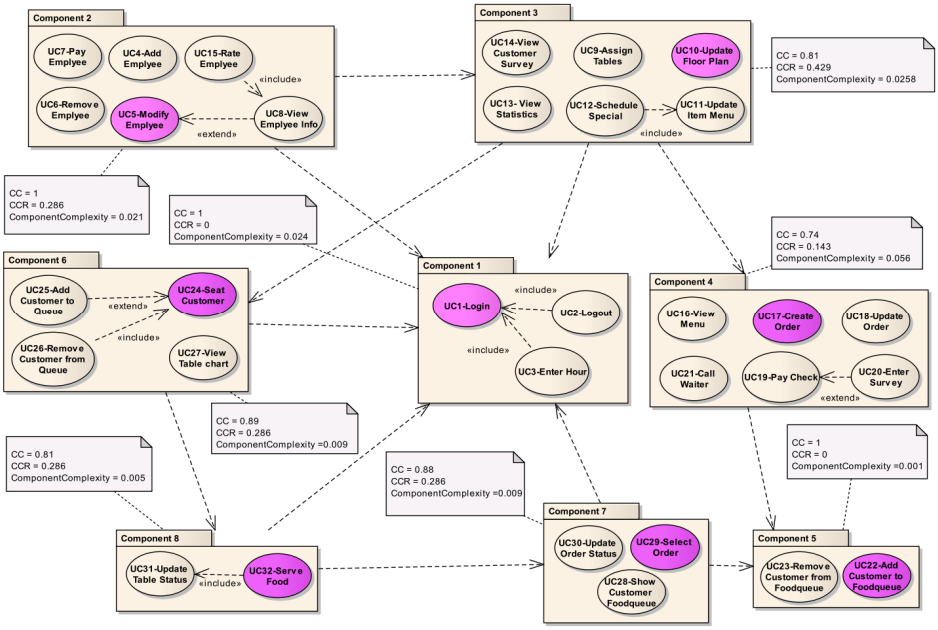


Figure 8 – Obtained components for RAS case by SCI-GA

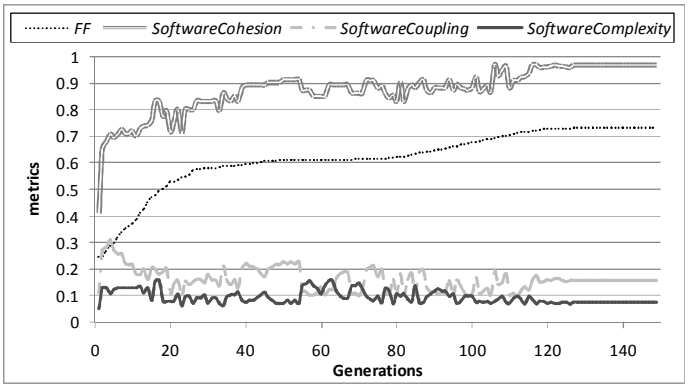
5.1 SCI-GA Effectiveness

To clearly show the SCI-GA effectiveness, Figure 9 demonstrates the best obtained values of the *SoftwareCohesion*, *SoftwareCoupling*, *SoftwareComplexity*, and *FF* metrics in each generation for OBS, RAS and *AgriInsurance System* cases. As shown in Figure 9, the value of *FF* metric is strictly increasing as the number of generations grows, while the values of other metrics are not necessarily increased. The upper curve in Figure 9(a) shows that the *SoftwareCohesion* climbs to a peak of 0.970 at approximately 120 generations, where as *FF, SoftwareCoupling, SoftwareComplexity* metrics, and the number of components of this solution (i.e., the identified components) are 0.723, 0.176, 0.071, and 6, respectively. In this solution, the value of *SoftwareCohesion* is slightly higher than the value (i.e., 0.969) in the best result obtained by SCI-GA (presented in Figure 7). However, the value of *SoftwareCoupling* is higher than the value (i.e., 0.159) in the best result obtained by SCI-GA. Therefore, as the value of *FF* metric for this solution is lower than the best result obtained by SCI-GA, it is not the best solution.

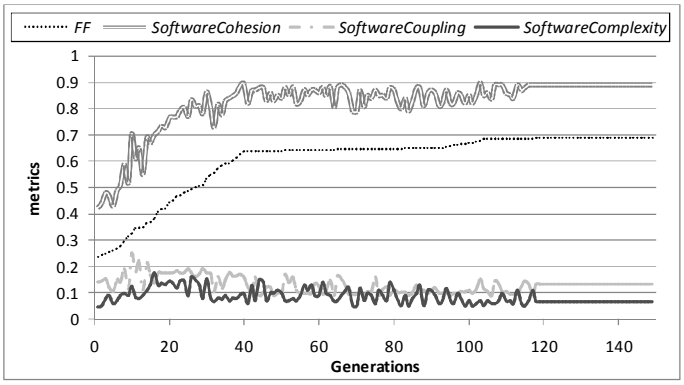
Considering all case studies reveals that when *SoftwareComplexity* is not considered in *FF* metric, the obtained results are not encouraging. For example, in the evolution process of RAS case, there is a solution with 0.875, 0.099, 0.132, and 4 for *SoftwareCohesion*, *SoftwareCoupling*, *SoftwareComplexity*, and the number of components, respectively. Therefore, when the original *FF* metric (Equation 14) is considered, the best components obtained (presented in Figure 8) with $FF = 0.689$ are much better than this solution with $FF = 0.644$. However, when *SoftwareComplexity* is omitted in Equation (14), i.e., $FF = (SoftwareCohesion - SoftwareCoupling)$, this solution with $FF = 0.776$ is better than the best obtained components with $FF = 0.758$. Accordingly, it is concluded that the proposed *FF* metric is practically able to achieve a good trade-off among *SoftwareCohesion*, *SoftwareCoupling* and *SoftwareComplexity* metrics, i.e., identified cohesive components with loosely interconnections and low complexity.

5.2 SCI-GA Component Evaluation

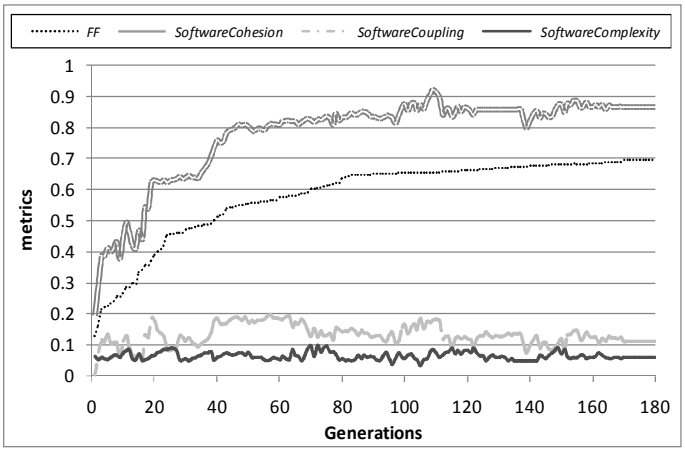
To solve invalid component solution, four methods are proposed in Section 4.4. In this section, we report the results of applying the four proposed methods presented in Section 4.4 on three used case studies. Table 7 shows the average time of one iteration, average number of iterations to convergence, average time to convergence, and the best identified *FF* for each method. Note that all of these experiments are performed with a PC with 2.80 GHz Intel Core i7 CPU and 16 GB RAM. As shown in Table 7, using *Omission* method leads to achieve at least average time of one iteration, but its best *FF* is lower than the other methods.



(a)



(b)



(c)

Figure 9 –The effectiveness of SCI-GA for (a) OBS, (b) RAS, and (c) *AgriInsurance System* case studies

Therefore, the main advantage of the use of *Omission* method is its simplicity; however, its main shortcoming is that it is not able to achieve suitable results. Using *Applying Again* method leads to have enormous computational cost, and is not able to achieve a suitable result according to the *FF* metric. It should be noted that when *Applying Again* method is used in these experiments, there are some cases in which GA operators are called more than 50 times, so this method wastes a lot of time in the evolution process and is not suggested to use. As shown in Table 7, although *Punishment* method achieves plausible results for OBS case, but it is not able to identify good results for both RAS case and *AgriInsurance System*. The reason for this seems to be that determining an effective penalty function is an extremely difficult task [Sal09], and the penalty function used in *Punishment* method is not good for different case studies. Note that to achieve an effective penalty function, adaptive penalty functions like [PB07] should be suggested. As shown in Table 7, *Correction* method achieves the best *FF* in comparison with the other methods in all three case studies. In addition, the experiments reveal that although the average time for one iteration of *Correction* method is higher than some other methods, its average time to converge is lower than the other methods. The reason for this is that using *Correction* method leads to converge very quickly in comparison to the other methods and it requires less iteration than the other methods. It is concluded that among four methods presented in Section 4.4 to handle invalid solution, *Correction* method achieves better performance than the other methods.

Table 7 - The effectiveness of methods to handle invalid components (Average of 30 runs for OBS and RAS cases, and Average of 10 runs for *AgriInsurance System*)

Case Study	Method to handle invalid components	Average time of one iteration (Second)	Average number of iterations to convergence	Average time to convergence (Second)	The best identified *FF*
OBS	*Omission*	**3.99**	282.4	1126.78	0.60
	Applying Again	7.21	241.6	1741.94	0.63
	Punishment	4.09	184.2	753.38	0.72
	Correction	4.33	**160.1**	**693.23**	**0.73**
RAS	*Omission*	**4.41**	311.3	1372.83	0.59
	Applying Again	8.05	288.4	2321.62	0.61
	Punishment	4.98	193.3	962.63	0.53
	Correction	5.12	**143.3**	**733.70**	**0.75**
AgriInsurance System	*Omission*	**28.11**	475.4	13363.49	0.62
	Applying Again	85.04	382.5	32665.5	0.62
	Punishment	31.67	331.9	10511.27	0.59
	Correction	41.22	**191.2**	**7881.26**	**0.70**

It is worth mentioning that the best identified components for OBS, RAS and *AgriInsurance System* cases are obtained within 8.6, 10.1 and 117.3 minutes, respectively.

5.3 A Summary of the Evaluations and SCI-GA Limitations

In Table 8, we have compared SCI-GA to other used methods according to experiments preformed and mentioned in Tables 5 and 6. Among the four methods, just *FCA-Based* method [CYW11] and our method consider both the cohesion and coupling simultaneously throughout the identification process. Note that this feature leads to a good trade-off between these metrics.

Table 8 - The qualitative comparison of the component identification methods

Parameter	Method			
	Clustering-Based [SJH10]	CRUD-Based [LYC99]	FCA-Based [CYW11]	Evolutionary (SCI-GA)
Considering both Cohesion and Coupling simultaneously	No	No	Yes	Yes
Determining automatically the No. of components	No	No	No	Yes
Complexity metric	No	No	No	Yes
Need expert's experience	Medium	Medium	Low	Very Low
Precision (Match to expert's omponents)	Medium	Low	Medium	High

Although, all of these methods need to manually determine some parameters, but some of these parameters have an observable impact on performance. For example, in *FCA-Based* method [CYW11], T_D and Ts parameters (note that these are used as thresholds for computing cohesion and coupling) are manually determined based on expert experiences. However, our method needs to determine insignificant parameters, such as, the population number, crossover and mutation rates.

As shown in Table 8, our method has some advantages over other methods, particularly [SJH10]. First, it can not only set the number of components determined by experts a priori, but also automatically identify near-optimal number of components. Second, it uses a powerful optimization search algorithm

(i.e., GA) instead of simple heuristics like *k-means*. Third, it considers the complexity metric along with cohesion and coupling metrics throughout the identification process. Finally, according to our evaluation in Table 5, it identifies the most similar components to the ones identified by experts.

In the course of experimentations during the evaluation, a number of limitations of SCI-GA became apparent. First, using an evolutionary search algorithm leads to increase complexity, particularly time complexity. However, it should be noted that to identify logical components at an early stage of software design, it is not necessary to have a real-time method. Therefore, according to our experiments, SCI-GA identifies logical components during an acceptable period of time. The second limitation of SCI-GA is that similar to other existing methods, it cannot guarantee to achieve an optimal solution, because it is based on a Meta-heuristic method, i.e., GA. However, as discussed in [HCF09], evolutionary search-based methods are able to achieve better performance than simple heuristics like *k-means*. Furthermore, SCI-GA proposes four methods described in Section 4.4 to avoid identifying infeasible solutions.

6 Related Works

As mentioned earlier, software components can be divided into three categories: *Business, Logical,* and *Technical* components. The attempts for automatic identification of logical or business components can be divided into four approaches, *Graph Partitioning* [AOB08, PTZ08], *Clustering-Based* [JCI01, KC04, LJK01, SJH10], *CRUD-based* [GS01, LYC99], and *FCA-Based* [CYW11, Ham09] approach, which are discussed in detail below. Additionally, we use a new approach, called *Evolutionary*, to identify logical components. Note that this approach is used to identify *Technical* components, i.e., software modules, from source codes. For this reason, in the following section, we compare SCI-GA to other evolutionary-based component identification methods.

Graph Partitioning Approach. Albani et al. [AOB08] have 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, they have assigned weights to edges. Finally, the graph is partitioned into components using a heuristic from graph theories. Peng et al. [PTZ08] have transformed the relationship model among business elements to a weighted graph. Then, they have applied 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 have not demonstrated their claim. However, the main limitation of this approach is that weights are manually assigned to edges according to expert experiences.

Clustering-Based Approach. Lee et al. [LJK01] proposed a method for clustering classes into logical components with high cohesion and low coupling. At first, key classes are selected as candidate components; then, other classes are assigned to the components that have the highest level of dependency with them. Identifying key classes is a critical problem, and is manually determined by experts. Jain et al. [JCI01] 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 et al. [KC04] employed use case models, object models and collaboration diagrams to identify 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. [SJH10] 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*, *Hierarchal*, *Graph-based method*, and *Fuzzy C-means* have been examined to achieve good software architecture. All of these *Clustering-Based* methods used classic clustering techniques; however, they may achieve poor components due to their simple heuristics, and they have the problem of determining the best number of components in advance.

CRUD-Based Approach. Lee et al. [LYC99] presented a tool called COMO, in which "use case/class matrix" is created with respect to use case diagrams and class diagrams. It is then partitioned into blocks with tight cohesion as business components. Ganesan and Sengupta [GS01] 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, this approach has a number of limitations similar to *Clustering-Based Approach* due to the use of classical clustering techniques.

FCA-Based Approach. Hamza [Ham09] 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. [CYW11] 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 components. They used dispersion and distance concepts to measure the cohesion and coupling, respectively. However, they used two dispersion and distance thresholds (i.e., T_D and *Ts* thresholds for computing cohesion and coupling, respectively) with high

effect on the performance of their method, which must be manually determined by practical experiences. Moreover, this approach has a number of limitations similar to *Clustering-Based approach*, due to the use of classical clustering techniques.

Evolutionary Approach. Recently, evolutionary algorithms have been widely applied to software problems. Therefore, a new scope of software engineering has appeared by the name of *Search- Based Software Engineering* (SBSE) [HMZ12] to reformulate software problems as optimization problems. In [SP13], all works related to SBSE are categorized, and *Search-Based Design* works are particularly surveyed in [Räi10]. One of the popular scopes of search-based design is module clustering. In this field, source code of a legacy system is clustered into software modules with a high degree of cohesion and a low degree of coupling. SCI-GA like these search-based methods, aims at identifying components with high degree of cohesion and a low degree of coupling, but the main differences between them is their inputs. In fact, the inputs of SCI-GA are a use case model and analysis class diagrams as opposed to the inputs of search-based module clustering (i.e., source codes).

To identify new well-structured modules based on search-based clustering methods, a number of heuristics like hill-climbing [MHH03, MM06], and simulated annealing [MM06] and Meta-heuristics like genetic algorithm [DMM99, PHY11] are employed. Experiments presented in [PHY11] revealed that Meta-heuristic methods outperform simple heuristics like hill-climbing for dealing with complex search space, particularly software clustering search space.

In [DMM99], like SCI-GA, both cohesion and coupling metrics are combined into a single objective fitness function. On the contrary, in [PHY11], Pareto optimality is used to module clustering problem with multi-objective approach. However, the Pareto optimality has several shortcomings [DSK10]. First, it yields a set of solutions, among which software architects have to select one. For example, in [SPG10], an iterative multi-objective genetic algorithm is proposed to identify design classes. In this algorithm, software architects must rank a number of identified solutions in each generation. In practical applications, the use of Pareto optimality leads to a semi-automatic method, and its performance highly depends on experts. Second, the Pareto optimality has generally higher computational costs and is time-consuming. In fact, when the number of objectives is increased, the Pareto optimality approach is not suitable, because it needs more population members and more computations; therefore, its progress is slowed down. Another difference between SCI-GA and all the existing search-based module clustering methods is that they aim at optimizing clustering criteria like SSE, intra-edges and inter-edges of all clusters, in contrast to SCI-GA that aims at maximizing software cohesion and simultaneously minimizing coupling and complexity. Furthermore, unlike SCI-GA, the existing search-based module clustering methods do not take infeasible solutions into account, and do not apply any techniques to handle them.

7 Conclusions

With the advent of the evolutionary approach in software engineering, we are now able to automatically identify logical software components based on a powerful optimization search algorithm. In this paper, we presented a novel method to identify logical components based on the evolutionary approach called SCI-GA. The evolutionary approach supports the logical component identification in searching components space; therefore, it is more accurate than other approaches.

SCI-GA encodes entire components of a system in a *chromosome,* so that each use case is encoded as a *gene*. In this encoding, some use cases are considered as representatives of other use cases. The efficiency of SCI-GA was evaluated by using three real-world OBS, RAS and *AgriInsurance System* case studies; therefore, the evaluation results demonstrated that it outperforms other methods such as *FCA-Based* and *Clustering-Based* methods. Additionally, it has an ability to automatically identify the near-optimal number of logical components for all three case studies (see Table 5), as opposed to the other methods, in which the number of components is manually determined according to the number of components identified by experts. Moreover, in SCI-GA, a novel fitness function was proposed, and the evaluation results revealed that it is a close metric to expert opinions.

In a future work, we intend to use other optimization algorithms like Ant Colony or GA hybrid algorithms to improve the search performance and apply the idea of automatic design pattern selection [HJ99, HJ12] in order to design classes of each component.

ACKNOWLEDGEMENT

The research was supported by Iran Telecommunication Research Centers (ITRC).

References

[ABA04] M AlSharif, WP Bond, and T Al-Otaiby. Assessing the Complexity of Software Architecture. In *Proceedings of the 42nd annual Southeast regional conference, ACM*, pages 98-103, 2004. doi: 10.1145/986537.986562.

[AIF13] Agricultural Insurance Fund. URL:http://www.aiiri.gov.ir/HomePage.aspx?TabID=1&Site=aiiriPortal&Lang=en-US, *Accessed by May 2013*.

[AOB08] A Albani, S Overhage, and D Birkmeier. Towards a systematic method for identifying business components. In *Proceedings of CBSE, LNCS 5282*, pages 262–277, 2008. doi: 10.1007/978-3-540-87891-9_17.

[Bal96] NV Balasubramanian. Object-oriented metrics. In *Proceedings of Software Engineering Conference*, pages 30-34, 1996. doi: 10.1109/APSEC.1996.566737.

[BO09] D Birkmeier and S Overhage. On Component Identification Approaches–Classification, State of the Art, and Comparison. In *Proceedings of CBSE 2009, LNCS 5582,* pages 1-18, 2009. doi: 10.1007/978-3-642-02414-6_1.

[CC11] JF Cui and HS Chae: Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems. *Information and Software Technology.* 53:601-614, 2011. doi: 10.1016/j.bbr.2011.03.031.

[CKH09] M Choi, IJ Kim, J Hong, and J Kim. Component-based metrics applying the strength of dependency between classes. In *Proceedings of ACM symposium on Applied Computing*, pages 530-536, 2009. doi: 10.1145/1529282.1529392.

[CK94] SR Chidamber and CF Kemerer: A metrics suite for object oriented design. *IEEE Transactions on Software Engineering.* 20(6):476-492, 1994. doi: 10.1109/32.295895.

[CL06] M Choi and S Lee. A coupling metric applying the characteristics of components. In *Proceedings of Workshop on Component Based Software Engineering and Software Process Model*, pages 966-975, 2006. doi: 10.1007/11751632_104.

[CSC13] Case Studies of Component Identification Project. URL: http://www.modares.ac.ir/en/Schools/ece/grp/cmp/res/lab/SCSLAB/Project/Project2, *Accessed by May 2013.*

[CYW11] Z-g Cai, X-h Yang, X-y Wang, and A Kavs: A Fuzzy-based Approach for Business Component identification. *Journal of Zhejiang University-SCIENCE C (Computers & Electronics).* 12(9):707-720, 2011. doi: 10.1631/jzus.C1000337.

[Deb01] K Deb. *Multi-objective optimization using evolutionary algorithms*. Chichester, England: John Wiley & Sons, Ltd, 2001.

[DMM99] D Doval, S Mancoridis, and BS Mitchell: Automatic Clustering of Software Systems Using a Genetic Algorithm. In *Proceedings of Int'l Conf. Software Tools and Eng. Practice.* 1999. doi: 10.1109/STEP.1999.798481.

[DSK10] K Deb, A Sinha, PJ Korhonen, and J Wallenius: An Interactive Evolutionary Multiobjective Optimization Method Based on Progressively Approximated Value Functions. *IEEE Transactions On Evolutionary Computation.* 14(5):723-739, 2010. doi: 10.1109/TEVC.2010.2064323.

[Gre86] JJ Grefenstette: Optimization of control parameters for genetic algorithms. *IEEE Trans Syst Man Cybern.* 16(1):122–128, 1986. doi: 10.1109/TSMC.1986.289288.

[GS01] R Ganesan and S Sengupta. O2BC: a Technique for the Design of Component-Based Applications. In *Proceedings of the 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems*, pages 46-55, 2001. doi: 10.1109/TOOLS.2001.941658.

[Ham09] HS Hamza. A Framework for Identifying Reusable Software Components Using Formal Concept Analysis. In *Proceedings of the 6th International Conference on Information Technology: New Generations*, pages 813-818, 2009. doi: 10.1109/ITNG.2009.276.

[HCF09] ER Hruschka, RJGB Campello, AA Freitas, and ACPLF de Carvalho: A Survey of Evolutionary Algorithms for Clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews.* 39(2):133-155, 2009. doi: 10.1109/TSMCC.2008.2007252.

[HE03] ER Hruschka and NFF Ebecken: A genetic algorithm for cluster analysis. *Intell. Data Anal.* 7(1):15-25, 2003. URL: http://iospress.metapress.com/content/adhnkma5h48f1l0q/.

[HJ09] SMH Hasheminejad and S Jalili. Selecting Proper Security Patterns Using Text Classification. In *Proceedings of International Conference on Computational Intelligence and Software Engineering, CiSE 2009*, pages 1-5, 2009. doi: 10.1109/CISE.2009.5363861.

[HJ12] SMH Hasheminejad and S Jalili: Design patterns selection: An automatic two-phase method. *Journal of Systems and Software.* 85(2):408-424, 2012. doi: 10.1016/j.jss.2011.08.031.

[HMZ12] M Harman, SA Mansouri, and Y Zhang: Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR).* 45(4):11, 2012. doi: 10.1145/2379776.2379787.

[JCI01] H Jain, N Chalimeda, N Ivaturi, and B Reddy. Business Component Identification a Formal Approach. In *Proceedings of the 5th IEEE Int. Enterprise Distributed Object Computing Conf.*, pages 183-187, 2001. doi: 10.1109/EDOC.2001.950437.

[Kar93] G Karner. *Resource Estimationfor Objectory Projects*. Objectory Systems, 1993.

[KC04] S Kim and S Chang. A Systematic Method to Identify Software Components. In *Proceedings of the 11th Software Engineering Conf.*, pages 538-545, 2004. doi: 10.1109/APSEC.2004.11.

[KPS08] J Kim, S Park, and V Sugumaran: DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines. *The Journal of Systems and Software.* 81(1):37-55, 2008. doi: 10.1016/j.jss.2007.04.011.

[Kru00] P Kruchten. *The Rational Unified Process An Introduction*. 2nd ed. Addison Wesley, 2000.

[LJK01] JK Lee, SJ Jung, SD Kim, WH Jang, and DH Ham. Component Identification Method with Coupling and Cohesion. In *Proceedings of the*

8th Asia-Pacific Software Engineering Conference, pages 79-86, 2001. doi: 10.1109/APSEC.2001.991462.

[LYC99] SD Lee, YJ Yang, ES Cho, SD Kim, and SY Rhew. COMO: A UML-Based Component Development Methodology. In *Proceedings of the 6th Asia Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society, Los Alamitos*, pages 54-61, 1999. doi: 10.1109/APSEC.1999.809584.

[MHH03] K Mahdavi, M Harman, and RM Hierons. A Multiple Hill Climbing Approach to Software Module Clustering. In *Proceedings of IEEE Int'l Conf. Software Maintenance*, pages 315-324, 2003. doi: 10.1109/ICSM.2003.1235437.

[Mic96] Z Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. New York: Springer-Verlag, 1996.

[MM06] BS Mitchell and S Mancoridis: On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Trans. Software Eng.* 32(3):193-208, 2006. doi: 10.1109/TSE.2006.31.

[PB07] S Parsa and O Bushehrian: Genetic clustering with constraints. *Journal of research and practice in information technology.* 39(1):47-60, 2007. URL: http://ws.acs.org.au/jrpit/JRPITVolumes/JRPIT39/JRPIT39.1.47.pdf.

[PHY11] K Praditwong, M Harman, and X Yao: Software Module Clustering as a Multi-Objective Search Problem. *IEEE Trans. Software Eng.* 37(2):264-282, 2011. doi: 10.1109/TSE.2010.26.

[PTZ08] L Peng, Z Tong, and Y Zhang. Design of Business Component Identification Method with Graph Segmentation. In *Proceedings of the 3rd Int. Conf. on Intelligent System and Knowledge Engineering*, pages 296-301, 2008. doi: 10.1109/ISKE.2008.4730944.

[Räi10] O Räihä: A survey on search-based software design. *Computer Science Review, Elsevier* 4(4):203-249, 2010. doi: 10.1016/j.cosrev.2010.06.001.

[RW10] X Rui and DC Wunsch: Clustering Algorithms in Biomedical Research: A Review. *IEEE Reviews in Biomedical Engineering.* 3:120-154, 2010. doi: 10.1109/RBME.2010.2083647.

[Sal09] S Salcedo-Sanz: A survey of repair methods used as constraint handling techniques in evolutionary algorithms. *Computer Science Review, Elsevier.* 3(3):175-192, 2009. doi: 10.1016/j.cosrev.2009.07.001.

[SJH10] GR Shahmohammadi, S Jalili, and SMH Hasheminejad: Identification of System Software Components Using Clustering Approach. *Journal of Object Technology (JOT).* 9(6):77-98, 2010. doi: 10.5381/jot.2010.9.6.a4.

[SP13] SBSE Publications. URL: *www.sebase.org/sbse/publications*, *Accessed by May 2013.*

[SPG10] CL Simons, IC Parmee, and R Gwynllyw: Interactive, Evolutionary Search in Upstream Object-Oriented Class Design. *IEEE Transactions on Software Engineering*. 36(6):798-816, 2010. doi: 10.1109/TSE.2010.34.

[TH99] V Tzerpos and RC Holt. MoJo: A distance metric for software clusterings. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 187-193, 1999. doi: 10.1109/WCRE.1999.806959.

[WXZ05] Z Wang, X Xu, and D Zhan: A Survey of Business Component Identification Methods and Related Techniques. *International Journal of Information Technology*. 2(4):229-238, 2005.
URL:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.5794&rep=rep1&type=pdf.

[WYF03] H Washizaki, H Yamamoto, and Y Fukazawa. A metrics suite for measuring reusability of software components. In *Proceedings of the 9th International Software Metrics Symposium*, pages 211-223, 2003. doi: 10.1109/METRIC.2003.1232469.

About the author(s)

Saeed Jalili received the Ph.D. degree from Bradford University in 1991 and the M.Sc. degree in computer science from Sharif University of Technology in 1979. Since 1992, he has been associate professor at the Tarbiat Modares University. His main research interests are software testing, software runtime verification and quantitative evaluation of software architecture.
E-mail: Sjalili@modares.ac.ir

Seyed Mohammad Hossein Hasheminejad is a Ph.D. Candidate of computer engineering at Tarbiat Modares University (TMU). He received the M.Sc. degree in Software engineering from TMU in 2009, and the B.Sc. degree in Software engineering from Tarbiat Moalem University in 2007. His main research interests are Formal Methods for Software Engineering, Object-Oriented Analysis and Design, Search-Based Software Engineering, and Self-Adaptive Systems.
E-mail: SMH.Hasheminejad@Modares.ac.ir