An approach for the evolutionary discovery of software architectures

Aurora Ramírez, José Raúl Romero*, Sebastián Ventura

Department of Computer Science and Numerical Analysis, University of Córdoba, 14071 Córdoba Spain

Abstract

Software architectures constitute important analysis artefacts in software projects, as they reflect the main functional blocks of the software. They provide high-level analysis artefacts that are useful when architects need to analyse the structure of working systems. Normally, they do this process manually, supported by their prior experiences. Even so, the task can be very tedious when the actual design is unclear due to continuous uncontrolled modifications. Since the recent appearance of Search Based Software Engineering, multiple tasks in the area of Software Engineering have been formulated as complex search and optimisation problems, where Evolutionary Computation has found a new area of application. This paper explores the design of an evolutionary algorithm (EA) for the discovery of the underlying architecture of software systems. Important efforts have been directed towards the creation of a generic and human-oriented process. Hence, the selection of a comprehensible encoding, a fitness function inspired by accurate software design metrics, and a genetic operator simulating architectural transformations all represent important characteristics of the proposed approach. Finally, a complete parameter study and experimentation have been performed using real software systems, looking for a generic evolutionary approach to help software engineers towards their decision making process.

Keywords: Search based software engineering, software architecture discovery, evolutionary algorithms, ranking aggregation fitness

*Corresponding author. Tel.: +34 957 21 26 60
Email address: jrromero@uco.es. (José Raúl Romero)

1. Introduction

Throughout software development, software engineers need to make decisions about the most appropriate structures, platforms and styles of their designs. The automatic inference and evaluation of different design alternatives is a challenging application domain where computational intelligence techniques serve to provide support to software engineers, especially when limited information about the system being developed is still available.

In this context, architectural analysis constitutes an important phase in software projects, as it provides methods and techniques for handling the specification and design of software in the earlier stages [9]. It is considered a human-centered decision process with a great impact on the quality and reusability of the end product. During high level analysis, component identification allows the discovery of system blocks, their functionalities and interactions. For this reason, it is a good practice when dealing with complex systems [35], resulting in more comprehensible software and making its development and maintenance simpler and more affordable.

Frequently, software engineers need to tackle architectural analysis from a working system in order to migrate it or extend its functionality [10]. This could be a difficult task when the underlying system conception has been perverted due to requirements changes. A more dramatic situation occurs when reverse engineering techniques from source code are the only way to extract system information, leading to inappropriate abstractness because of missing documentation. In these cases, engineers must expend their time and effort, with their own experience as their only guarantee, in the manual discovery of these functional blocks.

Architectural optimisation methods in the field of Software Engineering (SE) have often proposed guidelines and recommendations to modellers for the identification and improvement of software architectures [5, 6]. Hence, semi-automatic tools and intelligent systems might be an efficient solution to support the engineering work in order to obtain quality models.

More specifically, the discovery of the architecture of a software specification can also be formulated as the search of the most appropriate distribution of available software artefacts in more abstract units of construction. Traditionally, proposed approaches are based on the refactoring of source code [21, 34], implying that architectural blocks are recovered at the end of the development process without regarding analysis decisions. Besides, it is frequent that source code is evolved without an exhaustive control from the

analysis perspective, and it is likely not to be representative of the original conception of the system. Instead, the discovery process can be carried out using earlier available information, like the detailed analysis models in the form of class diagrams. These models offer an intermediate view of the software, between the abstractness of the architecture specification and the specificity of the code.

Recently, the combination of metaheuristic approaches and Software Engineering as problem domain, denominated Search Based Software Engineering (SBSE), has undergone a huge growth [17]. Since the appearance of SBSE, Evolutionary Computation (EC) has emerged as the most applied metaheuristic [16], demonstrating that it constitutes an interesting and complementary way to help software engineers in the improvement of their object-oriented class designs [33] or user interfaces [36]. In this paper, EC is explored as a search technique to extract the underlying software architecture of a system. It constitutes a novelty in SBSE, where architectural discovery has been viewed as a re-engineering task from source code, which is more oriented towards maintenance and refactoring purposes. The main research questions posed in this work are the following:
RQ1: Can single-objective evolutionary algorithms (EA) help the software engineer to identify an initial candidate architecture of a system at a high level of abstraction? Such an approach should be heavily oriented towards the expert domain, looking for the interoperability with software engineering standards and tools, as well as for the comprehension of the elements involved within the evolutionary model.
RQ2: How does the configuration of the algorithm influence both the evolutionary performance and the quality of the returned solution? In order to answer to this question, an in-depth parameter study is required, aiming to provide useful guidelines on this regard to the software architect.

In the proposed evolutionary approach, class diagrams constitute the source artefacts used to abstract the software architecture, which is encoded using a flexible tree structure. Design alternatives are explored by a specific genetic operator applying domain knowledge. Concepts like cohesion and coupling guide the search, defining a ranking-based fitness function.

The rest of the paper is structured as follows. Section 2 introduces some background in SBSE and architectural modelling. Section 3 details the problem description, whereas the evolutionary model is described in Section 4. Next, experimentation is presented in Section 5, including a detailed parameter study. An illustrative example of the approach is explained in Section 6,

and results are discussed in Section 7. Finally, concluding remarks are outlined in Section 8.

2. Background

This section presents the most relevant subjects and background related to our work. More specifically, it introduces Evolutionary Computation as a technique to solve Software Engineering tasks, as well as the main terminology related to architectural analysis. Finally, previous works on software architecture optimisation in SBSE are presented.

2.1. Evolutionary Computation in Software Engineering

Evolutionary Computation [7] is one of the first population-based and bio-inspired metaheuristics for the resolution of optimisation problems. For this reason, EC has been applied for many years now to a variety of topics and considerable efforts have been applied in order to propose new techniques and operators [38] to solve complex applications.

Applying metaheuristics like EC to any domain requires that the scenario to be solved must be reformulated as an optimisation problem. Software Engineering is not an exception [8]. Since the appearance of SBSE, considerable efforts have been devoted to this field. Although the first and most studied area has been the automation of test case generation [12], other tasks related to the rest of phases in the software development, from requirements specification [39] to software verification [27], have already been studied. The advances in the field demonstrate that the application of EC to software enhancement is not only focused in the generation of automated programs, other activities classically performed by humans present new challenges.

Since SE is mainly a human-centered activity, the automation of the expert's reasoning presents a great challenge, especially in the analysis and design phases [29]. Design tasks considered in SBSE encompass problems like the conception of both object-oriented [33] and service-oriented [30] architectures, software module clustering [28] or software refactoring [20]. These activities are characterised by the need of constructing some type of software model from requirements. Both module clustering [28] and software refactoring [20] are more relevant to software maintenance, since existing software artefacts must be scrutinised in order to provide design alternatives.

In [33], an evolutionary algorithm is combined with software agents to extract the most fitting UML class diagram for a given set of methods and

attributes from use cases. This type of software requirement information is also taken as an input of the evolutionary approach proposed in [18], where logical groups of use cases are identified and put together into component packages. In this case, the authors presented a generic framework inspired by clustering techniques. In [30], genetic programming is used to deal with service composition in order to obtain the best orchestration of web services.

Frequently, popular evolutionary schemes and generic operators are selected and adapted when needed, since EC research history provides sufficient candidate elements [22]. Quite the opposite occurs when addressing more complex problems and specific implementations are required [11]. The problem description determines the need for either generic or specific elements. In this sense, the genetic algorithms conceived in [18, 28] handle integer encodings for the allocation of software artefacts, whereas those designed in [20, 30] require tree structures and special operators for the application of genetic programming approaches. Additionally, an object-oriented encoding to represent the set of classes, methods and attributes is proposed in [33].

2.2. Component-based software architectures

According to the ISO Std. 42010 [19], the architecture of a software system conceives "the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution". These high-level abstraction models are very appropriate for guiding and controlling its subsequent development, since it constitutes a bridge between software requirements and the implementation with specific programming languages [13]. Ideally, these models should be considered during the entire process, evolving as the software does. However, they frequently tend to be shifted during the implementation and the maintenance phases, since architectures do not have a direct representation in code artefacts.

A component-based architecture depicts a special type of architectural model, founded on the idea of constructing the software by means of independent artefacts that aim to promote the reuse of functionality. A commonly accepted definition of *component* is given by Szyperski [35]: "a component is a unit of composition with contractually specified interfaces and explicit context dependencies only". It is also mentioned that a component "can be deployed independently and is subject to third party composition". Internal objects implement its functionality, although they remain hidden and inaccessible beyond the component limits. Relations between components should

be defined by means of provided and/or required interfaces. A *provided interface* is defined as "a set of named operations that can be invoked by clients", separating the specification of the functionality from its real implementation. On the other hand, a *required interface* specifies the services invoked by the component, and provided by others. Finally, *connectors* link two or more interaction points between interfaces.

The importance of component-based architectures lies in its capability to represent a variety of software systems by means of abstract units of construction, without the consideration of the final specific technology or context in which the software will be deployed. Thus, components could be implemented as packages, modules or even single classes in object-oriented systems, as well as a set of services provided in the cloud or distributed objects in open and large distributed systems.

Software architects require, just like metaheuristics do, mechanisms to evaluate the adequacy of their models. The most frequently used measures are related to cohesion and coupling [14]. Cohesion refers to the degree to which the elements comprising the component are necessary and sufficient to carry out a single, well-defined function. Coupling is related to the interdependence between components, probably caused by references to other modules and data flows. Good component-based designs, i.e. specifying cohesive components with low dependencies, provide highly scalable software systems with better encapsulation and modularity. There exist other diverse metrics that help engineers in either the measurement of non-functional properties of its component-based designs, like integration [1] or usability [4], or the quantification of the symmetrical elegance of the software design [32].

2.3. Search-based architectural design

Current SBSE proposals in the context of software architectures can be viewed from diverse perspectives [2], since there are different kinds of factors to be considered in an architectural specification. The architecture definition (i.e. modelling languages, available constructs or the algorithm representation); the quality attributes to be measured, either functional or non-functional; the presence of prior components or design patterns, as well as the design purpose (recovery, refactoring, software implementation and hardware deployment) present a variety of application problems.

The conception of a low-level architecture from use cases as a software architecture synthesis problem is proposed in [31]. The authors present a genetic algorithm that takes as input an initial grouping of classes obtained

by a simulated annealing algorithm from a graph of functional dependencies, previously extracted from use cases. The resulting architectural specification is composed by classes and interfaces, according to the design patterns that best fit the requirements.

Next, the process proposed in [21] combines a clustering approach and a genetic algorithm for the recovery of component-based architectures from source code. It is a re-engineering model where the genetic search is performed over an initial architecture obtained after the study of relationships among source code elements (classes, interfaces, packages) from a functional dependency graph. It requires a complex mapping process, since it considers a fixed linear encoding representing the distribution of each class into a component, and needs a transformation mechanism to properly present the solution to the user. The proposed evolutionary model focuses on the architecture reconstruction from source code, its goal being closest to software maintenance. The software code constitutes a powerful source of detailed information about how the system works, but it is not clear that high-level characteristics can be directly extracted from it, since human decisions and abstract information are commonly faded away throughout the software construction process.

Assembling COTS (*Components Off-The-Shelf*) is another example of architecture construction. These COTS already implement specific functionalities whose combination is optimised in order to conform with the overall system functional requirements. In [24], well-known multi-objective genetic algorithms are used to generate design alternatives from an initial component-based architectural model. Besides, the proposal requires precise annotations of the evaluable metrics on each component from the expert, i.e. cost or performance, being components considered as black-box artefacts. Along the same lines, the authors explore in [3] the selection of the optimal subset of pre-existing components, which determine the next release of a system, using simulated annealing and greedy algorithms.

Finally, the framework presented in [23] addresses the issue concerned to architectural deployment, where software components within a distributed system must be allocated in hardware nodes in order to properly satisfy the non-functional requirements given, such as cost, latency and memory consumption. Here, the software specification already exists, so the evolutionary search is focused on exploring several different platforms where the deployed software would be executed.

3. Problem description

The identification of the architectural models is considered during the early stages of software conception, when software modellers still want to modify their current software structure as requirements change or they are requested to check the correctness of the resulting design.

When source code artefacts are not yet available, architects require other sources of information in order to discover the intended architecture. Initial class diagrams, usually the most used representations in the analysis phase, constitute an interesting starting point for architecture discovery. These diagrams offer more specific analysis information than source code, and they use modelling languages like UML 2 [26] instead of programming languages.

Therefore, the originally intended elements that conform a component-based architecture (components, interfaces and connectors) will be identified from these analysis models, resulting in an architecture represented with a UML 2 component diagram. At this point, the semi-automatic discovery of components including its internal structure, candidate interfaces and connectors can be constrained by the following assumptions:

1. A component is defined as a cohesive group of classes, meaning that they work together to satisfy the expected behaviour of the component. Thus, classes within the diagram will be organised searching the best abstraction of the different functionalities that can be identified in the software.

   A very important constraint to consider is that *any class in the input diagram must be contained in one and only one component in the resulting architecture*. Additionally, *any operation or transformation of the architecture must ensure that no empty components are returned*.

2. A directed relationship between classes in the analysis model belonging to different components represents a candidate interface. Although groups of related classes should be allocated in the same component, some interactions could remain between classes belonging to other components, representing operational flows among them. Then, these relationships, required to perform the overall functionality of the system, will be abstracted as interactions between components, i.e. defining its interfaces.

   It can be observed that not all the relationships can constitute a candidate interface. For instance, generalizations represent data abstractions, so they do not imply a flow of operational information. The navigability

of the relationship is also important because, if it is not explicitly represented, it would mean that information is exchanged in both directions, the corresponding classes being highly dependent. If the navigability is presented for only one direction, the flow represents a provided or required candidate service.

Focusing on the interactions between components, *isolated components are not appropriated as they do not provide any service to others*. Secondly, *mutually dependent components are not permitted* from the architectural perspective. This latter circumstance occurs when a component requires and provides services from another component.

3. Connectors can be described as the linkage between a pair of required / provided interfaces interconnecting different components. They will be identified after the discovery of the interfaces created between components.

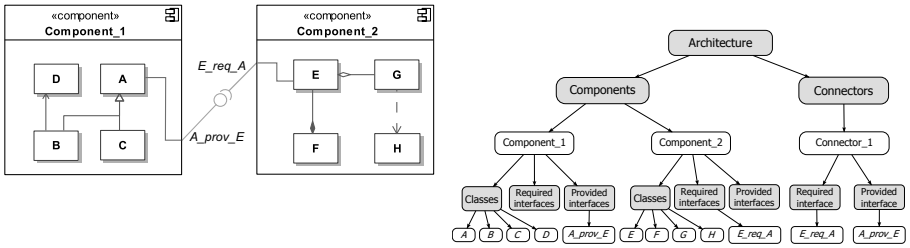
4. Proposed model for architecture discovery

In this section, the different elements of the proposed evolutionary model are presented, including the encoding chosen, the fitness function and the genetic operator. All these elements are conceived with the aim of creating a comprensible EA as posed by *RQ1*. Finally, the description of the evolutionary algorithm is detailed.

4.1. Encoding of solutions

Selecting the most appropriate problem encoding is a key step in any search algorithm. Usually, a trade-off between the performance and comprehensibility must be achieved, especially when genetic algorithms are aimed at supporting non expert users in metaheuristics. Although the linear encodings proposed in Sections 2.1 and 2.3 seem to be efficient representations, difficult design problems still require its adaptation by means of superstructures or groups of consecutive genes to represent more complex features. In these cases, efficiency decreases due to the use of operators which are too specific or the need for corrective procedures after the application of generic operators.

Human interpretation is usually hampered by complex genotype / phenotype mappings. Therefore, an easier mapping process for software design problems might be beneficial. Tree structures seem to be an interesting option, as they have been used successfully in both computational and human



(a) Phenotype (b) Genotype

Fig. 1. The phenotype/genotype mapping process.

domains. Moreover, these types of representation are also familiar to software architects, because they are common structures in modelling tools, and they allow a flexible management of solutions with different sizes, e.g. architectures with a variable number of components and connectors.

Components, interfaces, connectors and inner elements clearly present a hierarchical composition. Classes and their relationships may constitute a component, whose complete specification requires the definition of its provided and required interfaces. Connectors can be split into the interfaces they link. Then, mapping a component diagram into a tree structure is feasible as shown in Fig. 1, where shading nodes constitute the solution frame, comprised by those mandatory artefacts appearing in any architectural model. The rest of nodes represent the elements that can be different from one solution to another, i.e. a number of component and connectors as well as the distribution of classes and interfaces among them. More specifically, the root node, *Architecture*, represents the component diagram that is comprised of a set of components and connectors. Each component is defined by a node *Component* in terms of its internal classes and its interfaces. Similarly, each connector is described by the pair of required and provided interfaces that it links. Since they are compound elements, they are represented as non-terminal nodes. Finally, classes and interfaces constitute the terminal nodes.

4.2. Initial population

From the problem description (see Section 3), it can be noted that the search space is constituted by all possible combinations of class distribution

among components, also identifying its interfaces and the connectors. These candidate groups of classes, and the way in which interfaces and connectors are deduced from them, must also guarantee that the correspondent architecture represents a valid solution.

Firstly, a random number of components is selected between a minimum and a maximum. Default values are set to a minimum of two and a maximum of n components, n being the number of classes in the input model. The higher limit guarantees that no empty components will be generated. Then, each class is assigned to one component, assuring that each component has at least one class. After this initial assignment, the rest of the constraints detailed in Section 3 are omitted, allowing a faster initialisation process. As will be explained later, the main idea is that these invalid individuals will be progressively removed along the generations.

4.3. Ranking fitness function

As mentioned in Section 2.2, diverse functional or non-functional properties can be considered depending on the underlying goal of the architectural optimisation. In this case, the search process is mainly focused on structural aspects, closely related to reusability, since it looks for the optimal identification of well-defined components, interfaces and connectors. Thus, the fitness function considers the strength and independence of the inner functionality of each component.

The fitness function is calculated as an aggregation of rankings. The use of rankings cancels out the need for standarisation between metrics, which could result in an artificial procedure when they are not defined in an appropriate range for a fair scalarisation and aggregation. Each ranking belongs to a specific metric related to desirable characteristics in the architectural design. Therefore, evaluating these design criteria requires the existence of quantifiable measures applicable to the problem domain.

Firstly, the *Intra-Modular Coupling Density* (ICD) [15] in Eq. 1 serves to determine a trade-off between cohesion and coupling. For each component i, ICD_i is calculated as the ratio between internal and external relations, which has to be maximised. CI_i^{in} is the number of interactions inside the component, i.e. the relationships between classes allocated in the same component. CI_i^{out} represents the number of relationships between component i and the others, i.e. the number of candidate interfaces of the component. Then, every value is properly weighted with the ratio of classes that participate in these relationships. Hence, if two components reach the same ratio

of interactions, the smaller component, i.e. the one with less inner classes, is preferable meaning that the density of interactions per class is higher. ICD_i varies in [0,1]. Finally, the ICD of the overall architecture (individual) is calculated as the average of every ICD_i.

$$\begin{aligned} ICD_i &= ((\#classes_{total} - \#classes_i)/\#classes_{total}) \cdot \left(CI_i^{in}/(CI_i^{in} + CI_i^{out})\right) \\ ICD &= \sum_{i=1}^{n} ICD_i/n \end{aligned} \tag{1}$$

The second metric, named *External Relations Penalty* (ERP), applies a penalty if some relations are not specified by means of interfaces. The optimum value is 0, meaning that no relationship outside the identification of a candidate interface is presented between classes allocated in different components. The minimisation of these dependences between components is an important characteristic to be considered, as it reflects that only interactions among interfaces are adequate in good designs. These external relationships could be generalizations (ge) or not directed relationships like associations (as), aggregations (ag) and compositions (co), as they do not allow the abstraction of candidate interfaces. Dependences are not included because they always have a direction. Since the software architect might be interested in setting certain design preferences by demoting some relationships to others, a weight (w_x) can be assigned to the number of occurrences of each type of relationship (n_x). As an example, the modeller may want to avoid dividing into different components a parent class and its subclasses, i.e. sharing data structures, which could be more harmful to the overall cohesion than just splitting a single association between them, usually involving an operational flow. This would imply assigning a higher weight value to generalizations. Therefore, ERP is calculated using the expression in Eq. 2, where i and j represent each pair of components in the architectural solution.

$$ERP = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (w_{as} \cdot n_{as_{ij}} + w_{ag} \cdot n_{ag_{ij}} + w_{co} \cdot n_{co_{ij}} + w_{ge} \cdot n_{ge_{ij}}) \tag{2}$$

Finally, the *Groups/Components Ratio* (GCR) metric, presented in Eq. 3, is inspired by the *Component Packing Density* (CPD) metric defined in [25]. CPD calculates the ratio between the number of constituents, e.g. operations, classes or modules, and the number of components in the overall

architecture. Here, the constituents are groups of interdependent classes (*cgroups*). In a graph visualisation of the model, where classes are the nodes and its relationships, the edges, each *cgroup* is a *connected component* of this graph. Since software architects prefer a set of components with a well-defined functionality, the optimal value of GCR is equal to its minimum, 1, meaning that each component is comprised by an unique group of strongly interrelated classes.

$$GCR = \#cgroups/\#components \tag{3}$$

Once the three design metrics have been defined, the fitness function can be calculated as a ranking aggregation, where the best values are the lowest and, consequently, the overall fitness should be minimised. A ranking method is applied over the population and independently for each metric, resulting in three ranking values that are added, as can be seen in Eq. 4, where r returns the ranking position of a specific individual.

$$fitness_{ind} = \begin{cases} r(ICD_{ind}) + r(ERP_{ind}) + r(GCR_{ind}) & \text{if } ind \text{ is } valid \\ \#individuals \cdot \#metrics + 1 & \text{if } ind \text{ is } invalid \end{cases} \tag{4}$$

Special attention is given to invalid solutions. In such a case, a high value is assigned to the individual, i.e. a fitness value even greater than the value computed for the worst valid individual. If a valid individual would have reached the worst values in all the metrics, its ranking for each metric would be equal to the number of individuals in the population, and the aggregate value would be equal to the product of the number of metrics composing the ranking and the population size. Thus, an invalid solution always has a greater fitness than any valid individual just by adding 1 to this value.

4.4. Genetic operator

Genetic operators allow the creation of new solutions from others. Here, a mutation operator is considered for exploring design alternatives. Due to the characteristics of the problem, the execution of other kinds of operators does not seem to be applicable, as they would probably cause the replication of classes after the combination of components from different individuals.

Five mutation procedures are proposed in order to provide a variety of new solutions, simulating those architectural transformations that software architects could manually apply during the discovery process. Domain knowledge

is properly used in most cases, being an important success factor, as some of them have a great impact in the structure of the resulting architecture. Next, the description of each procedure is detailed.

Add a component. A new component is added to the architecture. Since empty components are not valid, one or more classes are selected from others to be inserted into the new one. The underlying heuristic considers the number of groups of classes inside the rest of components as a decision factor. More precisely, components built with more unconnected groups (which probably do not present a well defined functionality) are considered better candidates to provide classes than those with a unique group of classes.

At this point, the heuristic procedure uses the expression in Eq. 5 as a probability threshold of selection of each component i to act as contributor. As can be seen, this formula calculates a probabilistic value for each component i as the ratio between its number of groups of classes ($\#cgroups_i$) and the maximum number of groups ($max_{cgroups}$) corresponding to some component j of the architecture. Thus, the higher the number of groups inside the component i, the greater the probability of selecting some of its groups.

$$\begin{aligned} Prob(i_{contributor}) &= \#cgroups_i / max_{cgroups} \\ max_{cgroups} &= max(\#cgroups_j)\ j \in [1, n] \end{aligned} \tag{5}$$

The complete heuristic procedure is shown in Algorithm 1. Firstly, variables are initialised and the probability of "acting as contributor" is calculated for each component. If a random generated value surpasses the probability threshold, the groups of classes inside the component are obtained (lines 4 - 5). If the component comprises more than one group, their size (i.e. the number of classes composing it) is calculated and the smallest groups are searched (lines 6 - 13). Notice that small-sized groups are preferable because the new component could also receive groups of classes from others. Thus, a group of classes between those candidates, i.e. the smallest groups, is randomly selected, and its classes are removed and inserted into the new component, while the rest of the component is copied in the offspring (lines 14 - 16). The process is repeated for each available component in the parent. If no component in the parent meets the requirements (all of them presents a unique group of classes), or the randomness of the result cannot be guaranteed (only one candidate exists, so the descendant would be always the

same), the new component is generated completely at random, extracting classes from all existing components (lines 18 - 30).

At the end, the new component is added to the offspring (line 31) and the interfaces and connectors have to be arranged (line 32), considering that the new distribution of classes may produce changes in the interactions among components. More precisely, interfaces are moved from the contributors to the new component if the classes that would implement these interfaces have been displaced. At this point, two circumstances can occur: (a) an interface remains in the new component because the interaction target continues to exist within the original component, or (b) both interfaces must be removed, since the classes specifying them have been allocated in the new component, and the interaction only happens internally. Similarly, the movement or loss of interfaces may also affect the number of connectors.

Fig. 2b corresponds to the resultant individual after the application of this mutation procedure over the individual shown in Fig. 2a. As can be seen, the movement of classes F from $Comp\_2$ and B from $Comp\_1$ implies that interface B_req_D is also removed from $Comp\_2$ and allocated in the new one ($Comp\_3$). After that, $Comp\_1$ interacts with $Comp\_3$, providing it some services, instead of with $Comp\_2$.

Split a component. One component is divided into two new components. The heuristic firstly tries to randomly select a component among those with more than one group of classes. In Algorithm 2, candidate components are identified (lines 3 - 7). If more than one candidate exists, one of them is randomly selected and each of its inner groups is randomly allocated in one of the two new components with equal probability (lines 8 - 17). If all components present a unique group of classes, one component in the parent is chosen and its classes are randomly distributed (lines 18 - 26). Then, all components in the parent except the component to be split are copied, and the two new components are also added (lines 27 - 29). Finally, interfaces and connectors are identified again, as the creation of new components can produce the appearance of new interactions (line 30).

Remove a component. One component will be removed and its inner classes, randomly distributed among the remaining components. An aim of this operator is to improve the solution by reducing the ERP metric. As can be seen in Algorithm 3, the number of external relations outside the bounds of each component is obtained and those with the highest value are selected

Algorithm 1 *Add a component*

```
Require: parent
Ensure: offspring
 1: offspring ← ∅
 2: for all component in parent do
 3:   candidates ← ∅
 4:   if (random(0,1)> Prob(component)) then
 5:     allGroups ← getGroups(component)
 6:     if (size(groupsOf Classes) > 1) then
 7:       for all groupOf Classes in allGroups do
 8:         if (size(groupOf Classes) = min) then
 9:           candidates ← groupOf Classes
10:         end if
11:       end for
12:     end if
13:   end if
14:   newComp ← randomGroup(candidates)
15:   offspring ← component − candidates
16:   candidates ← ∅
17: end for
18: if (newComp = ∅) then
19:   offspring ← ∅
20:   for all component in parent do
21:     for all class in component do
22:       if (random(0,1) > 0.5) then
23:         candidates ← class
24:       end if
25:     end for
26:     offspring ← component − candidates
27:     newComp ← candidates
28:     candidates ← ∅
29:   end for
30: end if
31: offspring ← newComp
32: setInterfacesAndConnectors(offspring)
33: return offspring
```

Algorithm 2 *Split a component*

```
Require: parent
Ensure: offspring
 1: offspring ← ∅
 2: candidates ← ∅
 3: for all component in parent do
 4:   if (numberOfGroups(component) > 1) then
 5:     candidates ← component
 6:   end if
 7: end for
 8: if (size(candidates) > 0) then
 9:   compToSplit ← randomComponent(candidates)
10:   for all groupOfClasses in compToSplit do
11:     if (random(0,1) > 0.5) then
12:       component1 ← groupOfClasses
13:     else
14:       component2 ← groupOfClasses
15:     end if
16:   end for
17: else
18:   compToSplit ← randomComponent(parent)
19:   for all class in compToSplit do
20:     if (random(0,1) > 0.5) then
21:       component1 ← class
22:     else
23:       component2 ← class
24:     end if
25:   end for
26: end if
27: offspring ← parent − compToSplit
28: offspring ← component1
29: offspring ← component2
30: setInterfacesAndConnectors(offspring)
31: return offspring
```

(lines 4 - 8). Then, a random component is chosen among them and the rest of components are copied in the offspring (lines 9 - 10). Next, the inner classes of the removed component are randomly distributed in the remaining components of the offspring (lines 11 - 13).

Finally, interfaces and connectors are checked in the offspring (line 14). In this case, interfaces from the removed component are either bound to other components when they have received the corresponding classes, i.e. those specifying the required or provided service, or removed, if the target component was the owner of the other interaction point.

Merge two components. The elements of two previously selected components are all put together into a new component. The proposed procedure, detailed in Algorithm 4, looks for the reduction of the ERP metric. As can be seen, one of the two components taking part in the mutation is the component having the highest number of external relations (lines 4 - 9). When some components present the highest values, two of them are selected (lines 10 - 11). If not, the other component is randomly selected between the rest of

Algorithm 3 *Remove a component*

Require: *parent*
Ensure: *offspring*
1: *offspring* ← ∅
2: *candidates* ← ∅
3: *maxRel* ← maxNumExtRel(*parent*)
4: **for all** *component* in *parent* **do**
5: **if** (numExtRel(*component*)= *maxRel*) **then**
6: *candidates* ← *component*
7: **end if**
8: **end for**
9: *compToRemove* ← randomComponent(*candidates*)
10: *offspring* ← *parent* − *compToRemove*
11: **for all** *class* in *compToRemove* **do**
12: randomComponent(*offspring*) ← *class*
13: **end for**
14: setInterfacesAndConnectors(*offspring*)
15: **return** *offspring*

Algorithm 4 *Merge two components*

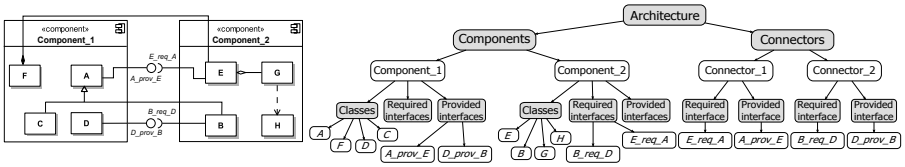
Require: *parent*
Ensure: *offspring*
1: *offspring* ← ∅
2: *candidates* ← ∅
3: *maxRel* ← maxNumExtRel(*parent*)
4: **for all** *component* in *parent* **do**
5: **if** (numExtRel(*component*)=*maxRel*) **then**
6: *candidates* ← *component*
7: **end if**
8: **end for**
9: *component*1 ← randomComponent(*candidates*)
10: **if** (size(*candidates*) > 1) **then**
11: *component*2 ← randomComponent(*candidates*)
12: **else**
13: *component*2 ← randomComponent(*parent*)
14: **end if**
15: *offspring* ← *parent* − (*component*1 ∪ *component*2)
16: *offspring* ← *component*1 ∪ *component*2
17: setInterfacesAndConnectors(*offspring*)
18: **return** *offspring*

components in the parent (lines 12 - 13). Next, components not selected in the parent are copied in the offspring, as well as the union of the two selected components (lines 15 - 16). Finally, interfaces and connectors must be compacted due to the merge of the two components, so the previous interactions between them are discarded (line 17).

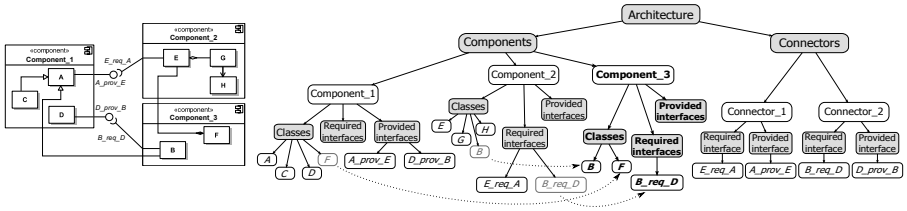
Move a class. A simple movement of a class from one component to another is performed. As it is the least destructive procedure in terms of structural modifications of the original solution, both the class and the source and target components are always randomly selected (see lines 2 - 4 of Algorithm 5). The selected class is removed from the origin component and added to the destination component (lines 5 - 6). Then, the original and modified components are copied to the offspring (lines 7 - 9). Since the class repositioning could also imply the creation of new interfaces in the target component and its elimination from the source, or vice versa, interfaces and connectors are revised (line 10).

An example of the application of this procedure is shown in Fig. 2c. In the original individual (see Fig. 2a), class B is chosen and moved from $Comp\_2$ to $Comp\_1$. This operation also affects the interaction between both components, since interfaces D_prov_B and B_req_D disappear because the classes B and D belong now to the same component.

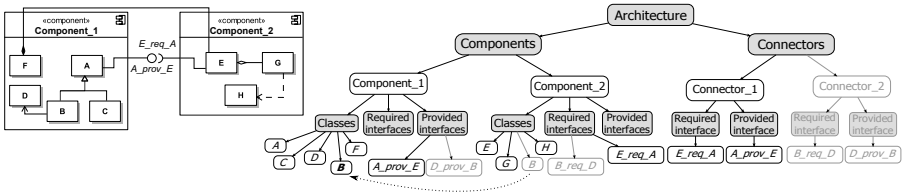
After the explanation of the different mutation procedures proposed, the mutator itself can be described. As detailed in Algorithm 6, a probabilistic roulette is built for each parent comprising only those mutation procedures that could be applied (lines 4 - 8). For example, a component can not be



(a) Initial individual



(b) Add a component mutation procedure



(c) Move a class mutation procedure

Fig. 2. Examples of mutation procedures.

removed if the individual already comprises the minimum number of components. Once the roulette is completed, a mutation procedure can be randomly selected according to its configured weight (line 9). If the resulting individual does not satisfy all the architectural constraints, a new mutation is performed until a valid individual is obtained or a maximum number of attempts is reached (lines 10 - 19).

If all attempts fail and no valid solution is found, the mutated individual could survive (lines 20 - 22) depending on the stage of the evolution process. A probabilistic method is proposed in order to determine whether this invalid individual will be considered as a candidate to be part of the new population, i.e. an offspring in the survival competition. A dynamic threshold, $Threshold_{invalid}$, which decreases with the elapse of generations ($gener$), is calculated in Eq. 6. Notice that, at the beginning of the algorithm, invalid individuals are permitted. Nevertheless, less invalid solutions generated by the

Algorithm 5 *Move a class*

Require: *parent*
Ensure: *offspring*
1: *offspring* ← ∅
2: *origin* ← randomComponent(*parent*)
3: *destination* ← randomComponent(*parent*)
4: *class* ← randomClass(*origin*)
5: removeClass(*class*, *origin*)
6: addClass(*class*, *destination*)
7: *offspring* ← *parent* − (*origin* ∪ *destination*)
8: *offspring* ← *origin*
9: *offspring* ← *destination*
10: *setInterfacesAndConnectors(offspring)*
11: **return** *offspring*

Algorithm 6 *Mutation operator*

Require: *parent*, *weights*
Ensure: *offspring*
1: *offspring* ← ∅
2: *roulette* ← ∅
3: *selectedMutator* ← ∅
4: **for all** *mutator* in *mutators* **do**
5: **if** (isApplicable(*mutator*, *parent*)) **then**
6: *roulette* ← mutator
7: **end if**
8: **end for**
9: *selectedMutator* ← rouletteSelection(*roulette*,*weights*)
10: *attempts* ← 0
11: *invalid* ← *TRUE*
12: **while** (*invalid* = *TRUE* and *attempts* < 10) **do**
13: *offspring* ← mutate(*selectedMutator*, *parent*)
14: **if** (isInvalid(*offspring*)) **then**
15: *attempts*++
16: **else**
17: *invalid* ← *FALSE*
18: **end if**
19: **end while**
20: **if** (isInvalid(*offspring*) **and**
random(0,1) < $Threshold_{invalid}$) **then**
21: *offspring* ← *parent*
22: **end if**
23: **return** *offspring*

mutator will survive due to the dynamic decrease of the threshold during the evolutionary process. Then, the replacement strategy determines whether the invalid individual will finally be part of the next generation.

$$Threshold_{invalid}(gener) = (\#generations - gener)/\#generations \quad (6)$$

4.5. Algorithm

The proposed algorithm (see Algorithm 7) follows the classical generational scheme. Firstly, some preprocessing is required (lines 1 - 3) in order to extract classes and its relationships from the analysis model ($classDiagram$). Then, candidate interfaces are identified using the information comprised by these relationships. Connectors are not explicitly obtained at this step, as they depend on the association of two specific candidate interfaces, and this process is performed during the creation of individuals. Next, these elements are used in combination with the number of individuals ($nInds$) and the minimum and maximum in the number of components ($minComp$ and $maxComp$ respectively), to initialise the population (line 4). Then, individuals are evaluated (line 5) and the iterative process begins. In each generation, parents are selected (line 8) and mutated (line 9) according to the mutation weights ($weights$). Candidate individuals must be evaluated next (line 10), so metrics are computed over them and the ranking fitness function is calculated. Note that this evaluation requires both the offsprings and the actual population in

Algorithm 7 *Proposed evolutionary algorithm*

Require: *classDiagram*, *nInds*, *maxGen*, *weights*, *minComp*, *maxComp*
Ensure: *candidateArchitecture*
1: *classes* ← extractClasses(*classDiagram*)
2: *relationships* ← extractRelationships(*classDiagram*)
3: *interfaces* ← identifyInterfaces(*relationships*)
4: *population* ← create(*nInds*, *minComp*, *maxComp*, *classes*, *interfaces*)
5: evaluate(*population*, *relationships*)
6: *generation* ← 0
7: **while** *generation* <= *maxGen* **do**
8: *parents* ← select(*population*)
9: *offspring* ← mutate(*parents*, *weights*)
10: evaluate(*population* ∪ *offspring*, *relationships*)
11: *population* ← replace(*population* ∪ *offspring*)
12: *generation*++
13: **end while**
14: *candidateArchitecture* ← best(*population*)
15: **return** *candidateArchitecture*

order to assign rankings in a proper way. Finally, the replacement strategy (line 11) chooses those individuals that will survive, assigning them to the next population. When the maximum number of generations ($maxGen$) is reached, the evolution ends and the best individual in the current population is returned as the candidate architecture (lines 14 - 15).

5. Experimentation

The complete approach and all the experiments performed have been written in Java. Additionally, its functionalities have been supported by diverse publicly available Java libraries. SDMetrics Open Core[1] offers some utilities for parsing XMI files, the most commonly used XML format for model interchange, providing interoperability and serialisation across different modelling tools. Thus, the proposed approach provides support to directly collect information from analysis models created, in this case, with MagicDraw tool[2]. The Datapro4j library[3] has been used to preprocess and manage internal data structures. Finally, the evolutionary algorithm has been coded using the JCLEC framework [37].

The experiments were run on a HPC cluster of 8 compute nodes with Rocks cluster 6.1 x64 operating system. Each node comprises two Intel Xeon E5645 CPUs with 6 cores at 2.4 GHz and 24 GB DDR memory.

[1] `http://www.sdmetrics.com/OpenCore.html`
[2] `http://www.nomagic.com/`
[3] `http://www.uco.es/grupos/kdis/datapro4j`

Table 1: Problem instances and its internal properties

Problem	*#Classes*	*#Relationships*					*#Interfaces*
		As	De	Ag	Co	Ge	
Aqualush	58	69	6	0	0	20	74
Borg	167	44	109	36	38	90	300
Datapro4j	59	3	4	3	2	49	12
Java2HTML	53	20	66	15	0	15	170
JSapar	46	7	33	21	9	19	80
Marvin	32	5	11	22	5	8	28
NekoHTML	47	6	17	15	18	17	46

5.1. Problem instances

Seven system designs were used for experimentation, allowing a variety of complexity in both number of classes and number of candidate interfaces. Table 1 shows the characteristics of the problem instances considered. The interfaces column ($\#Interfaces$) represents the number of relationships among classes where its navigability is explicitly specified, i.e. the number of candidate interfaces. Note that the total number of relationships (navigable in one or both directions) is also included and categorized by the types of relations defined by UML 2: associations (As), dependences (De), aggregations (Ag), compositions (Co) and generalizations (Ge).

Focusing on the nature of the software models, it is worth mentioning that six of them belong to real working systems, whereas the first one, *Aqualush*[4] is a benchmark used for educational purposes. All of them apart from *Datapro4j* can be accessed from the Java Open Source Code Project Website[5].

5.2. Parameter study

Due to the complexity of the problem, the performance of an accurate parameter study is recommended in order to analyse their suitability and influence. Firstly, different selection and replacement methods are combined and proved in order to check its influence in two important factors: the selection pressure and the capability to remove invalid solutions. Additionally,

[4] `http://www.ifi.uzh.ch/rerg/research/aqualush.html`
[5] `http://java-source.net/`

the behaviours shown by setting different weights associated to the roulette of mutation procedures (see Section 4.4) permit analysing their influence on the quality and type of returned solutions. Finally, other parameters, like the number of generations or the population size, need to be considered, since they represent key aspects in the evolutionary performance.

Regarding *RQ2*, the aim here is to obtain the most fitting setup for the proposed algorithm, whilst also providing some guidelines on the parameters that can be helpful to the software architect, who is likely not to be an expert in optimisation techniques. In this sense, we want to stress the ability of the algorithm to serve as a generic framework for architecture optimisation, where different types of solutions can be simultaneously evolved.

5.2.1. Selection and replacement strategies

Selection and replacement procedures constitute important factors in the algorithm design. Selection determines the way in which individuals are chosen to be mutated, whereas the replacement defines the type of survival competition between them. The selection methods probed are the following:

- *Deterministic selector* (DS): Each individual in the population is selected to act as a parent.
- *Tournament selector* (TS): A binary tournament is performed as often as the number of individuals in the population, in order to generate the same number of descendants than the previous method.
- *Roulette selector* (RS): A roulette is applied to select the parents. In the same way, the process is applied until the number of parents reaches the population size.

Focusing on the replacement strategies, some special constraints are considered in the replacement methods that are given below. Firstly, the best individual in the current population will survive. Secondly, when some type of competition must be established between a current invalid solution and a generated invalid descendant, both having the same maximum fitness value, the descendant is preferred, promoting the evolution of the population. The strategies considered in the preliminary study are the following:

- *Best individuals* (BR): The best n individuals from parents and descendants are selected to conform the new population, n being the population size.

- *Parent/descendant competition* (CR): A competition between each parent and its descendant is performed, and only the best survives.

- *Elitism (1) and descendants* (EL1R): After saving the best individual found in the current population, the rest of the population is filled with the $n-1$ best descendants.

- *Elitism (10%) and descendants* (EL10R): Similar to EL1R, but keeping a major percentage of individuals from the current population.

- *Binary Tournament* (TR): all individuals are participants of the tournament, and the n best individuals are selected for the next population.

To perform an accurate experimentation and setup, each selection method has been combined with the aforementioned replacement strategies, resulting in 15 different algorithm variants. Then, 30 executions for each algorithm version have been performed with different random seeds. The rest of the parameters are fixed to default values. All mutation procedures have the same probability to be executed, 0.2 being the corresponding weight for each one. The default minimum and maximum number of components is set to 2 and to the number of classes within the original analysis model, respectively. Here, the maximum limit has been fixed to 8, providing a wide enough range of types of solutions for the considered problem instances. The weights for the different types of UML relationships, used in the ERP metric, are internally fixed using the following configuration: $w_{as} = 2$, $w_{ag} = 3$, $w_{co} = 3$, $w_{ge} = 5$. Finally, 100 individuals and 100 generations complete the parameter configuration at this point of the study.

The first analysis of the obtained results has consisted in the evaluation of two important criteria: (a) the ability of removing invalid solutions and (b) an appropriate convergence of the population. Some variants have been discarded, as they do not achieve a final population of valid individuals, owing to an inappropriate selection pressure. This situation frequently occurs with the replacement based on TR, especially with the most difficult problem instances. On the contrary, other versions suffer an excessive convergence, so they too are rejected. In this case, the main factor that promotes this fact is the replacement strategy, since BR and CR methods strongly encourage the survival of the best individuals and also lead the search towards the same type of architectural solutions.

After this preliminary study, those variants showing an appropriate behaviour in terms of the criteria aforementioned have been analysed considering the best solutions found for all the problem instances. The Friedman test was applied to statistically validate these results, where the null hypothesis, H_0, determines that all the remaining variants perform equally well. Next, the Holm post-hoc test was used when H_0 was rejected with a significance level of 95% ($\alpha = 0.05$).

Since the ranking value reached by the best individual is relative to the population to which it belongs, fitness values are not directly comparable among different executions. Therefore, the aggregate rankings of all the individuals returned by each variant and execution were recomputed. Adding the ranking obtained for every individual for a given algorithm, i.e. aggregating the results of the 30 executions per algorithm, the quality of the obtained solutions can be estimated in a proper way. The first column in Table 2 compiles the results after applying the Friedman test to these representative values.

As can be observed, *DS-EL10R* obtains the lowest ranking value. The corresponding value of the Iman and Davenport statistics, called z, is 0.4312, whereas the critical value, according to the F-Distribution with 5 and 30 degrees of freedom, the $p - value$, is 2.5336. Since $p - value > z$, H_0 cannot be rejected. At this point, there are no significant differences between them, and a further analysis is still required.

The preceding procedure has been repeated considering the values of each metric associated with the fitness formula separately, i.e. ICD, ERP and GCR. Table 2 shows the average ranking values after performing the Friedman test over the corresponding metrics in the best individuals found. The value of the statistics, according to the aforementioned Iman and Davenport procedure, and the conclusion about the null hypothesis are also included. Thus, significant differences exist for the ICD and ERP metrics (highlighted in bold typeface), z being greater than the $p - value$ (2.5336).

In order to reveal those differences regarding ICD and ERP, the Holm test has been performed as a post-hoc procedure. Table 3 and Table 4 detail the obtained results for ICD and ERP metric, respectively. As for the ICD, the algorithm *RS-EL10R* obtained the best average ranking in the Friedman test, so it is the control algorithm. At a significance level of $\alpha = 0.05$, Holm test rejects the hypothesis that the algorithm performs equally well than the control algorithm when $p - value < 0.0167$. Regarding the correspondent α/i column, *RS-EL10R* is statistically better than *DS-EL10R* and *TS-EL1R*.

Table 2: Friedman rankings for fitness and design metrics

Algorithm	Fitness	ICD	ERP	GCR
DS-EL10R	2.8571	4.2857	2.1429	3.0000
TS-CR	4.9286	3.4286	5.4289	5.2143
TS-EL1R	3.3571	5.4286	**1.8571**	3.1429
TS-EL10R	2.9286	3.1429	3.0000	3.2143
RS-EL1R	3.2857	3.0000	3.6429	2.7143
RS-EL10R	3.6429	**1.7143**	4.9286	3.7143
z	1.1757	4.9468	9.1546	1.8132
H_0	Accepted	Rejected	Rejected	Accepted

Table 3: Holm test results for ICD

i	*Algorithm*	z	p	α/i	H_0
5	TS-EL1R	3.7143	2.0378	0.0100	Rejected
4	DS-EL10R	2.5714	0.0101	0.0125	Rejected
3	TS-CR	1.7143	0.0865	0.0167	Accepted
2	TS-EL10R	1.4286	0.1531	0.0250	Accepted
1	RS-EL1R	1.2858	0.1985	0.0500	Accepted

Table 4: Holm test results for ERP

i	*Algorithm*	z	p	α/i	H_0
5	TS-CR	3.5714	3.5504	0.0100	Rejected
4	RS-10R	3.0714	0.0021	0.0125	Rejected
3	RS-EL1R	1.7857	0.0714	0.0167	Accepted
2	TS-EL10R	1.1429	0.2531	0.0250	Accepted
1	DS-EL10R	0.2858	0.7751	0.0500	Accepted

Related to ERP, the same procedure can be realised. In this case, *TS-EL1R* acts as the control algorithm, and significant differences can be appreciated when $p - value < 0.0167$. As can be seen, *TS-EL1R* is better, in terms of ERP, than *RS-10R* and *TS-CR*.

After this analysis, some conclusions can be drawn. As shown, ERP and ICD constitute two conflicting metrics, whilst GCD is easily optimised by

the considered algorithms, since there are no significant differences between them. The algorithm with the best average ranking when comparing by fitness, *DS-EL10R*, has not such a proper behaviour, since ICD is heavily harmed in favour of ERP and GCR. On the contrary, *TS-10R* comes up as an interesting option since it shows good performance in terms of its fitness, having the second better ranking. Moreover, it can be noted that, regarding the ICD and ERP metrics, there is no significant difference between this and the control algorithm. Usually, this variant is able to discard solutions where ERP is highly optimised. Consequently, it leads to the loss of quality in the structure of the final individuals. More specifically, notice that low values for ICD illustrate the fact that the obtained solutions comprise large components with only a few interaction paths through interfaces, similarly to the case of *DS-EL10R*. The rest of variants of the algorithm can only obtain better ICD values by getting a fairly poor performance on the ERP metric. *TS-EL10R* achieves an appropriate trade-off between the three considered metrics, obtaining lower values for ERP and GCR without requiring a considerable decrease of ICD. Furthermore, it performs well in terms of convergence along the overall evolution. Consequently, *TS-EL10R* is the variant selected for the proposed version of the algorithm.

5.2.2. Mutation weights

One important characteristic of the proposed model lies in the existence of a roulette of mutation procedures as a way to control the diversity of solutions in the evolutionary process. These weights have a direct effect on two aspects of the generated solutions: its quality, as each procedure acts guided by their heuristics, and the diversity of types of solution, since they apply changes in their structure.

Several experiments have been performed to analyse the aforementioned characteristics of the generated solutions. The proposed roulette for mutation method selection comprises, as detailed in Section 4.4, five different procedures, each having an specific weight. Considering increments of 0.1 units, each mutation procedure could have a weight in the range [0.1,0.6], 126 being the total number of possible configurations. For each of those combinations, 30 executions have been carried, keeping the default values in the remaining setup, over all the problem instances. Afterwards, the procedure detailed in Section 5.2.1 is performed once again to reassign the ranking values of the best individuals.

Due to space limitations, only two representative instances, *Marvin* and

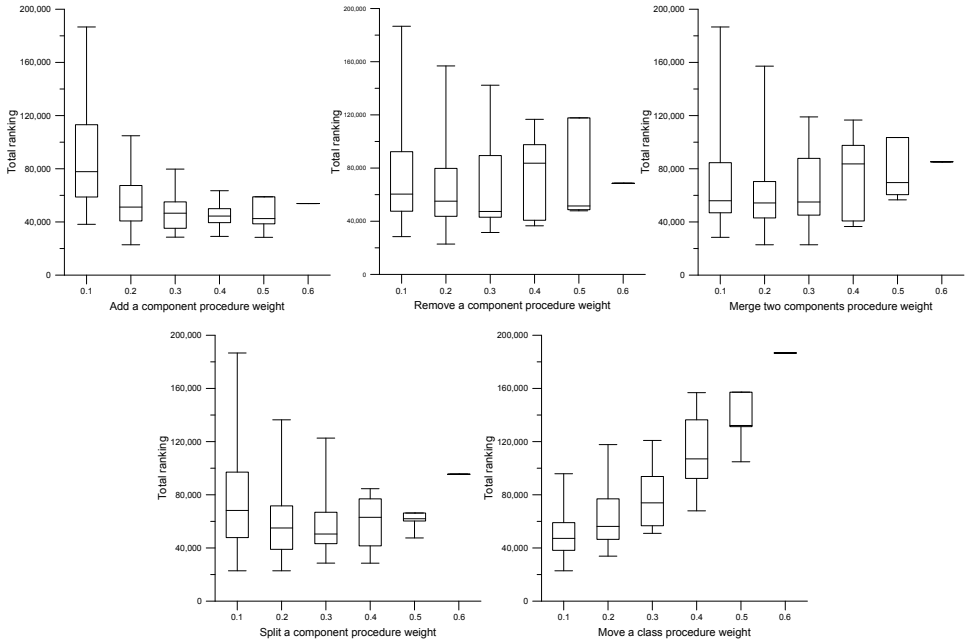


Fig. 3. Box plots of the distribution of best individuals found by the algorithm with different weights of the roulette in the mutation operator for *Marvin* problem instance.

Datapro4j, are shown and discussed next. Figs. 3 and 4 represent five box plots, where each one shows the distribution of the overall ranking value of the algorithm when each mutator weight is fixed at certain value in the range [0.1,0.6], whereas the others are combined in order to complete the roulette (the sum of all weights must be 1). Note that if the weight is fixed to 0.6, only one configuration can be generated for the rest of procedures (all fixed to 0.1), so a single line is drawn, representing the global ranking of this combination, i.e. the sum of the rankings of the best individual found in each of the 30 executions.

As for *Marvin*, interesting tendencies of fitness variation in most of the mutation procedures can be appreciated in Fig. 3. As the probability of the *Move a class* procedure is increased, the overall ranking of the algorithm is significantly punished. On the contrary, addition of components is beneficial. The *Merge two components* and *Split a component* procedures show an intermediate behaviour, where low or median weights seem to be more appropriate. In general, a trade-off between the removal and the addition of components is advisable to obtain an improvement of the quality of the

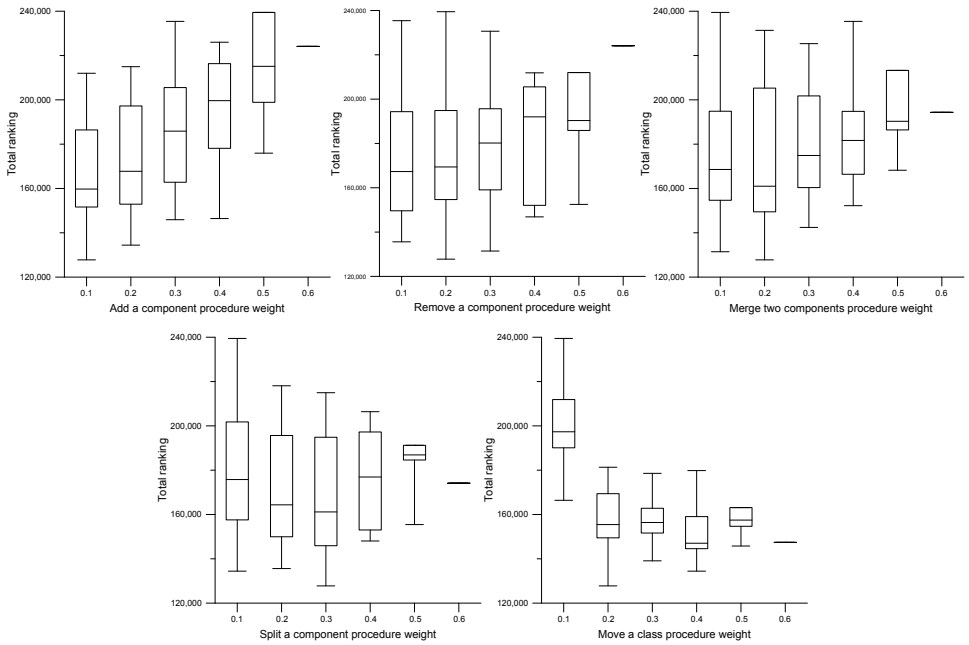


Fig. 4. Box plots of the distribution of best individuals found by the algorithm with different weights of the roulette in the mutation operator for *Dataprо4j* problem instance.

solutions without a loss of the population diversity.

Fig. 4 shows the box plots regarding *Dataprо4j*, a problem instance having more classes and relationships. The *Add a component* procedure presents worse performance than for *Marvin*, specially for high weight values, so its variation influences more strongly here than in simpler problems. *Remove a component* and *Merge to component* present an ascendant tendency, low weights being preferable. On the contrary, the algorithm behaviour with respect to the *Split a component* procedure is similar to the observed for *Marvin*, *Dataprо4j* having a slightly higher variation for each fixed weight. Finally, it seems that the search process can get better results for this kind of problem when the movement of classes is promoted. The algorithm performance is strongly impacted by those procedures that imply a large change in the solutions.

As can be seen, the algorithm behaviour substantially relies on the current weight configuration. In general, high values are not advisable in most procedures. The removal of components deserves special mention, where low values are highly expedient to restrict its influence on the overall perfor-

mance. The default weight of the *Add a component* procedure can remain unaltered, since it contributes to keep the trade-off necessary between the different problem instances. Considering the results obtained by the procedures *Split a component* and *Merge two components*, especially when applied to small-sized instances, a proper use of these procedures requires increasing their chances in the roulette selective process. Furthermore, notice that reducing too much the weight of the procedure *Move a class* could be harmful, as it is the only procedure maintaining the current structure of the individual to be mutated and, consequently, it promotes the convergence of the algorithm. This scenario is beneficial when going through the last algorithm iterations, specially for complex instances. In short, after scrutinizing the results obtained by the weight combinations satisfying all the constraints aforementioned, the proposed configuration is $w_{add} = 0.2$, $w_{remove} = 0.1$, $w_{merge} = 0.1$, $w_{split} = 0.3$ and $w_{move} = 0.3$.

5.2.3. Number of evaluations and population size

The previous experiments were focused on the algorithm needs with respect to its exploration and convergence. Furthermore that, even when the algorithm is able to properly deal with invalid individuals and diverse solutions, a further analysis of the most appropriate combination between the number of evaluations and the population size is still required.

Four different population sizes, from 50 to 200 individuals, have been set. The fitness convergence has been checked every 1,200 evaluations, up to 24,000 evaluations. Notice that previous experiments have reach a maximum of 10,000 evaluations, 100 individuals being evolved over 100 generations. Here, each variant of the algorithm has been iterated a different number of generations to fairly compare by the number of evaluations. These variants have been executed 30 times, as well.

Since the average fitness for the different runs is not representative, the following experimentation aims to evaluate the quality of the resulting solutions in terms of the three metrics comprising their fitness value separately. Again, the discussion is focused on the systems *Marvin* and *Dataprо4j* due to space limitations. On the one hand, Fig. 5 shows the convergence of the evolutionary process for the *Marvin* problem instance. The average values of the best individual found along the evaluations in terms of ICD, ERP and GCR are depicted for each considered population size. Remind that ICD should be maximised, whereas ERP and GCR should be minimised. As can be observed, the algorithm tends to perfom worse when the population size

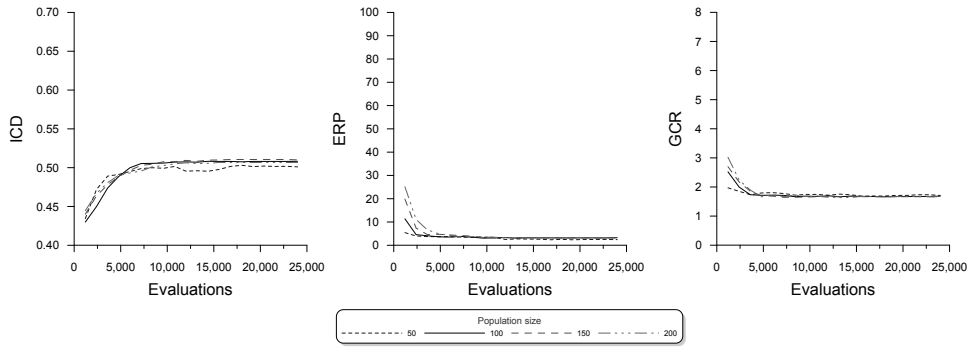


Fig. 5. Convergence of the algorithm for each selected population size (*Marvin*).

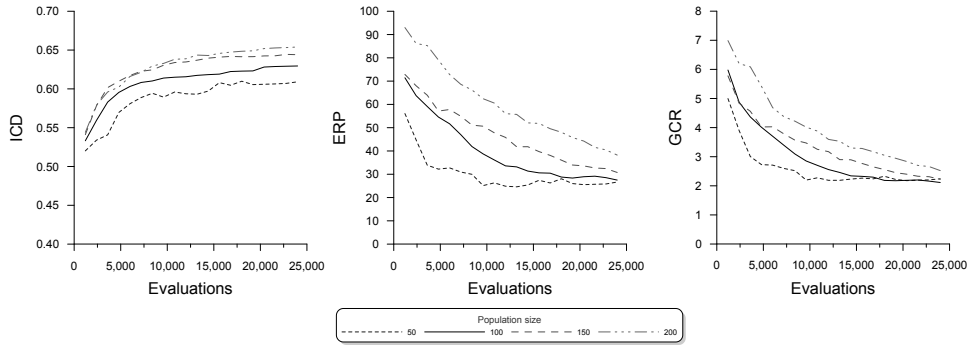


Fig. 6. Convergence of the algorithm for each selected population size (*Datapro4j*).

is set to 50 individuals, specially for the ICD metric. On the contrary, the algorithms evolving 100, 150 and 200 individuals behave similarly for the three metrics. Actually, a further analysis of the optimization process for each individual metric in relation to the number of evaluations shows that the four algorithms remain considerably steady beyond 10,000 evaluations. In fact, only some slight improvement concerning the ICD is obtained. Thus, standard values for both the population size and the number of evaluations, like those considered in previous experiments, still work properly for small problem instances.

On the other hand, Fig. 6 shows how the evolution for the *Datapro4j* problem takes place. In constrast with the *Marvin* instance, substantial differences among the different algorithms are noticeable. Firstly, the problem complexity clearly hampers the jointly optimization of the considered metrics. In this case, the algorithm with the population consisting of 50

individuals is prone to optimize ERP and GCR better than ICD. Just the opposite occurs when the population size has been set to 200 individuals. Then, the influence of the number of evaluations becomes important, and the value previously set to 10,000 is not sufficient to achieve good enough results.

Summarizing the experimental findings, the recommended population size is 150, whereas the number of maximum evaluations has to be fixed between 20,000 and 24,000, depending on the problem complexity. Here, the number of evaluations is set to 20,400 evaluations, which corresponds with 136 generations for the selected population size. In regard to RQ2, this study has served to find the configuration that best enhances the performance of the evolutionary algorithm. Given this configuration, the algorithm achieves good results for all the considered design metrics and problem instances, satisfactorily keeping the trade-off between exploration and exploitation.

6. An illustrative example of the approach

After explaining the setup process of the proposed algorithm, a more detailed description about how the evolutionary search operates through the generations can be illustrative. Due to space limitations, this section focuses on a simple example, which allows both intermediate and final solutions to be shown. This example requires a small number of generations and population size, fixed to 10 and 5 respectively, whilst the other parameters remain unaltered.

Fig. 7a shows the sample analysis model used as case study. As can be observed, it has 8 classes related among them with different types of relationships: 1 association, 2 dependences, 1 composition, 1 aggregation and 2 generalizations. Moreover, two groups of well-connected classes can be distinguished: *A-B-C-D* and *E-F-G-H*. The rest of snapshots in Fig. 7 represent the phenotype of the best individuals found at different stages of the search, as well as its quality in terms of the proposed metrics and fitness values.

As can be seen in Fig. 7b, the best individual in the initial population presents an architecture composed by 3 components, in which all the classes are randomly distributed. This architecture is, as expected at this evolutionary step, a non-optimal solution. More specifically, there are some external relationships among classes belonging to different components and a

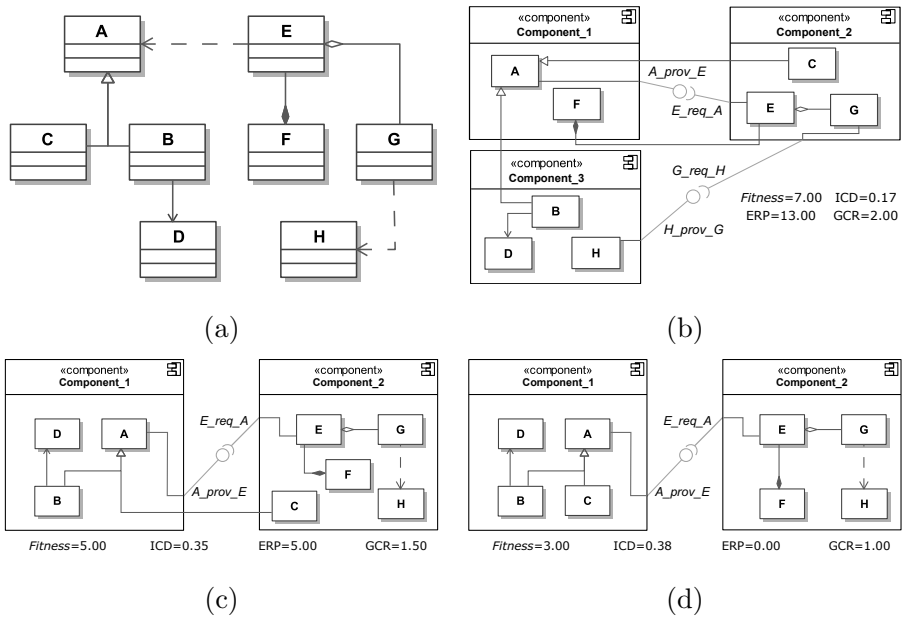


Fig.7. Phenotype of best individual, for the class diagram which is shown in (a), found in the initial population (b), after 5 generations (c), and the final solution after 10 generations (d).

few relationships among the inner classes on each one. Since two generalizations and one composition appear outside the components boundaries, then $ERP = 13.00$. Similarly, the GCR can be obtained from the number of groups (two per component) and the total number of components, so $GCR = 6.00/3.00 = 2.00$. Finally, ICD is calculated as the sum of ICD_i (from $i = 1...3$). For this solution, its first component does not comprise any internal relationship between its inner classes and therefore $ICD = (ICD_{Component\_2} + ICD_{Component\_3})/3 = (0.21 + 0.31)/3.00 = 0.17$. At this stage, the corresponding solution has not the minimum ranking, but a low value representing that the individual is fairly good enough for some of the design criteria.

After 5 generations (see Fig. 7c), the ERP metric has been reduced because the genetic operator has grouped together more related classes. For example, classes E, F, G and H belong now to the same component, creating a well-connected group of classes that implement the functionality of $Component\_2$. GCR has also been improved, since the number of groups of

classes has been reduced and only *Component_2* presents more than one separate group. Finally, ICD also achieves a greater value than the previous best individual, as there is a better trade-off between internal and external interactions of components, i.e. classes are combined more adequately to provide the functionalities of the system with fewer dependences among components. As a result of the improvements in all the values, the fitness achieves the minimum ranking value, 3.00, meaning that this individual represents the best solution in all the proposed design characteristics in comparison with the rest of members in the current population, even though it does not achieve the optimum value for every metric. For example, $ERP = 5.00$ since there is a generalization between the separate classes A and C. Moreover, the structure of the solution has also been modified, showing that the algorithm has got to a simpler architecture.

Then, after 10 generations updating the distribution of classes (see Fig. 7d), the fitness value representing the solution quality has been significantly improved, reflecting the suitability of the solution found in terms of a good design that effectively identifies both groups of classes. Firstly, all the metrics achieve optimal values: each component presents an unique group of classes that implements its functionality, these classes are highly cohesive inside the component, and components only interact by means of a pair of interfaces. The solution also represents the simplest architecture from those obtained in previous generations, showing that the algorithm is able to adapt the structure of the solution through the evolution process.

7. Results and discussion

Table 5 shows final average results of the proposed configuration from 30 executions, including the execution time. The standard deviation is shown as subindexes. An important aspect concerning *RQ2* is that the evolutionary algorithm is able to discover good solutions [6] in the major problem instances.

For example, the algorithm achieves very good GCR results, close to the minimum, 1.0. Average fitness ranking values are omitted because they depend on each specific execution and therefore they are not representative. Nevertheless, low values are usually obtained, meaning that the architecture

[6] A comparative analysis of the fitness metrics applied to a manually produced software architecture can be found at `http://www.uco.es/grupos/kdis/sbse/RRV14`.

Table 5: Final results of the proposed algorithm

Problem	ICD	ERP	GCR	Time (ms)
Aqualush	$0.4124_{0.0604}$	$6.2333_{3.8443}$	$1.0833_{0.1708}$	$116.1019_{8.4891}$
Borg	$0.2820_{0.0689}$	$3.9333_{2.4626}$	$1.1667_{0.2274}$	$2,489.1223_{171.2028}$
Datapro4j	$0.6425_{0.0356}$	$33.6000_{14.7436}$	$2.3989_{0.6744}$	$37.1101_{2.8308}$
Java2HTML	$0.2593_{0.0000}$	$0.0000_{0.0000}$	$1.0000_{0.0000}$	$250.4572_{7.4674}$
JSapar	$0.3751_{0.0307}$	$9.0000_{1.5492}$	$1.1667_{0.2981}$	$94.8891_{5.1418}$
Marvin	$0.5080_{0.0187}$	$3.1667_{1.0980}$	$1.6750_{0.1146}$	$14.1194_{0.6885}$
NekoHTML	$0.4594_{0.0345}$	$3.2667_{5.3037}$	$1.2389_{0.3768}$	$57.1150_{3.5524}$

returned by the discovery process is the best for at least one or two of the design criteria compared to the rest of solutions found in the final population.

Focusing on the trade-off between ICD and ERP, some differences can be established between the considered problem instances. Firstly, for software designs like *JSapar* or *NekoHTML*, smaller architectures are sufficient to abstract their designs, so the ICD does not need to achieve very high values. On the contrary, with more complex problem instances (*Borg* and *Java2HTML*), the behaviour of the algorithm is quite different, where the ERP metric, which is likely to be the most difficult metric to be optimised, is highly improved. The reason is that these systems are the most complex ones, as they present a great number of interactions among classes. Therefore, the algorithm is able to find an equivalent type of solution, consisting in components that cover many dependent functionalities that cannot easily be dispersed in smaller components without a dramatic increase of ERP values.

Other interesting analyses can be made with *Aqualush* and *Datapro4j*, which could be considered equivalent problems, because of their similar number of classes. However, the obtained results clearly show that it is not a really relevant characteristic for the performance of the algorithm. More precisely, the identification of the *Datapro4j* architecture produces the poorest ERP value of all the considered instances. The reason is that this system is strongly hierarchical, presenting a considerable amount of generalizations in contrast to other problem instances, where associations and dependencies are the most common relationships in the system specification. This has an impact in the ERP values, since generalizations determine the maximum penalty in this measure, i.e. hierarchically-dependent classes usually tend

to belong to the same component. Hence, although the number of external relationships could be similar to those obtained in the other cases, the computed ERP dramatically increases. In contrast, ICD values in *Datapro4j* are better than in *Aqualush*. This behaviour shows that the desirable trade-off between coupling and cohesion becomes difficult to achieve, not only as the amount of classes and relationships increases but also depending on the sort of software specification.

Focusing on the execution time, it is possible to determine some type of relation between the characteristics of the problem instances and the required time to perform the process. In this case, the number of relationships between classes clearly have an impact on the execution time. Medium or small instances only need a few seconds to complete the process, whereas more complex software specifications require several minutes. The underlying cause is that as the number of interactions increases, it is more difficult to generate architectures under the existing constraints, and the algorithm requires more mutation operations to obtain valid individuals.

Finally, in response to *RQ1*, some interesting information can also be extracted after studying the solutions obtained from an architectural perspective. The evolutionary algorithm provides solutions that identify and allocate well-connected groups of classes into components that correctly match with the possible intended architecture. In this way, results from the evolutionary process supply the software architect with valuable information that could be properly used to analyse the strengths and weaknesses of the system structure, reconsider some design decisions made and explore different configurations to appropriately mitigate risks. For example, some large components might be returned if the amount of relationships among classes is excessively high. Here, the software designer should remodel their interdependencies in order to get differentiate functional groups. It would reduce the system complexity and benefit maintenance and reusability. Moreover, it can be noted that the presented model is able to evolve and keep solutions with different architectural structures of interest during the search.

8. Concluding remarks

Making decisions during the software design process requires important human-centered contributions and skills that could be mitigated by search-based approaches, which are able to easily cover a great number of design alternatives. With the ultimate aim of providing support for such a decision

making process, this paper presents a single-objective evolutionary approach for the discovery of component-based software architectures from analysis models, where classes and their relationships are used in the search of architectural artefacts, like components and interfaces. This proposal constitutes the first approximation to semi-automatic architectural analysis as a way to help software engineers in the improvement of their highly abstract designs which facilitate the understanding of the software foundation.

The approach is conceived as an exploratory mechanism for decision support. The underlying methodology is focused on the comprehension of the metaheuristic formulation of the problem by the software architect. Moreover, the consideration of standards like UML 2 and XMI promotes the integrability of this approach within the software engineering communities and modelling tools.

The proposed encoding is based on tree structures, similarly to the way in which specification models are handled by the different tools in this domain, bringing a flexible and intuitive representation of software architectures. Design alternatives are explored by means of five different types of mutation procedures based on those architectural transformations that software engineers usually perform for their specifications. The fitness function is calculated as the ranking aggregation of design criteria, like coupling and cohesion, as well as some specific characteristics to the problem formulation. Focusing on the search for well-defined functionalities, the optimisation model considers the minimisation of interactions among classes and interfaces, as well as the presence of well-connected groups on classes inside the components.

The influence of the parameter setup has been discussed in detail in order to properly tune the method for semi-automatic architectural modelling, which has been tested over diverse software systems. The obtained results demonstrates the algorithm capabilities in the management of different types of solutions as well as a trade-off between the conflicting metrics, showing that the automatic discovery of the system architecture constitutes a difficult and stimulating problem.

The evolutionary approach has been conceived to deal with component-based architectures, even when it could serve as a basis for being applied to other sorts of design paradigms and areas. For example, dealing with service oriented architectures would imply a further study of the suitability of other factors, like cost and response time, whilst a model extended to comprise low-level details, like methods and properties, could serve to deal with refactoring tasks. Future research will explore the inclusion of the expert's opinion in

the evolutionary search.

Acknowledgements

Work supported by the Spanish Ministry of Science and Technology, project TIN2011-22408, and FEDER funds. This research was also supported by the Spanish Ministry of Education under the FPU program (FPU13/01466).

References

[1] M. Abdellatief, A. B. M. Sultan, A. A. A. Ghani, and M. A. Jabar. A mapping study to investigate component-based software system metrics. *J. Sys. Soft.*, 86:587–603, 2013.

[2] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Trans. Softw. Eng.*, 39(5):658–683, 2013.

[3] P. Baker, M. Harman, K. Steinhofel, and A. Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In *22nd IEEE Int. Conf. on Software Maintenance*, pages 176–185, 2006.

[4] M. F. Bertoa, J. M. Troya, and A. Vallecillo. Measuring the usability of software components. *J. Sys. Soft.*, 79(3):427–439, 2006.

[5] D. Birkmeier and S. Overhage. On Component Identification Approaches: Classification, State of the Art, and Comparison. In *Proc. 12th Int. Symp. on Component-Based Software Engineering*, pages 1–18, 2009.

[6] J. Bosch and P. Molin. Software architecture design: evaluation and transformation. In *Proc. IEEE Conf. and Workshop on Engineering of Computer-Based Systems*, pages 4–10, 1999.

[7] I. Boussaïd, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Inf. Sci.*, 237(0):82–117, 2013.

[8] J. A. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. F. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. J.

Shepperd. Reformulating Software Engineering as A Search Problem. *IEEE Proceedings - Software*, 150(3):161–175, 2003.

[9] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.*, 28(7):638–653, 2002.

[10] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Trans. Softw. Eng.*, 35(4):573–591, 2009.

[11] R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron. Problem-specific search operators for metaheuristic software architecture design. In *Proc. 4th Int. Symp. on Search Based Software Engineering*, pages 267–272. Springer, 2012.

[12] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba. Evolutionary Algorithm for Prioritized Pairwise Test Data Generation. In *Proc. 14th Genetic and Evolutionary Computation Conference*, pages 1213–1220, 2012.

[13] D. Garlan. Software architecture: a roadmap. In *Proc. 22th Int. Conf. of Software Engineering*, pages 91–101, 2000.

[14] P. D. S. Gui Gui. Measuring Software Component Reusability by Coupling and Cohesion Metrics. *Journal of Computers*, 4(9):797–805, 2009.

[15] P. Gupta, S. Verma, and M. Mehlawat. Optimization Model of COTS Selection Based on Cohesion and Coupling for Modular Software Systems under Multiple Applications Environment. In *Computational Science and Its Applications*, volume 7335 of *LNCS*, pages 87–102. Springer, 2012.

[16] M. Harman. Software Engineering Meets Evolutionary Computation. *Computer*, 44(10):31–39, 2011.

[17] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based Software Engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–61, 2012.

[18] S. M. H. Hasheminejad and S. Jalili. An evolutionary approach to identify logical components. *J. Sys. Soft.*, 96(0):24–50, 2014.

[19] ISO. *ISO/IEC FDIS 42010/D9. Systems and software engineering - Architecture description*, mar 2011.

[20] A. C. Jensen and B. H. Cheng. On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns. In *Proc. 12th Genetic and Evolutionary Computation Conference*, pages 1341–1348, 2010.

[21] S. Kebir, A.-D. Seriai, A. Chaoui, and S. Chardigny. Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. In *Proc. 5th Int. C* Conference on Computer Science and Software Engineering*, pages 1–8, 2012.

[22] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. In *Proc. 14th Ann. Conf. on Genetic and Evolutionary Computation*, pages 959–966, 2012.

[23] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An Extensible Framework for Improving a Distributed Software System's Deployment Architecture. *IEEE Trans. Softw. Eng.*, 38(1):73–100, 2012.

[24] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, and R. Reussner. A Hybrid Approach for Multi-attribute QoS Optimisation in Component Based Software Systems. In *Proc. 6th Int. Conf. on the Quality of Software Architectures*, pages 84–101, 2010.

[25] V. L. Narasimhan and B. Hendradjaya. Some theoretical considerations for a suite of metrics for the integration of software components. *Inf. Sci.*, 177(3):844–864, 2007.

[26] OMG. *Unified Modeling Language 2.4 Superstructure Specification*, nov 2010.

[27] C. M. Poskitt and S. Poulding. Using Contracts to Guide the Search-Based Verification of Concurrent Programs. In *Proc. 5th Int. Symp. on Search Based Software Engineering*, pages 263–268, 2013.

[28] K. Praditwong, M. Harman, and X. Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Trans. Softw. Eng.*, 37(2):264–282, 2010.

[29] O. Räihä. A survey on search-based software design. *Comput. Sci. Rev.*, 4(4):203–249, 2010.

[30] P. Rodríguez-Mier, M. Mucientes, M. Lama, and M. Couto. Composition of web services through genetic programming. *Evol. Intell.*, 3:171–186, 2010.

[31] O. Sievi-Korte, E. Mäkinen, and T. Poranen. Simulated annealing for aiding genetic algorithm in software architecture synthesis. *Acta Cybernetica*, 21(2):235–265, 2013.

[32] C. L. Simons and I. C. Parmee. Elegant Object-Oriented Software Design via Interactive, Evolutionary Computation. *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev*, 42(6):1797–1805, 2012.

[33] C. L. Simons, I. C. Parmee, and R. Gwynllyw. Interactive, Evolutionary Search in Upstream Object-Oriented Class Design. *IEEE Trans. Softw. Eng.*, 36(6):798–816, 2010.

[34] J. Smith and D. Stotts. SPQR: flexible automated design pattern extraction from source code. In *Proc. 18th IEEE Int. Conf. on Automated Software Engineering*, pages 215–224, 2003.

[35] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman., Boston, MA, 2nd edition, 2002.

[36] L. Troiano and C. Birtolo. Genetic algorithms supporting generative design of user interfaces: Examples. *Inf. Sci.*, 259(0):433–451, 2014.

[37] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás. JCLEC: a Java framework for evolutionary computation. *Soft Comput.*, 12(4):381–392, 2008.

[38] D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Inf. Softw. Technol.*, 43(14):817–831, 2001.

[39] Y. Zhang, M. Harman, and S. L. Lim. Empirical Evaluation of Search Based Requirements Interaction Management. *Inf. Softw. Technol.*, 55(1):126–152, 2013.