A Genetic Approach for Software Architecture Recovery from Object-Oriented Code

Abdelhak-Djamel Seriai
LIRMM, University of Montpellier 2/CNRS
162 rue Ada
F-34392 Montpellier Cedex 5, France
seriai@lirmm.fr

Sylvain Chardigny
MGPS
Port-Saint-Louis,
France
chardigny.sylvain@gmail.com

***Abstract*— Software architecture is recognized as a critical element in the successful development and evolution of software-intensive systems. Despite the important role of architecture representation and modeling many existing systems like legacy or eroded ones do not have a reliable architecture representation. In this paper we present an approach for architecture recovery from object-oriented code. It's based on a genetic algorithm which uses a fitness function measuring the semantic-correctness of software components. Following our model, architecture which is a partition of classes is considered as a chromosome. A group of classes is a gene. This algorithm gives satisfactory results in terms of consistency and adequacy metrics.**

Keywords- Component; software architecture; recovery; component based; object oriented; reverse engineering

I. INTRODUCTION

Software architecture is recognized as a critical element in the successful development and evolution of software-intensive systems [5]. Software architecture expresses the overall structure of a system in an abstract, structured manner. The main goal of a software architectural representation of a system is to identify the major components that constitute this system, and the interactions between these components [1]. According to Garlan [6], software architecture plays an important role in at least six aspects of software development: understanding, reuse, construction, evolution, analysis and management. Despite the important role of architecture representation and modeling, many existing systems do not have a reliable architecture representation. Indeed these systems could have been designed without an architecture design phase, as it is the case of most legacy systems. For other systems, the available representation can diverge from the system implementation. This appears, first, during the implementation phase due to gaps between the expected architecture and the implemented one. These gaps become greater because of lack of synchronization between software documentation and implementation. Taking into account the previous considerations, it is obvious that an approach of architecture recovery allows architects and developers to take advantage of all the benefits of having an architecture model available. In this context, we propose an approach to extract a component-based architecture from object-oriented systems. Our goal is to decrease the need for human expertise which is expensive and not always available. Our process aims at selecting among all the architectures which can be abstracted from a system, the best one according to the semantic-correctness of architecture. Based on the norm ISO 9126 [10], we formulate these characteristics as measurable properties and specify the recovery process as a balancing problem of these ones. Based on this formulation, we developed a recovery architecture process exploiting a hierarchical clustering algorithm. Nevertheless, the result of this process (i.e. architectures) was not completely satisfactory. We studied the process on several case studies (e.g. Jigsaw, ArgoUML, Eclipse, etc.). The results were sometimes offset from the known architectures. This is due to the nature of the clustering algorithm. In fact, this heuristic algorithm is memory less. For some systems, this feature may deter the clustering process from the best architectures. It explores only a limited number of possible ones. It is not a meta-heuristic and it does not use the space exploration of all possible architectures. Thus, our objective in this paper is to propose an alternative formulation of our approach based on a genetic algorithm (GA). This choice is due to the characteristics of this type of algorithm. A GA allows us to consider architecture recovery as a metaheuristic optimization problem. It aims to explore the solution space to identify the best possible. GAs were introduced in the late 1960's by John Holland [7]. They are based on the Darwinian theory of evolution whereby species compete to survive and the fittest get a higher chance to remain until the end and produce progeny. The basic idea of a GA is to start from a set of initial solutions, and to use biologically inspired evolution mechanisms to derive new and possibly better solutions.

The remainder of this paper is structured as follows. Section 2 presents principle of our architecture recovery approach: semantic-correctness driven. The GA encoding of the recovery process is presented in section 3. Case studies are presented in section 4. They present the results of our approach using GA and compare these results to the clustering-based ones. Section 5 discusses related work. Conclusion and future works are given in section 6.

II. SEMANTIC-CORRECTNESS DRIVEN ARCHITECTURE RECOVERY : AN OVERVIEW

In our approach, recovering a component-based architecture of an object oriented system consists of using its implementation code in order to identify the architectural elements. As first step toward this goal, we defined a mapping model of object

oriented concepts (i.e. classes, methods, interfaces, packages, etc.) and architectural ones (i.e. components, connectors, interfaces, etc.). It defines architecture as a partition of the system classes. Each element of this partition represents a component. These elements are named "shape" and include classes which can belong to different object-oriented packages. A shape is composed of two sets of classes: the "shape interface" includes classes linked with others from the outside of the shape, e.g. a method call to the outside; and the "center" composed of the remainder classes of the shape. We assimilate component to shape and component interfaces to "shape interface". Connectors are all links existing between components. Consequently, the architecture configuration is the set of shapes constituting a partition of the system classes. As a result of these considerations, the search-space of architecture recovery problem is composed of all architectures which are partitions of the system classes. This means that, in a system which contains *n* classes, the search-space contains *O (n!)* potential architectures.

Thus to select the best architecture compared to its semantic correctness, we propose to define fitness function measuring this characteristic. An architecture is semantically correct if its elements (components, connectors and configuration) are. We limit ourselves here to study component semantic-correctness and define function to measure it. This study is based on the most commonly admitted definitions of software component rather than architectural one. Indeed architectural component constraints are included in the software component ones. In addition these supplementary constraints make easier a migration from an object-oriented system to a component-based one.

A. *Semantic characteristics of a software component*

Szyperski defines in [17] a component as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. In [9] Heinemann and Councill define a component as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. In [12] Luer makes a distinction between component and deployable component. He defines a component as a software element that (a) encapsulates a reusable implementation of functionality, (b) can be composed without modification, and (c) adheres to a component model. A deployable component is (a) prepackaged, (b) independently distributed, (c) easily installed and uninstalled and (d) self-descriptive.

In combining and refining the common elements of these definitions and others commonly accepted ones [11], we propose the following definition of a component: A component is a software element that (a) can be composed without modification, (b) can be distributed in an autonomous way, (c) encapsulates an implementation of functionality, and (d) adheres to a component model. In our approach, the definition of a component model is the Luer one [12]: a model component is the combination of (a) a component standard that governs how to construct individual components and (b) a composition standard that governs how to organize a set of components into an application and how those components globally communicate and interact with each other. As compared to the definitions of Luer and Heineman and Councill, we intentionally do not include the criterion that a component must adhere on a composition theory and the properties of component self-descriptive, pre-packaged and easy to install and uninstall. These are covered through the criterion that a component must adhere to a component model and does not need to be repeated. In conclusion, according to our software component definition, we identify three semantic characteristics of software components: composability, autonomy and specificity. The specificity of a component means that it must contain a limited number of functionalities.

B. *Refinement model of the semantic correctness*

In the previous section, we have identified three semantic characteristics that we propose to evaluate. To do so, we adapt the characteristic refinement model given by the norm ISO-9126 [ISO]. According to this model, we can measure the characteristic semantic correctness by refining it in the previous three semantic characteristics which are consequently considered as sub-characteristics.

1) *From characteristic to properties*

Based on the study of the semantic sub-characteristics, we refine them into a set of component measurable properties. Thus, a component is autonomous if it has no required interface. Consequently, the property number of required interfaces should give us a good measure of the component autonomy. Then, a component can be composed by means of its provided and required interfaces. However, a component will be more easily composed with another if services, in each interface, are cohesive. Thus, the property average of service cohesion by component interface should be a correct measure of the component composability. Finally, the evaluation of the number of functionalities is based on the following statements. Firstly a component which provides many interfaces may provide various functionalities. Indeed each interface can offer different services. Thus the higher the number of interfaces is, the higher the number of functionalities can be. Secondly if interfaces (resp. services in each interface) are cohesive (i.e. share resources), they probably offer closely related functionalities. Thirdly if the code of the component is closely coupled (resp. cohesive), the different parts of the component code use each other (resp. common resources). Consequently, they probably work together in order to offer a small number of functionalities. From these statements, we refine the specificity sub characteristic to the following properties: number of provided interfaces, average of service cohesion by component interface, component interface cohesion and component cohesion and coupling.

2) *From properties to metrics*

According to our object-component/architecture model, component interfaces are assimilated to shape interface. Therefore, the *average of the interface-class cohesion* gives a correct measure of the *average of service cohesion by component interface*. Secondly the *component interface cohesion*, the *internal component cohesion* and the *internal*

component coupling can respectively be measured by the properties *interface class cohesion*, *shape class cohesion* and *shape class coupling*. Thirdly in order to link the *number of provided interfaces* property to a shape property, we associate a component provided interface to each shape-interface class having public methods. Thanks to this choice, we can measure the number of provided interfaces using the *number of shape interface classes having public methods*. Finally, the *number of required interfaces* can be evaluated by using coupling between the component and the outside. This coupling is linked to shape external coupling. Consequently, we can measure this property using the property *shape external coupling*. In order to measure these properties, we need to define metrics. The properties *shape class coupling* and *shape external coupling* require a coupling measurement. We define the metric *Coupl(E)* which measures the coupling of a shape *E* and *CouplExt(E)* which measures the coupling of *E* with the rest of classes. They measure three types of dependencies between objects: method calls, use of attributes and parameters of another class. Moreover they are percentages and are related through the equation: *CouplExt(E) = 100 − Coupl(E)*. Due to space limitations, we do not detail these metrics. Shape properties *average of interface-class cohesion*, *interface-class cohesion*, and *shape-class cohesion* require a cohesion measurement. The metric "Loose Class Cohesion" (LCC), proposed by Bieman and Kang [2], measures the percentage of pair of methods which are directly or indirectly connected. Two methods are connected if they use directly or indirectly a common attribute. Two methods are indirectly connected if a connected method chain connects them. This metric satisfies all our needs for the cohesion measurement: it reflects all sharing relations, *i.e.* sharing attributes in object oriented system, and it is a percentage. Consequently, we use this metric to compute the cohesion for these properties. The refinement model is summarized in Fig.1.

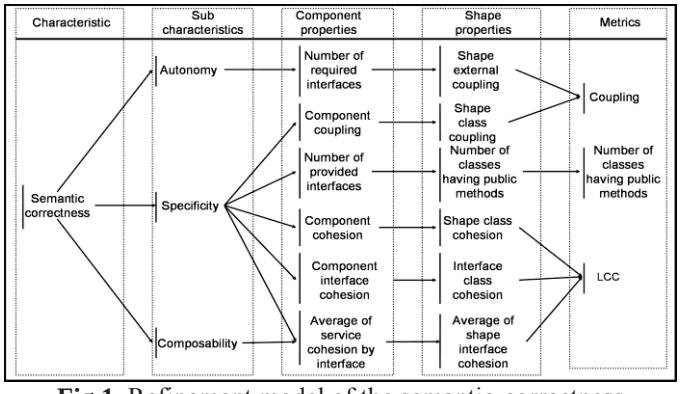

Fig.1. Refinement model of the semantic-correctness characteristic of component

3) Evaluation of the semantic correctness

According to our refinement model of semantic-correctness of component, we define the functions *Spe*, *A*, *C* which measure respectively specificity, autonomy and composability of this component. In these functions *nbPub(I)* is the number of interface classes having a public method and IiI is the shape interface cardinality.

$$Spe(E) = \frac{1}{5} \cdot \left(\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Coupl(E) + nbPub(I) \right)$$

$$A(E) = couplExt(E) = 100 - Coupl(E)$$

$$C(E) = \frac{1}{|I|} \sum_{i \in I} LCC(i)$$

The evaluation of the *semantic correctness* characteristic is based on the evaluation of each sub-characteristic. That is why we define this function as a linear combination of each sub-characteristic evaluation function (*Spe*, *A*, and *C*):

$$S(E) = \frac{1}{\sum_i LCC(i)\mu_i} \left(\mu_1 Spe(E) + \mu_2 C(E) + \mu_3 A(\right.$$

This form is linear because each of its parts must be considered uniformly. The weight μ_i associated with each function allows the software architect to modify, as needed, the importance of each sub-characteristic.

III. GENETIC MODEL FOR ARCHITECTURE RECOVERED FROM OBJECTI-ORIENTED CODE

GA starts derivations from an initial solution called the initial population and then generates a sequence of populations. Each derived population is obtained by "mutating" the previous one. Elements of the obtained solutions are called chromosomes. The fitness of each chromosome is measured by an objective function called the fitness function. Each chromosome (possible solution) consists of a set of genes. At each generation, the process consists to apply some genetic operators which are crossover, mutation and selection in order to generate the next generation. On each chromosome, the algorithm applies two operators: crossover and mutation. Each operator is applied following a specific probability given as an input parameter of the algorithm. During crossover, two chromosomes are selected using a selection method that gives priority to the fittest ones; they exchange some of their genes giving birth to two other chromosomes. Each selected pair of chromosomes produces a new pair of chromosomes that constitute the next generation. Mutation consists of changing randomly one or more genes in a chromosome. Finally a selection is operated on chromosomes to choose the next generation. This selection can increase or reduce or keep stable the size of the population. The algorithm stops if a convergence criterion is satisfied or if a fixed number of generations is reached. The implementation of GA to recover architecture requires specifying how solutions are encoded into chromosomes, how the three genetic operators crossover, mutation and selection are defined and which fitness function and initial population to be used. As we have already defined the function in the section II.B.3, we address the remaining questions in the following sections.

A. Encoding architecture as chromosome

Our genetic model of architecture must adhere to our object-component/architecture mapping model indicating that

an architecture is a partition of classes (cf. section II). This can be translated following different possible formulations. One of these formulations is to represent architecture as a chromosome. Thus, in this model a chromosome is a partition of classes. Another formulation is to model shape as chromosome and architecture as the whole population. This choice makes difficult to check partition property. Instead of the basic idea of GA which aims to optimize one element in a population, this model aims to optimize one population. We opt for representing architecture as a chromosome. Therefore this requires defining the genes which constitute the chromosomes. Again several options are available. Thus a gene may represent a class and the value of the gene may represent the shape which contains this class. An alternative formulation would be to represent a shape as a gene and then the value of the gene would be a set of classes. We opt for the second formulation because it makes easier to check the partition property. Indeed the union of the gene values must be all the system classes and each intersection of gene values must be empty.

B. *Definition of the genetic operators*

In order to apply GA we need to define the genetic operators. These are the selection, the crossover and the mutation operators. The process of evolution starts by selecting several pair of chromosomes whose the number vary according to a probability *PC*. Then the crossover is applied on each pair to generate two new chromosomes. The mutation is applied to each chromosome (new and old) with the probability *PM*. *PC* and *PM* are given as given as parameters of GA. Finally some chromosomes are selected for the next generation. We present in the following each of these three operators.

1) *Selection operator*

There are two selection operators which are used in GA. The first one selects the pair of chromosomes for the crossover and the second selects the next generation among the chromosomes. The selection of the chromosome pair is done according to the roulette-wheel technique [7]. Each chromosome is assigned a portion of the wheel that is proportional to its fitness. A marble is thrown and the chromosome where the marble halts is selected. The selection of the chromosomes for the next generation is done according to two criteria: the age of the chromosomes and their fitness. The new chromosomes are automatically added to the next generation. As we decided to keeps the population size constant, the other chromosomes are selected among the old chromosomes according to the fitness function.

2) *Crossover operator*

A standard way to perform the crossover operation on chromosomes is to cut each of the two parent chromosomes into two subsets of genes (shapes in our case). Two new chromosomes are created by interleaving the subsets. If we apply such operation, it is possible that the resulting chromosomes can no longer represent well-defined partitions. Two specific problems can occur. If the intersection of two shapes is not empty, then the solution is inconsistent. The second problem is when the solution is incomplete. This occurs when the union of all the shapes does not contain all the system classes. In both cases, the architecture represented by the chromosome is not even a partition. Figure 2(a) illustrates these two situations. To preserve the consistency and the completeness of the offspring, we propose a crossover operator based on the operator defined for grouping problems [4]. To obtain an offspring, we select a random subset of shapes from one parent and add it to the set of shapes of the second parent. By keeping all the shapes of one of the parents, completeness of the offspring is automatically ensured. To guarantee consistency, we eliminate from the older shapes, the classes contained in the added shapes. Figure 2(b) illustrates the new crossover operator.

3) *Mutation operator*

Mutation is a random change in the genes that happens with a small probability. In the case of our architecture recovery model, the mutation operator randomly moves some classes from one shape to another one. This mutation operator keeps the partition property safe.

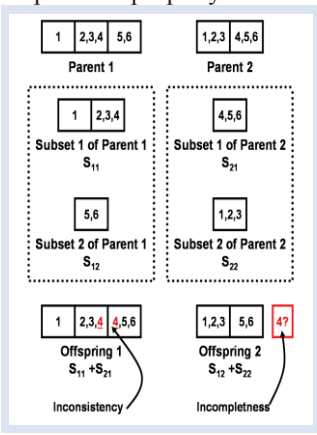

Fig.2 (a) insufficient of the standard crossover

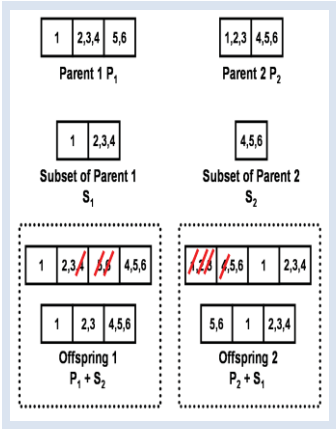

Fig.2 (b) Crossover that preserves consistency and completeness

C. *Choice of the initial population*

The initial population is the set of chromosomes which is used at the start of the GA. The choice of this population is often randomly made. Nevertheless better the initial population is better the solutions is. We opt for an initial population which represents partition obtained by the strongly-connected components of the system classes. In this graph whose vertices are the system classes, there is an arc between two vertices A and B if the class A uses the class B, i.e. uses an attribute, a parameter or a return variable whose type is B and creates or uses an object whose type can be B. A Strongly-connected component is a subset of the graph vertices where any vertex can be reached from any other vertex by a path. These strongly-connected components are a partition of the graph vertices and consequently a partition of the system classes. This partition is an approximation of the system architecture. We use it as an initial chromosome for our GA.

IV. CASE STUDY

As case studies, we validate the genetic algorithm implementation of our architecture recovery process on many systems with different sizes: small systems whose number of classes is less than 50 (e.g. JPhotoAlbum with 17 classes),

systems of medium size where the number of classes is between 50 and 500 (e.g. Jigsaw with 300 classes) and larger systems with more than 1,000 classes. (e.g. ArgoUML with over than 1500 classes). In most cases, the results show that the recovered architectures are closer to the known architectures than those obtained by the clustering algorithm. The difficulty was to choose the adequate parameters for GA. Due to space limitation, we give below only the case study of the Jigsaw system which is a Java based web server.

A. *GA parameters*

To execute the GA we have to determine some parameters. These are the elements of the initial population, the rate of crossover and mutation, the size of the initial population, and the number of generation. We launch several tests in order to analyze the impact of each parameter on the result. Firstly, we choose to use an initial population based on randomize partitions plus one partition calculated from the strongly-connected components of the system classes. Secondly, the tests realized show that the elements of the population become similar all along the process. This is due to our crossover operator. To avoid having only one element in the population after some generation, we choose a great rate of mutation. Indeed we choose to put the mutation and the crossover rate to 80 %. Nevertheless, this choice of rate is not enough to palliate totally the activity of the crossover operator. Consequently we choose a big initial population (i.e.100) which reduces the risk to obtain a generation composed of only one duplicated element. In order to keep a correct execution time for the test, we choose to do 100 generations.

B. *Results*

Fig.3 presents the recovered and the known architectures of Jigsaw. The comparison of these architectures shows that most of components of the recovered architecture are the same or sub-components of the known ones. Therefore the obtained solution is relevant according to the known architecture of Jigsaw. The recovered architecture has fitness function score of 80.2 %. This shows the correlation between our fitness function and the relevance of the recovered architecture solution compared to the expected one.

We validate the consistency of our approach by measuring the similarity of the recovered architectures of Jigsaw. We use the similarity measure proposed by Mitchell [18]. For each link between two classes it measures the number of solutions (architectures) for which this link is included in a component. To obtain the percentage of inclusion in a component, this value is then divided by the number of compared solutions. The different results are then aggregated at the levels: {[0, 0], (0, 10], (10, 75), [75, 100] }. These levels correspond to a similarity degree zero, low, medium and high.

Table 1 shows degree of similarity obtained over 99 executions of the recovery process. It shows that 81% of classes are in the zero or high categories. This demonstrates that the vast majority of classes are respectively always separated or together. Only 6.4% of class relationships are in the category medium. This shows that class neighborhoods are stable. Changes are due to classes that are on the borders of two components: the content of these classes is highly dependent on two distinct components. Therefore a significant portion of these class methods can be specified as connectors.

Zero (%) $S = 0\%$	Low (%) $0\% < S <= 10\%$	Medium (%) $10\% < S < 75\%$	High (%) $S <= 75\%$
22.9	12.6	6.4	58.1

TAB 1. A measure of degree of similarity of obtained solution on Jigsaw

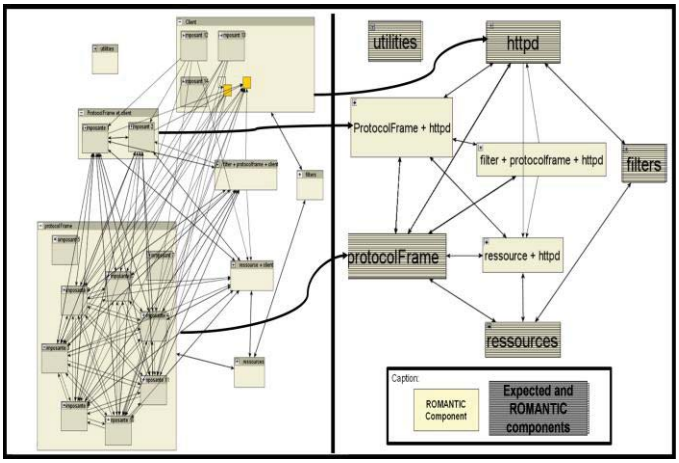

Fig.3 Recovered and known architectures of Jigsaw system

The result of clustering and genetic algorithms is significantly different. This difference is due to the way how the process explores the solution space. On the one hand, clustering explores one solution per iteration. On the other hand, GA explores several solutions (100 in our test) per iteration and the number of genetic operations done by iteration is limited to 2 operations by element of the population. It is clear that GA explores a bigger space than clustering algorithm. GA has a better ratio between the execution time and the quality of the resulting solution.

V. RELATED WORK

Various works are proposed in literature in order to recover architecture from an object-oriented system [15]. We distinguish these works according to two criteria: the input and the technique. Firstly the inputs of the recovery approaches are various. Most often it works from source code representations, but it also considers other kinds of information which for most of them are non-architectural. We can cite, for example, human expertise, which is used in an interactive way in order to guide the process [13], and physical organization, e.g. files, folders and packages [8]. Some works use architectural input. Medvidovic [13] uses styles in Focus in order to infer a conceptual architecture. Finally most works are based on the human expertise: some use the expertise of the architect which uses the tools as an input whereas others use the expertise of the one which proposed this approach. In our approach we use architectural semantic in order to reduce this need of human expertise. Secondly the techniques used to recover architecture

are various and can be classified according to their automation level. Firstly some approaches are quasi manual. For example, Focus [13] proposes a guideline to a hybrid process which regroups classes and maps the extracted entities to conceptual architecture obtained from an architectural style according to the human expertise. Secondly most approaches propose semi-automatic techniques. It automates repetitive aspects of the recovery process but the reverse engineer steers the iterative refinement or abstraction, leading to the identification of architectural elements. Thus ManSART [8] tries to match source code elements on the architectural styles and patterns defined by reverse engineers. Our approach is quasi-automatic too. The main difference with other quasi-automatic approaches is that it refines the commonly used definitions of components into semantic characteristics and refinement models whereas others works use the expertise of the authors in order to define rules driving the process. Some works aim to find the best grouping of elements to subsystems, i.e., the best clusters of an existing software system. Some of these works use a genetic algorithm to compute the best partition [3, 14]. For example, in [14] the problem representation uses chromosomes where each gene represents a class and contains the number of the corresponding cluster. Among these approaches of software clustering, the work of Mancoridis and Mitchell [3] is close to our approach. They introduced the concept of software modularization as a clustering problem for which search is applicable. Their tool Bunch uses a variety of search algorithms. Result is a graph of dependence between modules. This is not an architectural view of the system.

VI. CONCLUSION

We presented in this paper an approach of architecture recovery of object-oriented systems. Architecture recovery is formulated as a search-based problem based on a genetic algorithm (GA). To use genetic algorithms, we adapted our object-component/architecture model to manipulate the architectures (solutions) as chromosomes and group of classes as genes. The properties of this algorithm make it particularly efficient in cases where the computation time is less important compared to the quality of the result. Case studies show that fitness function score is proportional to the relevance of the obtained architectures compared to the expected ones.

As a perspective of this work, we intend to define a method to combine, for a given system, the results obtained by using the genetic implementation of our architecture recovery process with those obtained by the use of the implementation based on simulated annealing [16]. Our goal is to get more relevant architectures in all use cases.

REFERENCES

[1] Kaz1 : Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice,Addison-Wesley, 1998, ISBN 0-201-19930-0.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73

[2] J. M. Bieman and B.-K. Kang, “Cohesion and reuse in an object-oriented system,” in Proc. of the Symp. On Software reusability,SSR ’95, pp. 259–262, 1995.

[3] D. DOVAL, S. MANCORIDIS et B. S. MITCHELL. Automatic clustering of software systems using a genetic algorithm. In STEP ’99 : Proceedings of the Software Technology and Engineering Practice, page 73, Washington, DC, USA, 1999. IEEE Computer Society.

[4] Emanuel FALKENAUER. Genetic Algorithms and Grouping Problems. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[5] Garlan, D., Shaw, M.: "An Introduction to Software Architecture," In V. Ambriola and G. Tortora (ed.), Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, Singapore, pp. 1-39, 1993.

[6] Garlan. Software architecture: a roadmap. In ICSE – Future of SE Track, pp. 91–101, 2000.

[7] Holland, J. Adaptation in Natural and Artificial Systems, Ann Arbor, MI: The University of Michigan Press, 1975.

[8] D. R. Harris, H. B. Reubenstein, and A. S. Yeh, “Reverse engineering to the architectural level,” in Proc. of ICSE, pp. 186–195, ACM, Inc., 1995.

[9] G. Heinemann and W. Councill, Component-based software engineering. Addison-Wesley, 2001.

[10] ISO/IEC-9126-1 in Software engineering - Product quality - Part 1: Quality Model, ISO-IEC, 2001.

[11] I. Jacobson, M. Griss, and P. Jonsson, Software Reuse. Addison Wesley/ACM Press, 1997.GP

[12] C. Luer and A. van der Hoek, “Composition environments for deployable software components,” tech. rep., 2002.

[13] N. Medvidovic and V. Jakobac, “Using software evolution to focus architectural recovery,” Automated Software Engineering, vol. 13, pp. 225–256, 2006.

[14] Olaf SENG, Markus BAUER, Matthias BIEHL et Gert PACHE. Search-based improvement of subsystem decompositions. In GECCO ’05 : Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1045–1051, New York, NY, USA, 2005. ACM.

[15] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus, “Towards a process-oriented software architecture reconstruction taxonomy,” in Proc. of the CSMR, pp. 137–148, 2007.

[16] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, Dalila Tamzalit: Search-Based Extraction of Component-Based Architecture from Object-Oriented Systems. ECSA 2008: 322-325

[17] C. Szyperski, Component Software. ISBN: 0-201-17888-5, Addison-Wesley, 1998.

[18] Brian S. Mitchell et Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. Soft Comput., 12(1) :77–93, 2008.