ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)

Vol. 2, Special Issue 10, March 2016

# Quality Driven Approach for the Discovery of Software Architectures

Dharani Angamuthu[1] , G.Tamil Mani[2] , N.RajKumar[3]

M.tech,Department of Software Engineering,Veltech Dr.RR & Dr.SR Technical University,Avadi, Chennai.[1]
Asst.Prof,Department of CSE ,Veltech Dr.RR & Dr.SR Technical University,Avadi, Chennai.[2]
Asst.Prof,Department of CSE ,Veltech Dr.RR & Dr.SR Technical University,Avadi, Chennai.[3]

## 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 arte- facts 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 uncon- trolled 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 dis- covery of the underlying architecture of software systems. Important efforts have been directed towards the creation of a generic and human-oriented process. Hence, the selec- tion 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.

## 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. 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 prac- tice when dealing with complex system, resulting in more comprehensible software and making its development

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 10, March 2016

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. 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 the- se cases, engineers must expend their time and effort, with their own experience as their only guarantee, in the manual dis- covery of these functional blocks.

Architectural optimisation methods in the field of software engineering (SE) have often proposed guidelines and recom- mendations to modellers for the identification and improvement of software architectures. 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, 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 mod- els 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. Since the appearance of SBSE, evolutionary computation (EC) has emerged as the most applied metaheuristic, demonstrating that it constitutes an interesting and complementary way to help software engineers in the improvement of their object-oriented class designs or user interfaces. 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 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 inter- esting starting point for architecture

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 10, March 2016

discovery. These diagrams offer more specific analysis information than source code, and they use modelling languages like UML 2 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 func- tionalities 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 com- ponent 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, generalisations repre- sent 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 ser- vice 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 com- ponents. They will be identified after the discovery of the interfaces created between components.

4.Proposed model for architecture

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.

**Encoding of solutions**

Selecting the most appropriate problem encoding is a key step in any search

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 10, March 2016

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 to be efficient represen- tations, 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 pro- cess 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 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 solu- tions 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 rela- tionships 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 con- nectors. 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.

**Initial population**

From the problem description (see Section 3), it can be noted that the search space is constituted by all possible combi- nations 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 guar- antees that no empty components will be generated. Then, each class is assigned to one component, assuring that each com- ponent 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 pro- gressively removed along the generations.

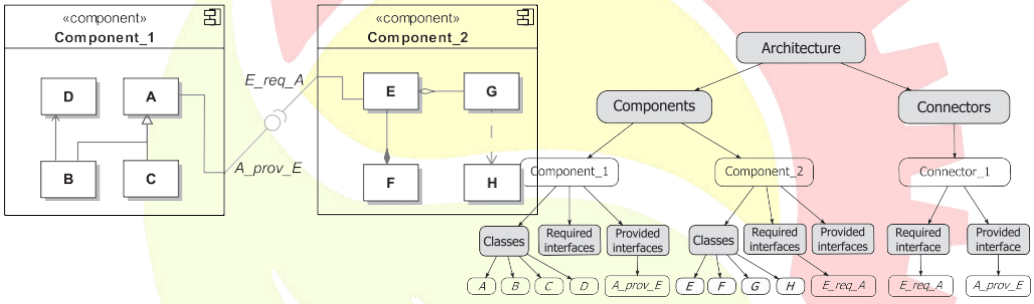
ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)

Vol. 2, Special Issue 10, March 2016

**Ranking fitness function**

Diverse functional or non-functional properties can be considered depending on the under- lying 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 stan- dardisation between metrics, which could result in an artificial procedure when they are not defined in an appropriate range




(a) Phenotype (b) Genotype

Fig. 1. The phenotype/genotype mapping process.

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) serves to determine a trade-off between cohesion and cou- pling. For each component i; ICDi is calculated as the ratio between internal and external relations, which has to be max-imised. CIin is the number of interactions inside the component, i.e. the relationships between classes allocated in the same component. CIout represents the number of relationships between component i and the others, i.e. the number of can- didate 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. ICDi varies in [0, 1]. Finally, the ICD of the overall architecture (individual) is calculated as the average of every ICDi .

Finally, the groups/components ratio (GCR) metric, presented in Eq. (3), is inspired by the component packing density (CPD) metric defined in CPD calculates the ratio between the number of constituents, e.g. operations, classes

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 10, March 2016

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 a unique group of strongly interre- lated classes.

**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 applicable, as they would cause probably the replication of classes after the combination of components from different indivduals.

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 as a probability threshold of selection of each com- ponent 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 (maxcgroups ) 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.

**Proposed Algorithm**

The proposed algorithm 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 mini- mum 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 order to assign rankings in a proper way. Finally, the replacement strategy (line11) chooses those individuals that will survive, assigning them to the next population. When the maximum number of gen- erations (maxGen) is reached, the evolution ends and the best individual in the current population is returned as the candi- date architecture

**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 rela- tionships 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 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 com- prise 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.

**References**

[1] D. Belanger, et al., Architecture Styles and Services: An Experiment Involving the Signal Operations Platforms-Provisioning Operations System, AT&T Technical Journal, Jan/Feb 1996, pp. 54-63.

[2] S. Bot, C.-H. Lung, and M. Farrell, A Stakeholder- Centric Software Architecture Analysis Approach, in Proc. ISAW 2 - Int'l Software Architecture Workshop, 1996.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996.

[4] C. Gacek, A. Abd-Allah, B. Clark, B. Boehm. On the Definition of Software System Architecture, in Proc. of ICSE 17 Software Architecture Workshop, April 1995.

[5] D. Garlan and M. Shaw. An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, vol. 1, 1993.

[6] R. Kazman, G. Abowd, L. Bass, M. Webb, SAAM: A Method for Analyzing the Properties of Software Architectures, in Proceedings of the 16th

International Conference on Software Engineering, May 1994, pp. 81-90.

[7] R. Kazman, G. Abowd, L. Bass, P. Clements.Scenario-Based Analysis of Software Architecture, IEEE Software, Nov 1996. - 11 -

[8] P. B. Kruchten. The 4+1 View Model of Architecture, IEEE Software, Nov 1995, pp. 42-50.

[9] C. Krueger, Software Reuse, ACM Computing Surveys, 24(2), 1992, pp. 131-183.

[10] C.-H. Lung and J. Urban. An Expanded View of Domain Modeling for Software Analogy. Proc. 19th AnnuaInt'l Comp Software & ApplicationsConf - COMPSAC, pp.77-82, 1995.

[11] C.-H. Lung, Empirical Experiences in Analyzing Software Architecture Sensitivity, in Proc. of COMPSAC, pp. 164-165, 1997.

[12] C.-H. Lung and K. Kalaichelvan, Metrics for Software Architecture Robustness Analysis, submitted for publication.

[13] S. Wage, Preventive Software Maintenance: Prevention is Better Than Cure, Tech. Report, School of InfO Science and Technology, LiverpoolPolytechnic, 1988.