Co-evolving Pattern Synthesis and Class Responsibility Assignment in Architectural Synthesis*

Yongrui Xu and Peng Liang**

State Key Lab of Software Engineering
School of Computer, Wuhan University, Wuhan, China
{xuyongrui,liangp}@whu.edu.cn

Abstract. Architectural synthesis (AS) activity plays a key role in architecture design as it essentially links the problem to the solution space. To reuse successful design experience, architects may use architectural patterns in AS to generate candidate solutions. In a pattern-based AS, there are two challenges: one is class responsibility assignment (CRA) when using specific patterns and the other is pattern synthesis which attempts to avoid the pattern constraint violations. In this paper, we propose a cooperative coevolution approach to assign class responsibility and synthesize pattern automatically in a pattern-based AS. We formally translate the problem of the automated pattern-based AS into a multi-objective optimization problem, and describe the approach in detail.

Keywords: Automated architectural synthesis, class responsibility assignment, architectural patterns, cooperative coevolution.

1 Introduction

During the architecting process, architects perform various activities for different purposes towards the construction of the architecture of a software-intensive system. Hofmeister *et al.* defined a general model of architecture design including three activities, i.e., *architectural analysis*, *architectural synthesis*, and *architectural evaluation* [1]. Architectural synthesis (AS) activity proposes a collection of candidate architecture solutions, which are composed of a set of architectural elements (e.g., classes, components, and connectors) to address architecturally significant requirements (ASRs) identified during architectural analysis. AS is the core activity of architecture design as it essentially links the problem to the solution space of architecture design. However, how to propose architecture solutions to a set of ASRs largely depends on the experience of architects in traditional AS.

To reduce the complexity and reuse successful design experience when designing software architectures, architects rely on a set of idiomatic architectural patterns, which are packages of architectural design decisions and are identified and used repeatedly in practice [2], such as MVC, pipe and filter, and layer patterns. Using architectural patterns for architectural synthesis gets lots of benefits, and the architecture of

* This work is partially sponsored by the NSFC under Grant No. 61170025.
** Corresponding author.

P. Avgeriou and U. Zdun (Eds.): ECSA 2014, LNCS 8627, pp. 74–81, 2014.

large and complex systems is increasingly designed by composing architectural patterns [2]. However, existing research has observed that the resulting architecture of a system does not always conform to the initial patterns employed which guide the design at the beginning [3]. It is mainly due to the reasons that (1) existing work mostly focuses on pattern recommendation and selection, but pays less attention to the conceptual gap between the abstract elements and the implementation units in the employed patterns; (2) each pattern has a set of design constraints when using it, and architects may use the pattern being unaware of the constraints or misinterpreting the constraints due to lack of experience (especially for novice architects). If the pattern constraints are not satisfied, architects may have to redesign the architecture in order to avoid negative impact to the quality of the system. In summary, most existing work focuses on "architectural patterns recommendation and selection" instead of "architectural patterns implementation" which is part of architectural synthesis [1], and they did not address how to arrange structural elements (e.g., components and connectors) elegantly in a pattern to avoid the violations to the pattern constraints.

On the other hand, assigning responsibilities to classes is another vital task in object-oriented architectural synthesis [4], in which responsibilities are represented in terms of methods and attributes. Class responsibility assignment (CRA) has a great impact on the overall design of the application [4], and many methods were developed to help recognize the responsibilities of a system and assign them to classes [4] [5]. But one of deficiencies of these works is that they addressed the CRA problem in isolation. In practical software design, architects should not only consider CRA issues, but also think about the quality impact of architecture (e.g., which patterns can be used to satisfy given quality attributes). In most situations, the system quality attributes have great influences on CRA (e.g., some responsibilities may be replicated in both client- and server-side when using Client-Server pattern in order to improve the performance or reliability of a system), and CRA should be considered together with the quality aspects of the system.

To this end, we propose a cooperative coevolution approach that aims at synthesizing pattern-based architecture solutions automatically. It is composed of two processes: *responsibility synthesis* (RS) and *pattern synthesis* (PS). RS process addresses the CRA problem while PS process focuses on pattern implementation at architectural level. This approach tries to avoid the violations to the pattern constraints while considering the responsibility assignment to architecture elements (e.g., classes) in the resulting architecture solutions. In our approach, we use meta-heuristic search techniques (e.g., NSGA-II) to explore (i.e., to visit entirely new regions of a search space) and exploit (i.e., to visit those regions that are explored within the neighborhood of previous visited points) [6] pattern-based architecture design space automatically when the populations of RS and PS cooperatively coevolve. On one hand, we use pattern metrics that were proposed in our previous work [7] to evaluate the pattern constraint violations; on the other hand, we use CK object-oriented metrics to evaluate the quality of CRA [8]. We demonstrate how the proposed approach can help architects arrange architectural elements with minimum constraint violations to implement architectural patterns in specific design context. The contributions of this work are: (1) the first attempt to consider responsibility synthesis (i.e., CRA) and pattern synthesis simultaneously in pattern-based architectural synthesis; (2) translating the pattern-based architectural synthesis to a cooperative coevolution problem, which can be automated.

2 Background

For pattern synthesis (PS) process, our work is rooted in the concept of pattern metrics [7], which is based on pattern constraints. For responsibility synthesis (RS) process, we employ the responsibility collecting techniques. We describe these concepts and techniques as well as their relationships to our proposed approach below.

Architectural Pattern Metrics are used to measure the pattern constraint violations of candidate architecture solutions generated in automated architectural synthesis in PS. We introduce the pattern metrics definition process and describe MVC pattern metrics as an example in [7], which helps architects to translate the pattern constraints of an architectural pattern to its pattern metrics.

Responsibility Collecting Techniques help to collect responsibilities of a system and the dependencies between the responsibilities. Some of these techniques directly identify classes from requirements specifications that can be used to further collect responsibilities of a system, such as common class pattern, and class responsibility collaboration (CRC) card [5]. In this paper, we focus on the part of automated RS, and we assume that the responsibilities of a system and their dependencies have been collected and made available using abovementioned techniques.

Fig. 1 shows the relationship between our proposed cooperative coevolution approach for architectural synthesis (inside the rectangle) and the concepts and techniques mentioned above. On one hand, architects use a definition process [7] to define pattern metrics from constraints of the patterns they plan to use; on the other hand, using certain responsibility collecting technique, architects acquire responsibilities from requirement specifications. Pattern metrics and responsibilities are used as inputs for pattern synthesis and responsibility synthesis respectively in our approach to finally generate architecture solutions. We will present the formal definition of the automated pattern-based AS problem in the next section, which is further translated into a cooperative coevolution problem detailed in Section 4.

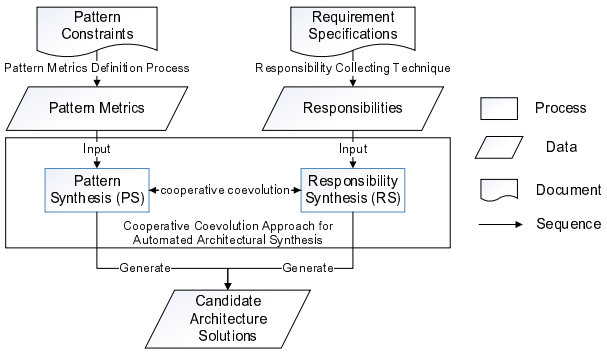

Fig. 1. The relationship between our cooperative coevolution approach and related techniques

3 The Problem

A pattern provides a generic solution for a recurring problem: a solution that can be implemented in many ways without necessarily being “twice the same”, and there is

no configurable generic implementations for patterns that cover their whole design space [9]. However, every pattern has its invariable roles, such as *model*, *view*, and *controller* in MVC pattern, and we define these invariable roles as the set of pattern roles for each pattern, e.g., {*model, view, controller*} for MVC pattern.

Let R represent a set of responsibilities which are derived from requirement specifications by responsibility collecting techniques, and every element in R represents which class one specific responsibility belongs to. Hence the number of elements in R equals the number of responsibilities, and let the number as *n*. In addition, let P represent the pattern role that each responsibility plays for the chosen pattern plus the connector type of all the relations among the responsibilities during pattern synthesis. Let the number of relations between responsibilities as *m,* hence the number of elements in P equals to the number of responsibilities in R (i.e., *n*) plus the number of relations (i.e., *m*). The first *n* elements in P represent the pattern role that each responsibility plays in a specific pattern, while the last *m* elements indicate the connector type (e.g., procedure call, event, and data access) [10] for each relation.

Fig. 2 depicts an example of R and P set that has two classes (Class 1 and Class 2) with five responsibilities: three operations and two attributes, and there is a relation between Operation 2 (O2) and Operation 3 (O3) that O3 *depends on* O2. This example uses Layer pattern and the two classes are located in different layers. The number of elements in R (i.e., number of responsibilities) is five, and each element indicates which class the responsibility belongs to. For instance, O1 belongs to Class 1, while Attribute 2 (A2) belongs to Class 2. Hence the value of R set is {C1, C1, C2, C1, C2}. In Layer pattern, layers constitute the set of pattern roles. The first five elements in P represent the layers that corresponding responsibilities belongs to, for example O1 belongs to Layer 0 (L0), while O3 belongs to Layer 1 (L1). In addition, as there is one dependency between responsibilities O2 and O3, P set has one extra element which describes the connector type of this relation, remote procedure call (RPC).

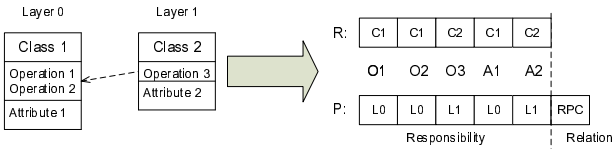

Fig. 2. An example of R and P set in Layer pattern

Formally speaking, the automated pattern-based AS problem consists in establishing an automated search for two optimal sets R and P following the definitions below:

- *The individuals from the responsibility population (IndR)*: IndR is an individual (chromosome) from responsibility population expressed as R set, and the value of each element v_i in R set represents one specific class. If two elements have the same value, it means the responsibilities these two elements represent belong to the same class. Therefore, v_i represents one feasible design decision of responsibility R_i, which assigns this responsibility to one class, in the whole design space.
- *The individual from pattern population (IndP)*: IndP is an individual (chromosome) from pattern population expressed as P set. P has two parts: responsibility and relation parts. The value of each element v_i in responsibility part represents the type of

pattern role for the corresponding responsibility. If two elements in the responsibility part have the same value, it means these two responsibilities play the same pattern role for a given pattern (e.g., they are in the same layer in Layer pattern). For relation part, the value of each element v_i represents the connector type (e.g., RPC, event) of this relation. Therefore, the value v_i of each element in P also represents one feasible design decision of pattern synthesis in the whole design space.

An optimal set R means this individual gets the highest score for evaluation metrics of CRA in responsibility population, while an optimal set P means this individual has the least pattern constraint violations in pattern population. For automated pattern-based AS problem, our objective is to acquire the solutions which not only achieve a high cohesion, low coupling and complexity design for CRA, but also have minimal pattern constraint violations. Therefore, the problem can be featured as a multi-objective optimization problem, and the responsibility and pattern population have a cooperative coevolution relationship for acquiring the final optimal architecture solutions.

4 Cooperative Coevolution Approach

The proposed approach coevolves two populations: responsibility and pattern populations. Fig. 3 illustrates the cooperative coevolution procedure with following steps:

1. *Population initialization*: The two populations are randomly generated, taking into account the set of responsibilities, pattern roles of these responsibilities, and all the relations among these responsibilities. Each population has a fixed size to form individuals (i.e., a series of IndR and IndP), which is described in Section 3.
2. *Calculating the fitness of population*:
   (a) *Best individual selection for responsibility population (BestIndR)*: in the first generation, an individual from responsibility population is randomly selected as the best individual. From the second generation, BestIndR is the individual with the best score for the fitness function defined in Section 4.2 for CRA problem.
   (b) *Best individual selection for pattern population (BestIndP)*: the best individual from pattern population is selected. We define a fitness function in Section 4.2 to ensure BestIndP has the least pattern constraint violations.
3. *Applying genetic operators*. Genetic operators include selection, crossover, and mutation operators [11]. In this step, for each population, a selection operator is used to select parents from all individuals for the crossover operation which generates sons using a crossover operator. Then mutation is performed using a mutation operator for each individual to produce the next generation for the two populations.
4. *Stopping condition satisfied*. If the limit of generations established by the input parameters of the meta-heuristic algorithm is reached, the execution of the cooperative coevolution procedure is stopped. As there are two objectives to be optimized (i.e., maximized cohesion metrics, minimized coupling and complexity metrics for CRA; and minimized pattern constraint violations for pattern implementation at architectural level), our approach returns a Pareto front (i.e., a set of solutions represent the best possible trade-offs among the two objectives) constructed by BestIndR and BestIndP when the stopping condition is satisfied.

It is expected that using this cooperative coevolution approach architects can acquire optimized solutions with better CRA and minimized pattern constraint violations.

In the following sub-sections, we describe representation of individuals for each population, and the fitness functions in detail.

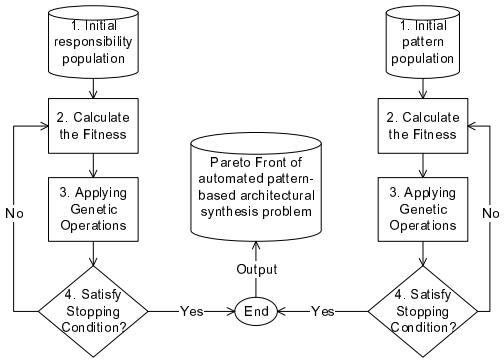

Fig. 3. Cooperative coevolution procedure of our approach

4.1 Representation of Individual

As illustrated in Fig. 2, there are two kinds of individuals, one is R set from responsibility population, and the other is P set from pattern population. We describe them in the next two sub-sections.

a) Individuals in Responsibility Population

To encode the class responsibility assignment of a system in a chromosome (i.e., R set), we define the concept *centroid responsibility* (CR) similar to the concept *centroid use case* in [12]. In our approach, for each class, a CR is considered as a representative of other responsibilities belonging to that class, and R set is represented as a binary string of length n, where n is the total number of responsibilities. If the value of an element is "1", its corresponding responsibility is a CR; and otherwise when the value of the element is "0", then its corresponding responsibility is a non-CR. Therefore, the number of "1" in the binary string of R set shows the number of classes. For instance, one possible binary string in Fig. 2 is [10100], in which O1, O3 are CRs of Class 1 and Class 2. For those responsibilities that are not CRs, they are assigned to the most similar class. Here, the similarity between a non-CR and a class is equivalent to the similarity between the non-CR and the CR of that class. For example, in Fig. 2, if O2 is more similar to O1 compared with O3, then O2 is assigned to Class 1.

The core part of the encoding scheme for responsibility population is to define the similarity function to calculate the similarity between any two responsibilities. When using responsibility collecting techniques to collect the responsibilities of a system and the dependencies between responsibilities, we can acquire many types of dependencies between two responsibilities, such as Direct Method-Method dependency (DMM), Direct Method-Attribute dependency (DMA), and Direct Responsibility-Responsibility Semantic dependency (DRRS) [5]. For each type of dependency, we use a binary matrix to show the presence or absence of dependencies between responsibilities, and we use Jaccard binary similarity measurement [5] to calculate the similarity between two vectors in binary matrix.

As many types of dependencies (e.g., DMM) exist between two responsibilities, we assign each dependency type a specific weight to calculate the similarity between two responsibilities (e.g., R_i and R_j) using Formula (1), in which w_k denotes the weight of certain dependency type *dep*, $Sim_{dep}(R_i, R_j)$ calculates the similarity between two responsibilities with the binary matrix of dependency type *dep*, and *n* represents the number of dependency types:

$$\mathrm{Sim}(R_i, R_j) = \frac{\sum_{k=1}^{n} w_k Sim_{dep}(R_i, R_j)}{\sum_{k=1}^{n} w_k} \quad (1)$$

b) Individuals in Pattern Population

To encode the pattern synthesis of a system in a chromosome (i.e., P set), we use integer number for both responsibility and relation part of P set. For responsibility part, each integer number represents the pattern role for the corresponding responsibility. For Layer pattern, the integer number represents the specific layer directly. For instance, in Fig. 2, the responsibility part of P is [0,0,1,0,1], in which only O3 and A2 are in Layer 1, and other responsibilities belong to Layer 0. The maximum value of the integer number equals the total number of responsibilities, which means in an extreme condition, every responsibility belongs to a separate layer. For other patterns (e.g., MVC), the specific value of the integer number represents a specific role in that pattern (e.g., in MVC, 0 for model, 1 for view, and 2 for controller). For relation part, each integer number represents the connector type for corresponding relation. For example, we can use 0 to 6 to represent the 7 different connector types in [10].

4.2 Fitness Function

Many approaches have been proposed to address the CRA problem, and most of them use object-oriented metrics which are based on the CK metrics to define their fitness function. We also use these metrics. Formula (2) shows the fitness function for CRA problem, which is used to maximize the overall cohesion and minimize the overall coupling and complexity of software. The details about the cohesion, coupling, and complexity metrics are introduced respectively in [5].

$$\text{Fitness Score}_{RS} = \text{SoftwareCohesion} - \text{SoftwareCoupling} - \text{SoftwareComplexity} \quad (2)$$

For pattern synthesis, our objective is to minimize the violations of pattern constraints. In our recent work [7], we proposed a definition process for pattern metrics, which is composed of three steps (identify the roles of a pattern, the relations within a pattern, and the domain related metrics). The definition process takes pattern constraints as input, and output a series of pattern metrics. Each metric measures the number of violations for a specific type of pattern constraint. It is worth noting that every pattern has its own metrics, and different metrics have different weights [7]. Hence we can quantify the pattern constraint violations for a specific pattern using Formula (3), in which w_k represents the weight of a specific metric $metric_k$ of the chosen pattern, and *n* is the number of pattern metrics for that pattern.

$$\text{Fitness Score}_{PS} = \frac{\sum_{k=1}^{n} w_k metric_k}{\sum_{k=1}^{n} w_k} \quad (3)$$

Architects can acquire architecture solutions which consider both CRA and pattern synthesis. In each generation, the Pareto front can be established and updated with these solutions automatically. When the cooperative coevolution procedure terminates,

any solution of the Pareto front may be considered as optimal, since it means that no further improvement can be made for an objective without degrading another.

5 Conclusions

Architectural synthesis essentially links the problem to the solution space and plays a key role in architecting process from requirements to initial architecture design. However, due to its complexity, this architecting activity heavily depends on the experience of architects. In this paper, we propose a cooperative coevolution approach to assign class responsibility and synthesize pattern automatically in the pattern-based AS. We formally translate the problem of the automated pattern-based AS into a multi-objective optimization problem. One objective is to maximize the cohesion and minimize the coupling and complexity of solutions for class responsibility assignment, and the other objective is to minimize the pattern constraint violations for the chosen pattern. We then describe the cooperative coevolution approach, including the representation of the problem, and the fitness function to evaluate the solutions. In the next step, we plan to conduct controlled experiments that compare the quality of architecture design between the generated pattern-based AS solutions using our approach and the solutions by human architects based on the same design problems.

References

1. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: A general model of software architecture design derived from five industrial approaches. J. Syst. Softw. 80(1), 106–126 (2007)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley Professional (2012)
3. Belle, A., El Boussaidi, G., Desrosiers, C., Mili, H.: The layered architecture revisited: Is it an optimization problem? In: SEKE, pp. 344–349 (2013)
4. Bowman, M., Briand, L.C., Labiche, Y.: Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Trans. Softw. Eng. 36(6), 817–837 (2010)
5. Masoud, H., Jalili, S.: A clustering-based model for class responsibility assignment problem in object-oriented analysis. J. Syst. Softw. (2014)
6. Črepinšek, M., Liu, S., Mernik, M.: Exploration and exploitation in evolutionary algorithms: A survey. ACM Comput. Surv. 45(3), 1–33 (2013)
7. Xu, Y., Liang, P.: Automated software architectural synthesis using patterns: A cooperative coevolution approach. In: SEKE, pp. 174–180 (2014)
8. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20(6), 476–493 (1994)
9. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, 1st edn. Wiley (2007)
10. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: ICSE, pp. 178–187 (2000)
11. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based Software Engineering: Techniques, Taxonomy, Tutorial. In: Meyer, B., Nordio, M. (eds.) Empirical Software Engineering and Verification. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012)
12. Hasheminejad, S.M.H., Jalili, S.: SCI-GA: Software component identification using genetic algorithm. J. Object Technol. 12(2), 1–34 (2013)