A Cooperative Coevolution Approach to Automate Pattern-based Software Architectural Synthesis

Yongrui Xu† and Peng Liang‡

State Key Lab of Software Engineering, School of Computer Science
Wuhan University, 430072 Wuhan, China
†xuyongru@whu.edu.cn
‡liangp@whu.edu.cn

To reuse successful experience in software architecture design, architects use architectural patterns as reusable architectural knowledge for architectural synthesis. However, it has been observed that the resulting architecture does not always conform to the initial architectural patterns employed. Architectural synthesis using architectural patterns is also recognized as a challenging task, especially for novice architects due to lack of experience. In this paper, we propose a cooperative coevolution approach to automate architectural synthesis using architectural patterns. We first analyze several common architectural patterns and the constraints when using them. We then extend existing architectural synthesis activity with patterns based on the results of this analysis. Finally, we map the extended architectural synthesis to a cooperative coevolution model, which can optimize the resulting architectural solutions and avoid the violations to the pattern constraints automatically. We evaluate the proposed approach through a case study: architecture design of a cinema booking system. The results show that the proposed approach can generate architectural solutions which are closer to the expert design.

Keywords: Automated architectural synthesis; architectural patterns; cooperative coevolution.

1. Introduction

Large software systems are composed of lots of components, and the interactions between them are very complex. To reduce the complexity 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 [1], such as MVC, pipe and filter, blackboard, and layer patterns.
Using architectural patterns for architectural synthesis gets lots of benefits, and the software architecture of large systems is increasingly designed by composing architectural patterns [1, 2]. Therefore, many existing work focuses on how to select appropriate patterns from a pattern repository in specific design context by considering quality requirements in architectural synthesis [3, 4]. However, many researchers observed that the resulting architecture of a system does not always conform to the initial patterns employed which guide the design at the beginning [5]. It is mainly due to the reasons that (1) existing work 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 [6], and they did not address how to arrange components and connectors elegantly in a pattern to avoid the violations to the pattern constraints.

On the other hand, architectural synthesis heavily depends on the experience of architects, especially when the design space is increased exponentially with increasing system scale. Many approaches have been proposed to support exploring and exploiting architecture design space automatically [7]. Most of them use the Search-Based Software Engineering (SBSE) or Search-Based Software Design (SBSD) techniques [8]. However, as noted in [8], although many aspects of Software Engineering problems lend themselves to a coevolution model of optimization, surprisingly, there is little work that has been done on using this coevolution model to address Software Engineering problems. Automated architectural synthesis is one of these problems.

To this end, we propose a cooperative coevolution approach that aims at synthesizing pattern-based architectural solutions automatically. This approach tries to avoid the violations to the pattern constraints while considering the responsibility assignment in the resulting architectural solutions. In our approach, we first extend the classical architectural synthesis activity, which essentially links the problem space to the solution space of architecture design, with two parts: manual steps by architects and automated steps by tools. We then investigate on the constraints of existing architectural patterns, and define pattern metrics based on these constraints to construct the fitness function for automated architectural synthesis by tools. As mentioned before, when the candidate architectural solutions are synthesized, there

---

For readability and clarity, we list all the abbreviations used in this paper in Appendix A for reference.
are two main objectives (i.e. avoid violations to pattern constraints and assign responsibility to architectural elements). Each objective may correspond to one population (a set of solutions). When the two objectives are not strongly correlated, their two populations can be coevolved to work better together, which offers great potential for architectural synthesis [8]. Hence the problem of automated architectural synthesis using patterns is translated into a cooperative coevolution optimization problem. We demonstrate how the proposed approach can help architects arrange components and connectors with minimum constraint violations to implement architectural patterns in a specific design context.

A detailed description and explanation of automated responsibility synthesis and pattern synthesis in the proposed approach, and an evaluation of the proposed approach through a case study, are the major extensions to our previous work reported in [9].

The rest of this paper is organized as follows. The automated architectural synthesis approach is described in Sec. 2 in detail. A case study was conducted to evaluate our proposed approach in Sec. 3. Related work is discussed in Sec. 4 and we conclude this work and outline future directions in Sec. 5.

2. Approach

In this section, we first analyze several common architectural patterns and the constraints when using them. We then extend classical architectural synthesis activity presented in [10] to a pattern-based architectural synthesis. We also describe the definition process for pattern metrics in detail in order to improve the applicability of our approach when architects use their own patterns. With the definition process, architects can define the pattern metrics from pattern constraints. Finally, we map the extended architectural synthesis to a cooperative coevolution optimization model, which tries to avoid the violations to the pattern constraints while considering the responsibility assignment of architectural elements automatically. The proposed approach makes the resulting architecture conform to the initial patterns employed at the beginning, while allocating responsibility for architectural elements in architectural synthesis.

2.1. Extended architectural synthesis

General architecting process is composed of three activities: architectural analysis, synthesis, and evaluation [10], in which classical architectural synthesis (AS) activity proposes a collection of candidate architectural solutions (e.g. architectural patterns) to address the architecturally significant requirements (ASRs) identified during architectural analysis. Architectural synthesis essentially links the problem to the solution space of architecture design. However, how to propose architectural solutions to a set of ASRs largely depends on the experience of architects in classical AS, and to make matters worse, there are no available guidelines and steps to perform this
activity for architects. In order to reduce the probability of making mistakes in AS (e.g. due to the lack of experience), we introduce the extended AS for architects.

As shown in Fig. 1, the steps in the extended AS are divided into two parts: manual steps by architects and automated steps by tools. The output of architectural analysis (AA) is a set of function requirements and ASRs, and architects need to choose or create appropriate patterns to address them in architectural synthesis. Distinguished from other ways of using architectural patterns, our approach focuses on pattern constraints in pattern implementation. Architects can use pattern constraints which have been defined for common patterns or define the constraints for their own specific patterns. The next step of our approach is to define pattern metrics from pattern constraints by architects. In this paper, we provide the pattern metrics for an example architectural pattern — MVC, and we describe the definition process of pattern metrics from pattern constraints for architects who want to use other patterns. As shown in Fig. 1, when the pattern metrics are defined, the automated part of the extended AS starts. The automated part is composed of two sub-processes: responsibility synthesis (RS) and pattern synthesis (PS), in which responsibility means functional requirements that should be implemented. In RS, the functional requirements are used as input, and this sub-process only considers the responsibility of the system by focusing on what the system should do. Unlike RS, PS sub-process is independent of business context and it only focuses on pattern implementation. It takes pattern metrics as input, and uses them to construct a fitness function which tries to minimize violations to the pattern constraints. PS and RS are automated and executed simultaneously, and they form the cooperative coevolution optimization model. The benefit of this partition between RS and PS is that PS is independent of business context, while RS focuses on the functional aspect of a system.

Fig. 1. Extended architectural synthesis activity.
Hence, in our approach, we follow the design concept of “divide-and-conquer” for architectural synthesis to implement the principle of “separation of concerns” [2].

2.2. Constraints and metrics of architectural patterns

An architectural pattern is composed of a triple \(\{\text{context}, \text{problem}, \text{solution}\}\). In [1], Bass et al. further refine the solution of a pattern to five parts including overview, elements, relations, constraints, and weaknesses. The constraints of a pattern play an important role in limiting the possible pattern implementations. However, in practice, architects may choose to violate the constraints of the selected pattern in order to make a tradeoff among different factors, such as system quality attributes, implementation cost. This is the major reason why the resulting architecture of a system does not always conform to the initial patterns which guide the architecture design at the beginning [5].

The importance of design constraints in design has been recognized in [11] and design constraints are represented as a first-class entity in architecture design reasoning [12]. Hence we represent pattern constraints, a type of design constraint, as a first-class entity when using patterns in architectural synthesis, and we need to answer “how to represent the pattern constraints when using architectural patterns?” We choose several widely-used architectural patterns, and analyze their constraints based on the pattern descriptions in [1, 2]. We summarize the constraints of several common architectural patterns in Table 1.

In order to evaluate the quality of candidate architectural solutions generated in automated architectural synthesis, we need to define pattern metrics, which are used to measure the quality attributes of solutions, from pattern constraints. Due to space limitations, we only describe the pattern metrics from the pattern constraints for one pattern: MVC, as an example, since this pattern is one of the most well-known patterns in architecture design. We will further introduce the definition process for pattern metrics in the next subsection.

In an interactive application, it is important to keep modifications to the user interface separate from the rest of the system. To address this design issue, MVC separates application’s functionality into three types of components: model, view, and controller, which are essential for an application of using MVC pattern. Hence, we define \(\text{LegalMVC}\) metric to judge whether the solution satisfies Constraint (1) of MVC pattern (as shown in Table 1). For Constraint (2), we define \(\text{ControllerUse}(m)\), in which \(m\) is a given model, to count the number of relations that an element in model \(m\) depends on the element in the controller element. For Constraint (3), we define \(\text{CorrespondingViewUse}(m)\) to count the number of change notifications from \(m\) to all the views that correspond to it. Note that, for Constraint (2), if model elements depend on controller elements, it has an apparently negative impact on modifiability, portability, and reuse of architecture elements. Hence for some metrics like \(\text{ControllerUse}(m)\), which have a great impact on quality attributes, we give them a high weight generally (i.e. the higher weight, the more the influence on the
calculation of constraint violations). In Eq. (1), we define model cost (MC) for a model element \( i \) to calculate pattern constraint violation cost for element \( i \), where \( i \) belongs to model type in MVC. The \( w_1 \) and \( w_2 \) are the value of weight. Since ControllerUse has a higher weight than CorrespondingViewUse, we set \( w_1 \ll w_2 \). From Eq. (1), the model cost for a given model element depends on the number of relations between this model element and its related view and controller elements.

\[
MC(i) = w_1 \text{CorrespondingViewUse}(i) + w_2 \text{ControllerUse}(i), \quad \text{where} \quad w_1 \ll w_2 \quad (1)
\]

Similar to the pattern metrics definition of model element, for a given view \( v \), there are also two metrics: for Constraint (4), we define CorrespondingModelUse(\( v \)), in which \( v \) is a given view, to count the number of all the state-query relations from \( v \) to its corresponding models, while we define ControllerUse(\( v \)) to count the number of relations from \( v \) to its controllers. For Constraint (5), we define

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Pattern constraints</th>
</tr>
</thead>
</table>
| Model-View-Controller (MVC)      | 1. There should be at least one instance of each pattern elements, i.e. model, view, and controller.  
|                                  | 2. The model element should not interact directly with the controller.                
|                                  | 3. There should be at least one view corresponding to an instance of model.          
|                                  | 4. Every view should be associated with at least one model.                           
|                                  | 5. There is a one-to-one relationship between views and controllers.                 |
| Blackboard                       | 1. There should be at least one instance of each pattern elements, i.e. blackboard, knowledge source, and controller.  
|                                  | 2. Control data and partial solutions are stored in blackboard, and knowledge sources access data through blackboard’s interfaces.  
|                                  | 3. Every knowledge source should not depend on other knowledge sources.              
|                                  | 4. Besides itself, every knowledge source should only access blackboard.            |
| Pipe-and-Filter                  | 1. Both pipe and filter can only connect the other type (i.e. filter and pipe) directly.  
|                                  | 2. In order to reduce the complexity, it restricts the association of components to an acyclic graph or in a linear sequence.  
|                                  | 3. The type of filter is either passive or active.                                   |
| Reflection                       | 1. There are at least two levels, including a meta level and a base level.           
|                                  | 2. Base level components may only communicate with each other via a metaobject at meta level.  
|                                  | 3. System aspects that are expected to stay stable should not be at meta-level.     
|                                  | 4. A metaobject does not allow the base level components to modify its internal state.  
| Layer                            | 1. Every piece of a system is allocated to exactly one layer.                       
|                                  | 2. There are at least two layers.                                                   
|                                  | 3. The allowed-to-use relations should not be circular.                             
|                                  | 4. The number of relations between components that travel through a subset of the layers should be insignificant compared to the number of adjacent dependencies between adjacent layers.  

**Table 1.** Constraints of common architectural patterns.
CorrespondingControllerNumber(v) for a given view v to check whether there is a one-to-one relationship between v and its controllers. For a view element i, we define view cost (VC) that is similar to model cost.

\[ VC(i) = w_1 \text{CorrespondingModelUse}(i) + w_2 \text{ControllerUse}(i) + w_3 \text{CorrespondingControllerNumber}(i), \quad \text{where } w_1 \ll w_3, w_2 \ll w_3 \]  

(2)

In addition, for controller element c, we define CorrespondingModelUse(c) to count the state-change messages from c to its corresponding models. Similarly, two metrics CorrespondingViewNumber(c) and ViewUse(c) are defined for controller elements. Similar to MC and VC, we define controller cost (CC) for a given individual controller element i in MVC pattern:

\[ CC(i) = w_1 \text{CorrespondingModelUse}(i) + w_2 \text{ViewUse}(i) + w_3 \text{CorrespondingViewNumber}(i), \quad \text{where } w_1 \ll w_3, w_2 \ll w_3 \]  

(3)

In order to evaluate the quality of different candidate solutions in automated pattern synthesis, we need to calculate the fitness score for a given MVC solution. If LegalMVC is false for one solution, it means that the solution is not a reasonable MVC solution, and we simply set the fitness score as \( \infty \); or if LegalMVC is true, the fitness score is calculated by summing the individual cost for every model, view, and controller elements (Here, we suppose that the solution is composed of \( r \) models, \( s \) views, and \( t \) controllers).

\[
\text{Fitness}(s) = \begin{cases} 
  w_1 \sum_{i=1}^{r} MC(i) + w_2 \sum_{i=1}^{s} VC(i) + w_3 \sum_{i=1}^{t} CC(i) & \text{where LegalMVC is True} \\
  \infty & \text{where LegalMVC is False}
\end{cases}
\]  

(4)

As we can see from the MVC example, the evaluation for pattern constraint violations depends on the defined pattern metrics from pattern constraints. Since the quality of pattern metrics has a great impact on pattern synthesis, we describe the definition process for pattern metrics in detail in the next subsection so that architects can define the pattern metrics of specific architectural patterns of their own.

### 2.3. Definition process for pattern metrics

The definition process for pattern metrics is composed of three steps (discover the roles of a pattern, discover the relations within a pattern, and discover the domain related metrics), which are detailed below:

**Step 1: Discover the roles of a pattern.** As 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” [13], and there is no configurable generic implementations for patterns that cover their whole design space. However, every pattern has its invariable roles, such as model, view, and controller in MVC pattern. Therefore, we can identify some metrics from pattern roles. This step may include the following sub-steps:

(a) Define the metric of upper and lower limit about each role from pattern constraints. For example, in Layer pattern, we define $LegalLayers$ to ensure that the number of layers is more than one.

(b) Define the metric about the quantity relationship between different roles from pattern constraints. For example, in MVC pattern, we define the metrics $CorrespondingControllerNumber(v)$ and $CorrespondingViewNumber(c)$.

(c) Define the metric about type of roles from pattern constraints. For example, there is a passive filter or active filter in Pipe-and-Filter pattern, and different types of filter may influence the quality attribute of a system (e.g. performance). We define $PassiveFilterNumber$ and $ActiveFilterNumber$ for the two filter types in Pipe-and-Filter.

(d) Define the metric about responsibility of roles from pattern constraints. For example, according to the description of Constraint (2) in Blackboard pattern, the partial solutions acquired from each knowledge source and the control data should be stored in the role of blackboard. We define $ImproperDataNumber$ to count the number of data which are improperly stored outside blackboard.

(e) Define the metric about the mapping relations for components and pattern roles. In some patterns, one component may play multiple roles, or vice versa. For example, one component in Blackboard may play the role of blackboard and controller simultaneously, which leads to coupling between data and control logic. Hence we define $PureBlackboard$ for every blackboard in Blackboard pattern.

**Step 2:** Discover the relations within a pattern. Every pattern contains a set of interactions between the roles in the pattern. In this step, we identify the metrics from the interactions. It includes the following sub-steps:

(a) Define the metric for direction of interaction. The interaction between different roles in a pattern is usually unidirectional, such as the lower layer should not access the higher layer in Layer pattern, and model elements cannot depend on controller elements in MVC pattern.

(b) Define the metric for cycle interaction. This sub-step is similar to the previous sub-step, but it is more complex, since cycle interaction often includes more than two role elements.

(c) Define the metrics about constraints of relations between different roles. For example, for Constraint (1) of Pipe-and-Filter (as shown in Table 1), the relations between different roles are often limited in pattern constraints, which should be defined in pattern metrics.
(d) Define the metrics of relation types. There are many relation types between two elements in a pattern, such as inheritance, implementation, association, and so on. As different relation types may have different influence on quality attributes, we define specific metrics for the relation types.

(e) Define the metrics for interaction mechanisms. A set of interaction mechanisms exist between elements in a pattern (e.g. events or messages), we should consider the interaction mechanisms in patterns, and define metrics for them.

Step 3: Discover the domain related metrics. Every domain has its specific knowledge e.g. documented in literature and standards. Similarly, every software has its own application principles. In this step, architects define the metrics according to the domain and application principles.

The results of this definition process form a starting point for automated architectural synthesis. When this process finishes, the manual work by architects is completed as shown in Fig. 1. The pattern metrics, which are acquired through this definition process, are used to construct the fitness function for automated pattern synthesis.

2.4. Automated architectural synthesis

Automated architectural synthesis uses the outputs (patter metrics) of the definitions process as inputs, as shown in Fig. 1. During the automated process, we use a cooperative coevolution model for the two sub-processes: responsibility synthesis (RS) and pattern synthesis (PS). In the cooperative coevolution model, two or more populations evolve simultaneously with the fitness of each population depending upon the current population of the other [8]. We use two populations to represent candidate architectural solutions for RS and PS respectively, and one candidate solution acts as an *individual* in a population. In this section, we describe the cooperative coevolution model in detail.

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) [14] for each relation.

Figure 2 depicts an example of R and P set that has two classes (Classes 1 and 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, $O_1$ belongs to Class 1, while $A_2$ belongs to Class 2. Hence the value of $R$ set is $\{C_1, C_1, C_2, C_1, C_2\}$. 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 $O_1$ belongs to Layer 0 ($L_0$), while $O_3$ belongs to Layer 1 ($L_1$). In addition, as there is one dependency between responsibilities $O_2$ and $O_3$, $P$ set has one extra element which describes the connector type of this relation, remote procedure call (RPC).

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 ($\text{IndR}$):** $\text{IndR}$ 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 ($\text{IndP}$):** $\text{IndP}$ 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 (e.g. coupling, cohesion metrics) 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 responsibility assignment, 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 architectural solutions.

Hence, the proposed approach coevolves two populations: responsibility and pattern populations. Figure 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 before.

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 Sec. 2.4.1 for responsibility synthesis.
   b) *Best individual selection for pattern population (BestIndP)*: The best individual from pattern population is selected. We define a fitness function in Sec. 2.4.2 to ensure BestIndP has the least pattern constraint violations.

3) *Applying genetic operators*. Genetic operators include selection, crossover, and mutation operators [15]. In this step, for each population, a selection operator is

![Fig. 3. Cooperative coevolution procedure of our approach.](image-url)
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 responsibility synthesis; 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 responsibility assignment and minimized pattern constraint violations. In the following subsections, we describe representation of individuals for each population, and the fitness functions in detail.

### 2.4.1. 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 subsections.

(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 [16]. 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 Classes 1 and 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) [17]. 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 [17] 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 Eq. (5), 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:

$$Sim(R_i, R_j) = \frac{\sum_{k=1}^{n} w_k Sim_{dep}(R_i, R_j)}{\sum_{k=1}^{n} w_k}.$$  \hspace{1cm} (5)

(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 [14].

2.4.2. Fitness function

Many approaches have been proposed to address the responsibility assignment 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. Equation (6) shows the fitness function for responsibility synthesis, 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 [17].

$$Fitness\text{Score}_{RS} = Cohesion - Coupling - Complexity$$ \hspace{1cm} (6)

For pattern synthesis, our objective is to minimize the violations of pattern constraints. With the abovementioned definition process of pattern metrics, we can
define the pattern metrics to measure the number of violations for a specific type of pattern constraints. Hence we can quantify the pattern constraint violations for a specific pattern using Eq. (7), in which \( w_k \) represents the weight of a specific metric \( \text{metric}_k \) of the chosen pattern, and \( n \) is the number of pattern metrics for that pattern.

\[
\text{Fitness Score}_{PS} = \sum_{k=1}^{n} w_k \text{metric}_k
\] (7)

Architects can acquire architectural solutions which consider both responsibility synthesis and pattern synthesis. In each generation, the Pareto front can be established and updated with these solutions automatically. When the cooperative co-evolution 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.

3. Case Study

We conducted a case study to validate our proposed approach. The case used is a simplified cinema booking system, derived from a number of deployed Web-based cinema booking systems. This cinema booking system has a number of functional requirements, such as payment for tickets and making an advance booking. A specification of the use cases of the system is available from [18], which comprises 31 responsibilities (including 15 actions and 16 data), and 39 dependence relationships between the responsibilities.

The goal of our study is twofold: (1) validate whether the proposed approach can generate architectural candidate solutions with high cohesion, low coupling, and low complexity; (2) evaluate the similarity degree of automatically-generated design solutions compared with the expert design, which is produced by experienced architects.

The case study is described in two subsections: Section 3.1 defines the evaluation model, which is used to validate our approach. Section 3.2 reports the evaluation results. The procedure and rationale of this case study is as follows: We first calculate the object-oriented metrics for the solutions generated by our approach. The results are then compared with the results of traditional automated search-based approaches (e.g. MOGA [19] and Evolutionary Search [20]). However, all the object-oriented metrics introduced in existing work (e.g. [19–22]) only provide structural metrics, and we try to comprehensively evaluate the proposed approach in a “reality” perspective: to what extent the automatically-generated solutions are close to the expert design. Therefore we use the expert design as a baseline, and measure the similarity degree of the automatically-generated design solutions with the expert design.

3.1. Evaluation model

We evaluate the automated cooperative coevolution approach for architectural synthesis in two perspectives: (1) the object-oriented metrics, and (2) the
3.1.1. Object-oriented metrics

For object-oriented software systems, using object-oriented metrics is the most popular way to evaluate the quality of design. Many object-oriented metrics have been proposed to characterize, evaluate, and improve the design of object-oriented systems [23], and the quality of a software design is mostly measured with the coupling, cohesion, and complexity metrics [24]. For software architecture design, architects try to maximize cohesion while minimizing coupling and complexity. In this paper, a set of metrics based on CK metrics [25] (as shown in Table 2) are used to measure the cohesion, coupling, and complexity of candidate solutions. The expatiation of these metrics can be found in [19, 21, 22].

3.1.2. Similarity to the expert design

To evaluate the similarity degree of the design solutions generated by automated cooperative coevolution approach with the expert design, we use F-Score, which measures the accuracy of grouping responsibilities in classes for responsibility synthesis. If one candidate design solution is closer (more similar) to the expert design than other solutions, it is regarded as a more accurate responsibilities grouping. The definition and calculation of F-Score is provided in Eq. (8).

$$\text{F-Score}(\text{Cls}, C)_{RS} = \sum_{clsi \in \text{Cls}} \frac{|clsi| \times \max_{cj \in C} \{F1(clsi, cj)\}}{|R|}$$

(8)

in which

$$F1(clsi, cj) = 2 \times \frac{\text{Recall}(clsi, cj) \times \text{Precision}(clsi, cj)}{\text{Recall}(clsi, cj) + \text{Precision}(clsi, cj)}$$

in which

$$\text{Precision}(clsi, cj) = \frac{n_{ij}}{|c_j|}$$

$$\text{Recall}(clsi, cj) = \frac{n_{ij}}{|clsi|}$$

In Eq. (8), Cls denotes an architectural solution designed by experts, and clsi denotes a class in this design solution, which includes |clsi| responsibilities. Similarly, C represents a candidate solution that is generated automatically by our proposed approach for responsibility synthesis, and cj represents a class in C. n_{ij} records the number of responsibilities of class clsi covered by cj. The calculation of F-Score depends on the calculation of F1, which combines precision and recall from information retrieval [26]. When calculating F-Score, for each class clsi in the expert
design, the $F_1$ values between $cls_i$ and all the classes in $C$ are calculated. If $c_j$ gets the maximum $F_1(\text{cls}_i, c_j)$ value for $\text{cls}_i$, $c_j$ is regarded as the most similar class to $\text{cls}_i$. To keep the $F$-Score value in the interval $[0,1]$, we further normalize this value, which is divided by the number of total responsibilities $|R|$. The larger the $F$-Score value is, the higher the similarity is between the automated solutions and the expert design solution.

On the other hand, we also use $F$-Score to measure the similarity of automated solutions and the expert design for pattern synthesis. Similar to Eq. (8), the definition and calculation is shown in Eq. (9).

$$F\text{-Score} = \frac{\text{Score}(\text{Rol}, \text{Ro})_{PS}}{|R| + |\text{Relations}|}$$

in which

$$F_1(\text{rol}_i, \text{ro}_i) = 2 \times \frac{\text{Recall}(\text{rol}_i, \text{ro}_i) \times \text{Precision}(\text{rol}_i, \text{ro}_i)}{\text{Recall}(\text{rol}_i, \text{ro}_i) + \text{Precision}(\text{rol}_i, \text{ro}_i)}$$

in which

$$\text{Precision}(\text{rol}_i, \text{ro}_i) = \frac{n_{ii}}{|\text{ro}_i|}$$

$$\text{Recall}(\text{rol}_i, \text{ro}_i) = \frac{n_{ii}}{|\text{rol}_i|}$$

In Eq. (9), $\text{Rol}$ denotes the set of pattern roles for responsibilities and the set of pattern relations in the expert design solution. Similar to $\text{Rol}$, $\text{Ro}$ denotes the set of pattern roles for responsibilities and the set of pattern relations in an automated generated solution. To calculate $F$-Score, the $F_1$ values between each pattern role/pattern relation in the expert design and the same pattern role/pattern relation in generated solution are calculated. The $F_1$ values of all the pattern roles and pattern relations are accumulated, and then normalized by the size of $P$ set, introduced in Sec. 2.4, to get $F$-Score.

3.2. Evaluation

For evaluation, we first validate whether the proposed approach can generate architectural candidate solutions with high cohesion, low coupling, and low complexity. The results of this evaluation are presented in Sec. 3.2.1, which includes two parts. In the first part, we calculate the object-oriented metrics, which are shown in Table 2, for the solutions generated by our approach with different meta-heuristic algorithms and generations. The object-oriented metrics of the expert design are used as baseline value of the object-oriented metrics in Table 2. In the second part, we compare the cohesion, coupling, and complexity metrics of the solutions generated by our approach with those generated by other automated architectural synthesis approaches. However, the object-oriented metrics only evaluate the quality of design in a structural perspective, which is limited in design evaluation. To address this issue, we evaluate the proposed approach in a “reality” perspective: to what extent
the automatically-generated solutions are close to the expert design as presented in Sec. 3.2.2. We use the expert design as baseline, and measure the similarity degree of the automatically generated design solutions with the expert design for different automated architectural synthesis approaches. We use $F$-Score$_{RS}$ and $F$-Score$_{PS}$ introduced in Sec. 3.1.2 to calculate the similarity degree.

For comparison, we use NSGAII [27], IBEA [28], and random search as the meta-heuristic algorithms to generate the two populations during automated synthesis of architectural candidate solutions. The parameters of these algorithms are listed in Table 3.

Table 2. Object-oriented metrics used to evaluate our approach.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>For a given class $c_1$, MAC($c_1$) counts all method-attribute invocations from $c_1$ to other class $c_2$.</td>
<td>Coupling</td>
</tr>
<tr>
<td>MMC</td>
<td>For a given class $c_1$, MMC($c_1$) counts all method-method invocations from $c_1$ to other class $c_2$.</td>
<td>Coupling</td>
</tr>
<tr>
<td>CI</td>
<td>For a given class $c$, CI($c$) is the set of all indirect method-attribute or method-association end dependencies between the methods and the attributes or association ends in class $c$. And we assume CI$_{max}$ ($c$) is the set of all possible cohesive interactions in the class.</td>
<td>Cohesion</td>
</tr>
<tr>
<td>RCI</td>
<td>RCI($c$) denotes the number of cohesive interactions in $c$ over the number of possible such interactions. Formally, we define RCI($c$) =</td>
<td>CI($c$)</td>
</tr>
<tr>
<td>TCC</td>
<td>TCC($c$) denotes the number of pairs of methods with common usage in class $c$. Common usage means all the methods use an attribute or association end, or one method calls the other method. TCC($c$) is normalized by the number of possible method pairs in $c$.</td>
<td>Complexity</td>
</tr>
<tr>
<td>CIS</td>
<td>A count of the number of public responsibilities in classes.</td>
<td>Complexity</td>
</tr>
<tr>
<td>CSC</td>
<td>The ratio of classes’ size difference to the number of total responsibilities. It is used to control the size of classes.</td>
<td>Complexity</td>
</tr>
</tbody>
</table>

Table 3. Parameter settings for meta-heuristic algorithms evaluation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameters</th>
</tr>
</thead>
</table>
| NSGAII    | Responsibility Population Size: 100  
Pattern Population Size: 100  
Crossover Operation: Single Point Crossover  
Crossover Rate: 0.9  
Mutation Operation: Swap Mutation  
Mutation Rate: 1/length of chromosome  
Selection Operation: Binary Tournament |
| IBEA      | Responsibility Population Archive Size: 100  
Pattern Population Archive Size: 100  
Responsibility Population Size: 100  
Pattern Population Size: 100  
Crossover Operation: Single Point Crossover  
Crossover Rate: 0.9  
Mutation Operation: Swap Mutation  
Mutation Rate: 1/length of chromosome  
Selection Operation: Binary Tournament |
| Random Search | None |
### 3.2.1. Object-oriented metrics evaluation

Table 4 shows the value of different object-oriented metrics (coupling, cohesion, and complexity) of the best run of each meta-heuristics algorithm at 50, 100, 200, 300 generations. Each cell in the table represents the average value of the solutions. Table 5 shows the results of the object-oriented metrics of the expert design, which acts as a baseline to compare the three automated meta-heuristics algorithms. From the results shown in Table 4, we can find that all the three meta-heuristics can get higher cohesion, lower coupling, and complexity when the number of generations increases. After 300 generations, either NSGAII or IBEA can acquire ideal metrics values which are very close to those of the expert design. However, the design solutions generated by random search are not ideal because of much higher coupling, lower cohesion, and higher complexity compared with other meta-heuristic algorithms.

#### Table 4. Results of object-oriented metrics of solutions with different algorithms and generations.

<table>
<thead>
<tr>
<th>Generations</th>
<th>Property</th>
<th>Metric</th>
<th>Random search</th>
<th>NSGAII</th>
<th>IBEA</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Coupling</td>
<td>MAC</td>
<td>22</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMC</td>
<td>12</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Cohesion</td>
<td>RCI</td>
<td>0.42</td>
<td>0.61</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TCC</td>
<td>0.31</td>
<td>0.37</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>CIS</td>
<td>24</td>
<td>28</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSC</td>
<td>4.32</td>
<td>1.7</td>
<td>1.37</td>
</tr>
<tr>
<td>100</td>
<td>Coupling</td>
<td>MAC</td>
<td>25</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMC</td>
<td>8</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Cohesion</td>
<td>RCI</td>
<td>0.44</td>
<td>0.58</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TCC</td>
<td>0.35</td>
<td>0.35</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>CIS</td>
<td>20</td>
<td>34</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSC</td>
<td>3.89</td>
<td>1.42</td>
<td>1.09</td>
</tr>
<tr>
<td>200</td>
<td>Coupling</td>
<td>MAC</td>
<td>16</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMC</td>
<td>9</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Cohesion</td>
<td>RCI</td>
<td>0.39</td>
<td>0.64</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TCC</td>
<td>0.36</td>
<td>0.39</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>CIS</td>
<td>26</td>
<td>32</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSC</td>
<td>2.13</td>
<td>1.13</td>
<td>0.88</td>
</tr>
<tr>
<td>300</td>
<td>Coupling</td>
<td>MAC</td>
<td>12</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMC</td>
<td>11</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Cohesion</td>
<td>RCI</td>
<td>0.44</td>
<td>0.63</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TCC</td>
<td>0.35</td>
<td>0.39</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>CIS</td>
<td>27</td>
<td>30</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSC</td>
<td>2.54</td>
<td>1.02</td>
<td>0.81</td>
</tr>
</tbody>
</table>

#### Table 5. Results of object-oriented metrics of the expert design.

<table>
<thead>
<tr>
<th>Coupling</th>
<th>Cohesion</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>MMC</td>
<td>RCI</td>
</tr>
<tr>
<td>Expert Design</td>
<td>0.00</td>
<td>6.00</td>
</tr>
</tbody>
</table>
is because both responsibility synthesis and pattern synthesis are multi-objective optimization problem, and it is difficult to define a single objective fitness function appropriately when the two multi-objective optimization problems coevolve. However, the number of method-attribute invocations (MAC) is 0 in the expert design, because it is a design principle that data elements should be encapsulated by public interfaces of the class or component, and they are not allowed to be accessed directly from outside. However, none of the three meta-heuristics consider this design principle in optimization. Although we may introduce certain heuristics during the automated architectural synthesis to avoid e.g. the method-attribute invocations between different classes in a small system (e.g. the cinema booking system in this case study), it is difficult to do this for large and complex software systems. Existing automated approaches optimize the design solutions only from the structural perspective (i.e. coupling, cohesion, and complexity), which seems to be “cheating” software engineers [29]. Research in the next step should consider design principles to avoid such problem.

To estimate how cooperative coevolution approach influences on the object-oriented metrics, we compare the automated design results with (1) the results by traditional optimization approaches, which only consider responsibility synthesis (RS-only approaches in Table 6); and (2) the results by independent pattern synthesis, which only tries to avoid pattern constraint violations (PS-only approaches in Table 6). Since automated pattern recommendation is out of the scope of this work, we choose Layer pattern for pattern synthesis. Similarly, the expert design is used as the baseline to compare the design results by different automated approaches.

Table 6. Results of object-oriented metrics of solutions by different architectural synthesis approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Coupling</th>
<th>Cohesion</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAC</td>
<td>MMC</td>
<td>RCI</td>
</tr>
<tr>
<td>RS-Only</td>
<td>4</td>
<td>5</td>
<td>0.76</td>
</tr>
<tr>
<td>PS-Only</td>
<td>13</td>
<td>17</td>
<td>0.13</td>
</tr>
<tr>
<td>Cooperative Coevolution</td>
<td>3</td>
<td>8</td>
<td>0.67</td>
</tr>
<tr>
<td>Expert Design</td>
<td>0</td>
<td>6</td>
<td>0.71</td>
</tr>
</tbody>
</table>

To estimate how cooperative coevolution approach influences on the object-oriented metrics, we compare the automated design results with (1) the results by traditional optimization approaches, which only consider responsibility synthesis (RS-only approaches in Table 6); and (2) the results by independent pattern synthesis, which only tries to avoid pattern constraint violations (PS-only approaches in Table 6). Since automated pattern recommendation is out of the scope of this work, we choose Layer pattern for pattern synthesis. Similarly, the expert design is used as the baseline to compare the design results by different automated approaches.

For each automated architectural synthesis approach, we use IBEA as the default meta-heuristic algorithm, and the algorithm is run at 300 generations. The parameters setting is the same as that shown in Table 3. Since pattern synthesis only tries to avoid pattern constraints, and it does not consider the object-oriented metrics during synthesis process, the result of these object-oriented metrics has low quality for PS-only approach. On the contrary, since the RS-only approach is optimized for the object-oriented metrics, it has the best result of coupling, cohesion, and complexity. However, the better results based on structural metrics may not lead to real benefits for architecture because the automated architectural synthesis process using structural metrics for design optimization does not consider the expert design
experience and knowledge (e.g. design principles), and it may not meet the expectations of architects. In Sec. 3.2.2, we evaluate the proposed approach in a “reality” perspective: to what extent the automatically-generated solutions are close to the expert design.

3.2.2. Similarity degree evaluation

In this subsection, we set the expert design solution as a baseline, and measure the similarity degree of the design solutions generated by automated architectural synthesis approaches to the expert design. The approach to measure the similarity degree is described in Sec. 3.1.2, and the $F$-Score results are shown in Table 7.

Table 7. F-Score of different automated architectural synthesis approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Algorithm</th>
<th>$F$-Score$_{RS}$</th>
<th>$F$-Score$_{PS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS-Only</td>
<td>Random Search</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NSGAII</td>
<td>0.57</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>0.59</td>
<td></td>
</tr>
<tr>
<td>PS-Only</td>
<td>Random Search</td>
<td>0.66</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NSGAII</td>
<td>0.73</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>Cooperative Coevolution</td>
<td>Random Search</td>
<td>0.74</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>NSGAII</td>
<td>0.81</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>IBEA</td>
<td>0.79</td>
<td>0.91</td>
</tr>
</tbody>
</table>

The results in Table 7 show that the automatically-generated architectural solutions by our cooperative coevolution approach are most close to the expert design result. Compared with other architectural synthesis approaches, our approach gets a higher $F$-Score$_{RS}$ than traditional responsibility synthesis (RS-only) approach, and our approach also gets a higher $F$-Score$_{PS}$ than the pattern synthesis only (PS-only) approach. The results imply that architects should consider not only the responsibility assignment of the architecture, but also the design constraints (e.g. pattern constraints) of the architecture, when they synthesize architectural solutions.

The case study results show that our cooperative coevolution approach can generate design solutions that are most close to the solutions by experienced architects. In summary, the generated solutions by our proposed approach can include more architects’ experience and knowledge than the solutions generated by other automated architectural synthesis approaches.

4. Related Work

We summarize and discuss relevant work on automated architectural synthesis and pattern constraints in this section.

Cui et al. [30] presented an automated decision-centric architectural synthesis approach, which transits from requirements to architectures through a solution
exploiting and synthesizing process. In their approach, solution exploiting is accomplished by architects. For each elicited design issue, architects proposed solutions mainly based on their expertise and experience. Solution synthesizing in their approach is automated, which combines and evaluates all the feasible solutions from the solution exploiting results. Therefore, the quality of resulting solutions still heavily depends on the experience of architects.

Räihä [31] proposed to synthesize architecture using Genetic Algorithms (GA). In her approach, architectural styles and design patterns are used to transform the initial high-level architecture model to a detailed design. The architectural synthesis is based on an analysis model which contains information on functional requirements only. The differences between her approach and our proposed approach are that (1) design patterns and architectural styles are used as mutator for GA in Räihä’s approach, and these patterns are inserted or deleted randomly in GA mutation. In our approach, we focus on the constraints of patterns, and which patterns are used is determined; (2) the criteria for evaluating candidate architectural solutions are different. Our approach considers the design quality (e.g. cohesion and coupling metrics) which is similar to Räihä’s approach, while we also take pattern constraint violations into account.

Belle et al. [5] revisited the layer pattern to extract a minimum set of fundamental principles for using layer pattern, which are used to specify a series of constraints that a layered architecture should conform. They further made use of these constraints to guide the recovery of the layered architecture in a system, and model the architectural recovery as an optimization problem using automated heuristic search algorithm. However, their approach focuses on architecture recovery instead of architecture design, and their approach didn’t consider the responsibility assignment of architectural elements in recovering layer pattern.

Bagheri and Sullivan [32] showed that it is possible to separate and combine formal representations of application properties and architectural styles. The key idea of their approach is to map the application which is independent of architecture to models (i.e. Platform Independent Model, PIM) in model-based development, whilst to map the architectural styles to platforms (i.e. Platform Definition Model, PDM). Similar to our proposed approach, their approach separates the application synthesis and architectural style synthesis during architectural synthesis, which also followed the design concept of “divide-and-conquer”. However, their approach is different from our approach in that (1) they used ADLs to formally define the specifications about application models and architectural styles, then a mapping engine is used for translating from these specifications to architectural models in given styles during architectural synthesis, but our approach uses a search-based technique which is more flexible to explore the whole design space for candidate architectural solutions; (2) the treatment of pattern constraints is different. They used a constraint solver to support incremental analysis and construction of models, but we use pattern constraints to define pattern metrics which are used to evaluate different solutions.

Maoz et al. [33] used component and connector views (C&C views) to investigate the architectural synthesis problem, and further extended this basic problem with
support for architectural styles. Similar to [32], in their approach, they also used ADLs to formally define the C&C models and the architectural styles. Formal specifications using ADLs for architectural synthesis may end up with many satisfied solutions, and these satisfied solutions have different qualities (e.g. both solutions A and B are satisfied with the performance requirements of an application, but the performance of solution A is better than solution B). It is difficult to recommend better solutions with formal techniques, which is the issue that our approach addresses.

5. Conclusion and Future Work
Architectural synthesis essentially links the problem to the solution space, and it plays a key role in architecting process from requirements to initial architecture design. However, due to its essential complexity, this architecting activity heavily depends on the experience of architects. In this paper, we extend the existing architectural synthesis (AS) activity to a pattern-based AS, and propose a cooperative coevolution approach that synthesizes architecture automatically using architectural patterns. We first analyze several common architectural patterns, identify their pattern constraints, and represent the pattern constraints as first-class entities for pattern implementation. We then present a process to define pattern metrics from pattern constraints, and acquire the pattern metrics to construct the fitness function for automated AS. The automated AS process is composed of two sub-processes: pattern synthesis (PS) and responsibility synthesis (RS). We further model these two sub-processes as a cooperative coevolution problem, which can be executed automatically to synthesize candidate architectural solutions. We evaluate the proposed approach through a case study: architecture design of a cinema booking system, and the results show that compared with other automated architectural synthesis approaches, our cooperative coevolution approach can generate architectural solutions which are closer to the expert design.

We outline the future work in two points: (1) to improve the fitness function of responsibility synthesis based on the knowledge of architects and developers, such as object-oriented design principles; (2) to develop a dedicated tool that supports the pattern-based automated AS using cooperative coevolution and can be integrated into development platforms (e.g. IDE) for a practical use in architecture design.

Appendix A. Abbreviations Used in This Paper

AA : Architectural Analysis
ADL : Architecture Description Language
AS : Architectural Synthesis
ASR : Architectural Significant Requirement
CC : Controller Cost
CI : Cohesive Interaction
CIS : Class Interface Size
CR : Centroid Responsibility
CSC : Class Size Complexity
C&C : Component and Connector
DMA : Direct Method-Attribute Dependency
DMM : Direct Method-Method Dependency
DRRS : Direct Responsibility-Responsibility Semantic Dependency
GA : Genetic Algorithm
IBEA : Indicator-Based Evolutionary Algorithm
MAC : Method-Attribute Coupling
MC : Model Cost
MMC : Method-Method Coupling
MOGA : Multi-Objective Genetic Algorithm
MVC : Model-View-Controller
NSGA : Non-Dominated Sorting Genetic Algorithm
PDM : Platform Definition Model
PIM : Platform Independent Model
PS : Pattern Synthesis
RCI : Ratio of Cohesive Interaction
RPC : Remote Procedure Call
RS : Responsibility Synthesis
SBSD : Search-Based Software Design
SBSE : Search-Based Software Engineering
TCC : Tight Class Cohesion
VC : View Cost

References


30. X. Cui, Y. Sun and A. H. Mei, Towards automated solution synthesis and rationale capture in decision-centric architecture design, in Proc. 7th Working IEEE/IFIP Conf. on Software Architecture, 2008, pp. 253–256.

