Automated Software Architectural Synthesis using Patterns: A Cooperative Coevolution Approach

Yongrui Xu
State Key Lab of Software Engineering
School of Computer, Wuhan University
Wuhan, China
xuyongrui@whu.edu.cn

Peng Liang*
State Key Lab of Software Engineering
School of Computer, Wuhan University
Wuhan, China
liangp@whu.edu.cn

***Abstract*—In software architecting process, 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 with patterns based on the results of this analysis. We also describe the definition process for pattern metrics, which are used for automated architectural synthesis, from pattern constraints. 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. Myx architectural pattern is used as an example to illustrate our approach.**

Keywords—automated architectural synthesis; architectural patterns; cooperative coevolution

I. 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 mentioned 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 architecture solutions automatically. This approach tries to avoid the violations to the pattern constraints while considering the responsibility assignment in the resulting architecture 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 we mentioned before, when the candidate architecture solutions are synthesized, there 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

* Corresponding author

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

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 specific design context. The contributions of this work are: (1) defining and using pattern metrics to measure the violations to pattern constraints in architecture design; and (2) translating the pattern-based architectural synthesis to a cooperative coevolution problem, which can be automated.

The rest of this paper is organized as follows. Section II introduces the automated architectural synthesis approach in detail. Section III uses Myx architectural pattern, which is used in ArchStudio [9], as an example, to explain the use of our approach. Related work is discussed in Section IV and we conclude and outline future directions in Section V.

II. 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, which are not covered in this paper. 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.

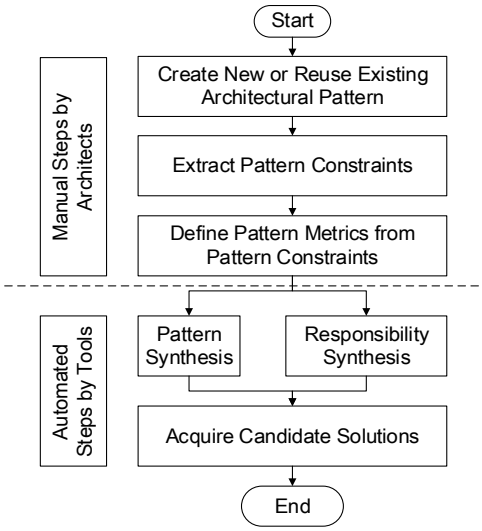

Figure 1. Extended architectural synthesis activity

A. 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 architecture solutions to a set of ASRs largely depends on the experience of architects in classical AS, and to make the matter worse, there is 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 Figure 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 Figure 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. 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].

B. Constraints and Metrics of Architectural Patterns

An architectural pattern is composed of a triple {*context*, *problem*, *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.

Table 1. Constraints of common architectural patterns

Pattern Name	Pattern Constraints
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.

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 *LegalMVC* metric to judge whether the solution satisfies Constraint (1) of MVC pattern (as shown in Table 1). For Constraint (2), we define *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 *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 *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 Formula (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. α and β are the value of weight. Since *ControllerUse* has a higher weight than *CorrespondingViewUse*, we set $\alpha << \beta$. From Formula (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.

$$MC(i) = \alpha CorrespondingViewUse(i) + \beta ControllerUse(i), \quad where\ \alpha << \beta \quad (1)$$

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 *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.

$$VC(i) = \alpha CorrespondingModelUse(i) + \beta ControllerUse(i) + \gamma CorrespondingControllerNumber(i), \quad where\ \alpha << \gamma, \beta << \gamma \quad (2)$$

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:

$$CC(i) = \alpha CorrespondingModelUse(i) + \beta ViewUse(i) + \gamma CorrespondingViewNumber(i), where\ \alpha << \gamma, \beta << \gamma \quad (3)$$

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 ∞; 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).

$$Fitness(s) = \begin{cases} \alpha\sum_{i=1}^{r} MC(i) + \beta\sum_{i=1}^{s} VC(i) + \gamma\sum_{i=1}^{t} CC(i) & when\ LegalMVC\ is\ True \\ \infty & when\ LegalMVC\ is\ False \end{cases} \quad (4)$$

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.

C. 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:

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 attributes of a system (e.g., performance). We define *PassiveFilterNumber* and *ActiveFilterNumber* for the two filter types in Pipe-and-Filter pattern.

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.

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 ofter 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.

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 Figure 1. The pattern metrics, which are acquired through this definition process, are used to construct the fitness function for automated pattern synthesis.

D. Automated Architectural Synthesis

As discussed in Section II.A, with the extended pattern-based AS, architects can propose architectural solutions using patterns either by themselves (e.g., based on their experiences) or through pattern recommendation [14]. However, how to assign responsibilities to architecture design elements in RS sub-process and how to group elements to implement the architectural pattern in PS sub-process are challenging tasks in practice. Addressing both of them heavily depends on the experience of architects. Automation of AS activity is beneficial in that (1) it can evaluate the rationality of responsibility assignment to architectural elements; and (2)

avoid the violation to pattern constraints to a certain extent automatically. In addition, in the architecture design of large and complex systems, most of design problems may have a massive number of possible solutions, which is impossible to manually explore. To this end, we choose to employ a scalable meta-heuristic search technique to assist automated AS using patterns.

In our recent work [15], we formally defined the problem of automated pattern-based architectural synthesis. In our approach, the outcome of RS and PS sub-processes are regarded as two populations of solutions that evolve simultaneously, and the fitness of each individual in one population depends on the status of individuals in another population, i.e., the two populations have a cooperative coevolution relationship [8]. Therefore, it is feasible and reasonable to model the pattern-based AS as a cooperative co-evolution optimization problem, which can be executed automatically to synthesize candidate architecture solutions.

The choice of the representations for collaboration problems and the definitions of the fitness functions for individuals in different populations are two ingredients for cooperative co-evolution. In RS (responsibility synthesis) sub-process, on one hand, the main decision is whether an architectural element should take certain responsibility. We use a binary string encoding scheme from the SBSE representation techniques [16] to represent a responsibility that is assigned to certain architectural elements. On the other hand, as there have been some mature metrics associated with the responsibility assignment problem that form good initial candidates for the RS fitness function (e.g., cohesion and coupling metrics), we directly use these metrics for RS in the cooperative coevolution. In PS sub-process, we consider what specific role an architectural element plays in that pattern, and we use one digit (0~9) to represent the role type. Then, we use the pattern metrics defined for each pattern constraint as the PS fitness function in the cooperative coevolution. Due to space limitation, the representations for RS and PS populations and the fitness functions for evaluating these two populations are detailed in [14].

III. EXAMPLE

In this section, we use Myx architectural pattern, which is employed in ArchStudio [9], as an example, to explain the use of our proposed approach.

A. Architectural Pattern Constraints

We omit the details of Myx pattern, which can be found in [17], due to space limitations. We only describe the major constraints of this pattern: (1) Each component has two zones (i.e., top and bottom), and all interfaces belonging to a component should be assigned to either of these two zones. In any invocation or dependency relationships, if one interface is assigned to the top zone of one component, the other interface in the same relationship should be assigned to the bottom zone of another component, or the other way round (hence, in Myx pattern, the top and bottom zones of each component is the pattern roles); (2) Cycle invocations (i.e., one invocation from the top zone of a component to the bottom zone of same component) are not permitted (whether a cycle invocation exists is determined by the grouping of architectural elements; (3) Only upward synchronous invocations, in which the invocation direction is from the top zone of one component to the bottom zone of another component, are permitted, while asynchronous invocations have no any limitations, i.e., either upward or downward asynchronous invocations are permitted (an invocation direction also depends on the grouping of architectural elements).

B. Metrics from Pattern Constraints

Existing cohesion and coupling metrics (e.g., [18][19]) provide initial candidates for fitness function in RS sub-process, and we focus on the metrics from pattern constraints for fitness function in PS sub-process in this subsection. To use these metrics, we should specify above or below relationships between two components. We propose a heuristic approach to decide which component between two components is an "above" component. A "counter" property is introduced for each component. When there is an invocation from the top zone of component *A* to the bottom zone of another component *B*, we increase the *B*'s counter with 1. Otherwise, if the invocation comes from the bottom zone of component *A* to the top zone of component *B*, *A*'s counter is increased by 1. Finally, we compare the value of the counters in two components, and the component that has a bigger counter value is regarded as the "above" component. For example, the counter values of component *A* and *B* are 1 and 3 respectively as shown in Figure 2, and then component *B* is "above" component *A*.

According to the definition process described in Section II.C, we define a set of metrics specified below based on the constraints of Myx pattern presented in Section III.A:

(1) *IntraUse(i)*: Since cycle invocations are not permitted, a component may never be above or below itself in invocation relationships. This metric measures the number of invocation relationships whose two connected interfaces are in the same component *i*, that violate Constraint (2).

(2) *IllegalUse(i,j)*: This metric denotes the number of invocation relationships between components *i* and *j* that violate Constraint (1).

(3) *Intra&IllegalUse(i)*: This metric is similar to *IntraUse(i)*. The difference between the two metrics is that this metric counts the invocation relationships that the two connected interfaces are not only in the same component *i,* but also in the same zone. It measures the number of invocation relationships that violate both Constraint (1) and Constraint (2).

(4) *ComplexUse(i,j)*: According to Constraint (3), downward invocation is only permitted for asynchronous invocations. In component-based software design, asynchronous invocations may complicate the system design, which is difficult to understand. The metric *ComplexUse(i,j)* denotes the number of asynchronous calls between component *i* and *j*. It indirectly measures the complexity of the design.

(5) *CycleUse(i,j)*: When component *i* is above component *j* in invocation relationships, the invocations from *i*'s top zone to *j*'s bottom zone lead to cycle invocations between the two components. Thus, we introduce this metric to count the number of calls that violate Constraint (2).

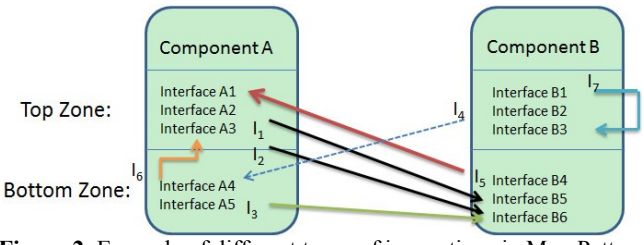

Figure 2. Example of different types of invocations in Myx Pattern

In Figure 2, there are 7 invocations between different interfaces (from I_1 to I_7), and we distinguish different kinds of invocations with different colors and lines. Invocation I_1 and I_2 are from component *A* to its above component *B*. These invocations satisfy all of the pattern constraints to be considered when using Myx pattern, since the two interfaces are in different components and only one of them is in top (or bottom) zone. According to the metrics definition in Section III.B, Invocation I_3 can be counted in metric *IllegalUse(A,B)*; I_4 can be counted in metric *CycleUse(A,B)*; I_5 is a typical asynchronous invocation that can be counted in metric *ComplexUse(A,B)*; I_6 and I_7 can be counted in metric *IntraUse(A)* and *Intra&IllegalUse(B)* respectively.

We further define component violation cost (*CVC*) for each component in the architecture design. In our definition, *CVC* accumulates the numbers of metrics for all the violated constraints to one component, while it can set weight for different types of constraint violations. Hence, we calculate *CVC* for a given component *i* using Formula (5):

$$CVC(i) = \alpha CycleUse(i,j) + \beta Intra\&IllegalUse(i) + \gamma IllegalUse(i,j) + \delta ComplexUse(i,j) + \varepsilon IntraUse(i), \quad where\ i \neq j \qquad (5)$$

Then, we define an objective function that calculates the total violation cost (*TVC*) of a given architecture design solution using Formula (6). We try to acquire a candidate architecture solution with minimized *TVC*, which means the pattern constraint violations in the architecture design solution are minimized as well.

$$TVC = \sum_{i=1}^{n} CVC(i) \qquad (6)$$

C. Evaluation of Synthesis Results

The *TVC* metric defined in previous subsection can be used as an evaluation criterion for the PS sub-process. In the RS sub-process, most existing work focuses on the evaluation for the distribution of the responsibilities among components in the architecture design, such as the cohesion and coupling metrics [19]. In our proposed cooperative coevolution approach, one population evaluated by the cohesion and coupling metrics is used for RS, and another population evaluated by the *TVC* metric is used for PS. In every generation, the best individuals (i.e., the solutions that have high cohesion and low coupling) in RS population are combined with the individuals in population used by PS, and these hybrid individuals are used as input for the next generation in PS. Similarly, the best individuals (i.e., the solutions that have less pattern constraint violations) in PS are combined with the individuals in RS population as input for the next generation in RS. In each round of generation by PS and RS, we use a single fitness function for both populations to evaluate the overall quality of the resulting architecture solutions. Therefore, we define the quality of a given solution *s* in Formula (7):

$$Quality(s) = \alpha Cohesion\&Coupling(s) + \beta TVC(s) \qquad (7)$$

We use this formula to evaluate the candidate architecture solutions generated by the proposed approach using cooperative coevolution, and select and recommend the solution with highest (best) quality. It provides optimized architecture design solutions for architects.

IV. Related Work

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

Cui *et al.* [20] 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 [21] 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 architecture 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 [22] showed that it is possible to separate and combine formal representations of application properties (e.g., domain knowledge) and architectural styles. The key idea of their approach is to map the application which is independent of architecture styles to *models* (i.e., Platform Independent Model, PIM) in model-based development, and

map the architectural styles to *platforms* (i.e., Platform Definition Model, PDM). Similar to our proposed approach, their approach seperates 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 to translate these specifications to architectural models in given architectural styles during architectural synthesis, while 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 (solutions), but we use pattern constraints to define pattern metrics which are used to evaluate different solutions.

Maoz *et al.* [23] 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-based architectural synthesis. Similar to [22], they also used ADLs to formally define the C&C models and the architectural styles. Architectural synthesis with formal specifications using ADLs may end up with many satisfied solutions, and these satisfied solutions may have different qualities (e.g., both solutions *A* and *B* satisfy the performance requirements of an application, but the performance of solution *A* is better than solution *B*). It is difficult to recommend better solutions in candidate solutions with formal ADL techniques, which is the issue that our approach tried to addresses using cooperative coevolution.

V. Conclusions 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 AS 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 the pattern constraints, and represent them as a first-class entity 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 synthesis. The automated synthesis 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 architecture solutions. We use Myx architectural pattern as a concrete example to explain the use of the proposed approach.

We outline the future work in two points: (1) to conduct controlled experiments that compare the architecture design quality between the generated pattern-based AS solutions using our approach and the solutions by architects based on the same design problems in industry projects; (2) to develop a tool that support the pattern-based automated AS using cooperative coevolution.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practise, 3rd ed. Addison-Wesley Professional, 2012.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, 1st ed. Wiley, 1996.

[3] K. D. Babu, P. Govindarajulu, A. R. Reddy, "ANP-GP approach for selection of software architecture styles," Int. J. Softw. Eng., 1(5):91-104, 2011.

[4] M. Galster, A. Eberlein, M. Moussavi, "Systematic selection of software architecture styles," IET Softw., 4(5):349-360, 2010.

[5] A. Belle, G. El Boussaidi, C. Desrosiers, H. Mili, "The layered architecture revisited: Is it an optimization problem?," in SEKE, 2013, pp. 344-349.

[6] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, M. A. Babar, "A comparative study of architecture knowledge management tools," J. Syst. Softw., 83(3):352–370, 2010.

[7] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, "Software architecture optimization methods: A systematic literature review," IEEE Trans. Softw. Eng., 39(5):658-683, 2013.

[8] M. Harman, S. A. Mansouri, Y. Zhang, "Search-based software engineering: Trends, techniques and applications," ACM Comput. Surv., 45(1):1-61, 2012.

[9] J. Garcia, I. Krka, C. Mattmann, N. Medvidovic, "Obtaining ground-truth software architectures," in ICSE, 2013, pp. 901-910.

[10] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, P. America, "A general model of software architecture design derived from five industrial approaches," J. Syst. Softw., 80(1):106-126, 2007.

[11] F. P. Brooks, The Design of Design: Essays from a Computer Scientist, 1st ed. Pearson Education, 2010.

[12] A. Tang, H. van Vliet, "Modeling constraints improves software architecture design reasoning," in WICSA, 2009, pp. 253-256.

[13] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, 1st ed. Wiley, 2007.

[14] M. Kassab, G. El-Boussaidi, H. Mili, "A quantitative evaluation of the impact of architectural patterns on quality requirements," in SERA, 2011, pp. 173-184.

[15] Y. Xu, P. Liang, "Co-evolving pattern synthesis and class responsibility assignment in architectural synthesis", in ECSA, 2014. (under review).

[16] M. Harman, P. Mcminn, J. T. De Souza, S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in Empirical Software Engineering and Verification, 2012, pp. 1–59.

[17] "The Myx Architectural Style." Available at: http://isr.uci.edu/projects/archstudio/myx.html, accessed on 2013-11-22.

[18] M. Bowman, L. C. Briand, Y. Labiche, "Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms," IEEE Trans. Softw. Eng., 36(6):817-837, 2010.

[19] C. L. Simons, I. C. Parmee, R. Gwynllyw, "Interactive, evolutionary search in upstream object-oriented class design," IEEE Trans. Softw. Eng., 36(6):798-816, 2010.

[20] X. Cui, Y. Sun, H. Mei, "Towards automated solution synthesis and rationale capture in decision-centric architecture design," in WICSA, 2008, pp. 221-230.

[21] O. Räihä, "Genetic Algorithms in Software Architecture Synthesis," Ph.D Thesis, School of Information Sciences, Tampere University, 2011.

[22] H. Bagheri and K. Sullivan, "Monarch: Model-based development of software architectures," in MODELS, 2010, pp. 376–390.

[23] S. Maoz, J. O. Ringert, and B. Rumpe, "Synthesis of component and connector models from crosscutting structural views," in ESEC/FSE, 2013, pp. 444–454.