Detecting Architecture Erosion by Design Decision of Architectural Pattern

Lei Zhang, Yanchun Sun*, Hui Song, Franck Chauvel, Hong Mei
Institute of Software, School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China
Email: {zanglei07, sunyc, songhui06, franck.chauvel}@sei.pku.edu.cn, meih@pku.edu.cn

Abstract—In the software architecture design, architects usually adopt many classic architectural patterns. However, such important design decisions often fail to be effectively captured in practice. The problem called “architecture erosion” may happen easily. During the design iterations, the latest design may accidentally violate the constraints implied by existing architectural patterns. During the evolution of system, the improper modification may conflict with the original design.

In this paper, we present an approach for detecting architecture erosion, by capturing the most important design decisions about the adopted architectural patterns. Architects can use a collection of predefined and frequently-used architectural patterns in their design. Our supporting tool can capture what and how the architectural patterns are used, and then generate a set of corresponding OCL code automatically. With these specifications, we can both verify the architecture in design phase and validate the run-time architecture to detect architecture erosion. Finally, we use an illustrative example to demonstrate the feasibility of this approach.

keywords: architectural pattern, architecture erosion, design decision

I. INTRODUCTION

Nowadays as the size and complexity of software systems increase, software architecture plays an important role in software development [1]. It is important not just in the development phase, but in every aspect of software development, from the initial conceptualization of the system through requirements to the system’s deployment. [2]–[4]. Nevertheless, in current architecture centric development process, the problem called “architecture erosion” (or “decay”) [5] frequently happens in many projects. On one hand, it is common to see the architecture design of a system and its implementation diverged to a certain extend. On the other hand, when architecture undergoes several evolutions for changed or new added requirements, architecture erosions may also happen.

The key cause of architecture erosion is the loss of the important “design decisions” [5], which are taken during the architectural design and embedded in the resulting software architecture implicitly. To address this problem, design decision became an emerging research trend in recent years and it drew increasingly attention in architecture community [6]. Many useful approaches and methods have been proposed. Such as, Archium [7], AREL [8], ArchDesigner [9], AQUA [10], and our own ABC/DD [11]. Most of them provide different methods to capture kinds of design decisions. However, as pointed by van Vliet et al., there are too many design decisions but not all of them are that important [2], and it is expensive to document design decisions. Therefore, we hope to only capture the minimal set of the really important ones.

Adopting architectural patterns1 is one kind of the most important decision. Architectural pattern expresses a fundamental structural organization or schema for software architecture design [14], and in fact it is incorporated in most software architectures. Since 1994 when Shaw and Garlan classified architectural patterns in their seminal book [15], a number of patterns have been applied in many modern systems. Many are used frequently. For instance, pipe-filter, black-board, multi-layers, model-view-controller, client-server, and so on.

However, architectural pattern does not become the focus in most design decision approaches. In practice, adopted architectural patterns are usually hidden in architecture design without an explicit notation. When these design decisions are lost, it will take time to identify those used patterns. Thus, they might be distorted unintentionally by other patterns or improper modifications. Even worse, once those patterns have been programmed, they disappear in the code. It is more difficult to ensure whether they are implemented faithfully.

To address this problem, we present an approach to detect architecture erosion, by capturing the most important design decisions about the adopted architectural patterns. Architects can use a collection of predefined and frequently-used architectural patterns in their design. Our supporting tool can capture what and how the architectural patterns are used, and then generate a set of corresponding OCL code automatically. With these specifications, we can both verify the architecture in design phase and validate the run-time architecture to detect architecture erosion.

The rest of the paper is organized as follows: Section II describes the motiveative example; Section III presents our proposed approach of capturing architectural pattern, as well as the process of our approach; then Section IV provides an illustrated example using this approach; and related work are discussed in Section V; in the final Section VI, we discuss some limitations of our approach and future work.

---

1We use the term “architectural pattern” as synonym for “architecture style” as suggested by Hofmeister et al. [12] and Bass et al. [13]
II. Motivative Example

Multi-layer architecture (also referred as n-layers architecture) is one of the most common used architectural pattern. When adopting multi-layer pattern, the architect needs to break up a system into several layers which logically grouping components that offers a cohesive set of services. Each layer provides a set of services to the layer above and uses the services of the layer below. The most widespread use of multi-layer pattern is three-layer architecture as shown in Fig.1(a). It separates presentation, application processing, and the data management components into three independent layers.

Two topological constraints are imposed when adopted multi-layer pattern. Firstly, components in one layer cannot use services provided by the upper layer. Secondly, components cannot use services across the layer. The dashed line marked with error in Fig.1 (b) and (c) illustrate the two violations respectively.

Nevertheless in practice, throughout the entire software life cycle there are some situations in which architecture erosion may occur.

(a) Patterns in Architecture Design

Within one architecture, often several architectural patterns can coexist in the same design. Previously used patterns may conflict with the subsequently used patterns during architecture design iterations.

For example, multi-layer pattern in Fig.1 (a) was adopted already, then Model-View-Controller (MVC) pattern was adopted later. The architect wants to separate the components of model, view, and controller into different layers. Then the interaction between these components maybe conflict with multi-layer pattern, if it is not designed properly.

(b) Patterns in Implementation

The best architecture is worthless if the code does not follow it [1]. However, it is hard to guarantee each design introduced by architects can be faithfully transformed to program source code.

For example, the layered architecture design in Fig.1 can be violated accidentally. A report of a real project said that the key developer of the GUI directly accessed the database without modifying architecture design just like Fig.1 (c) [16]. Even the popular open-source IDE – Eclipse, which adopts multi-layer pattern, has similar layers violations, as discussed in [17].

Similar violations also can happen during the evolution phase of systems. Without an explicit representation of architectural pattern, people often modify the system without a fully understanding of the system. If it evolves in a way that conflicts with the original design, the architectural pattern may be broken.

III. Proposed Approach

A. Software Architecture Meta Model

Based on our previous underlying meta-model for architecture design [11], we extend it to support architectural pattern, as shown in Fig.2.

B. Predefined Architectural Pattern

In order to provide a handily cataloged architectural pattern repository of ready-to-use, we must clearly define what constitute a pattern in advance. According to existing literature, mainly [3], [14], [15], [18], we use the minimal set of constituent elements to characterize an architectural pattern:

(1) Structure

Each architectural pattern has a structure which is constituted by several subsystems. The name of each subsystem provides a vocabulary of design solution. For example, pipe, filter, blackboard, client, server. We use a package diagram to illustrate the structure of a pattern.

(2) Structural Constraints

The constraints define how elements can be integrated to form this pattern and determine allowed interaction between subsystems, e.g., as discussed in the motivative example, components in multi-layers architectural pattern, should not access the components across the layer, as Fig.1 (c). We use natural language to describe these constrains, and use a set of templates of OCL code to precisely describe them.

(3) Known Quality Attributes Impact

Most architectural pattern will affect the quality attributes. Applying a given architectural pattern may make it easier or harder to implement certain quality attributes. Based on existing literature, we collect known impact on quality attributes. Table.1 lists some of the most representative ones.
C. Process of Proposed Approach

The process of our approach is depicted in Fig.3. There are in all four segments horizontally representing four phases of the software life cycle, including requirement analysis, architecture design, implementation, deployment/evolution. In the vertical direction, the diagram is divided into two compartments: the upper one contains manual activities, while the lower one contains the activities finished by our tool.

(a) Requirement Analysis Phase

In the first phase, architects must organize all the architectural significant requirements (ASR) of software system, including both functional and non-functional requirements.

(b) Architecture Design Phase

(b.1) Select Architectural Pattern

By examining the predefined architectural patterns, architects should select a suitable one as a starting point according to identified non-functional requirement and the known quality attributes impact.

(b.2) Import Predefined Architectural Pattern

With the help of our modeling tool, adopting a specific architectural pattern is simple. All that the architect needs to do is to click on the selection from the palette of patterns, then drag and drop to the editing area. By doing this, our tool will automatically place a skeleton of this pattern. For example, Fig.4 shows the case that MVC pattern is adopted. The rounded rectangles labeled with the name of each subsystem are explicit notations of the adopted architectural patterns.

The action of adopting a pattern is an important design decision and it will be captured by our tool automatically in chronological order.

(b.3) Detailed Design

After importing the architectural pattern, architect then can

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Pattern Structure</th>
<th>Pattern Element</th>
<th>Structural Constraints</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe and Filter</td>
<td></td>
<td>A set of Pipes</td>
<td>Pipe is connected with a filter</td>
<td>+ modifiability, + portability, + scalability, - performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End with a pipe not a filter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model-View-Controller</td>
<td></td>
<td>Model, View, Controller</td>
<td>Components in View and Model should not directly access All the connections should be managed by Controller</td>
<td>+ scalability, + extensibility, + reusability, + evolvability, - performance</td>
</tr>
<tr>
<td>Layers</td>
<td></td>
<td>Layer</td>
<td>If component resides in layer i, then it can only access the components reside in layer i and i-1.</td>
<td>+ portability, + extensibility, + reusability, - performance, - evolvability, - reconfigurability</td>
</tr>
<tr>
<td>Client-Server</td>
<td></td>
<td>Client, Server</td>
<td>Client should only access the server Client can not access other clients</td>
<td>+ scalability, + evolvability, - reusability</td>
</tr>
</tbody>
</table>

Table 1. Predefined Architectural Patterns
continue the detailed design within the generated skeleton of pattern. Concrete components and connectors can be filled into the skeleton. The explicit skeleton makes the boundary of architectural pattern more clearly and prevents an ad-hoc architecture design.

(b.4) Generate Architectural Pattern Specification

When the architecture design is finished, the supporting tool will generate a set of OCL code to specify every constraints imposed by existing architectural patterns.

In our predefined architectural pattern repository, pattern constraints are denoted as several templates of OCL code. When generating the executive OCL code, parameters in those templates will be substituted by concrete components and connectors in pattern.

For example, in the Model-View-Controller architectural pattern, one constraint is that components of Model cannot directly access components of View, and all the connections should be managed by components of Controller. We denote this constraint as a template of code as shown in Fig.5. The parameters denoted as $c_1$ to $c_3$ and $r_1$ to $r_3$ represent components and connector respectively, and they will be substituted by concrete components, such as “OnlineShop”, “Compare” components in Fig.4.

The reason why we use OCL is because it is a precise language that provides constraint expressions on any MOF model or meta-model [19]. The meta-model that we used is conformed to MOF, therefore we chose OCL to specify pattern constraints.

context MVC_PATTERN:

\[
\text{invariant} \\
\begin{align*}
    c_1.\text{name} &= \text{"controller" and } c_2.\text{name} = \text{"view" and } c_3.\text{name} = \text{"model"} \\
    \text{and component } &:: \text{AllInstances()} \rightarrow \text{includes}(c_1) \text{ and } r_1.\text{source} = c_1 \text{ and } r_1.\text{target} = c_2 \\
    \text{and connector } &:: \text{AllInstances()} \rightarrow \text{exists}(r_3) \text{ and } r_3.\text{source} = c_2 \text{ and } r_3.\text{target} = c_3
\end{align*}
\]

(c) Detect Architecture Erosion

In our approach, the generated OCL code can be used to detect architecture erosion in two situations, as discussed in section II. Firstly, we can take a static verification to detect whether there are some conflicts among existing architectural patterns. When a conflict is detected, the tool will warn the architect and highlight the components and connector that are improperly designed in the architectural model.

When system has been implemented and deployed, we can leverage the ability of reflecting the runtime system’s architecture [20] to reflect the real architecture. By executing the OCL codes against this architecture, we can judge whether the system still conforms to its original design. When architecture erosion is detected, ABC tool also will prompt a warning and highlight the problematic components and connectors.
D. Tool Support

Based on our previous architecture modeling tool, ABC (Architecture-Based Component Composition) Tool [21], which we developed as a new plug-in to support the proposed approach. This tool implements an integrated environment that provides support for designing architecture, adopting predefined architectural pattern, capturing design decision and detecting architecture erosion.

ABC Tool is developed as a suite of Eclipse Plug-ins. We mainly use three open source frameworks: EMF (Eclipse Modeling Framework) to model our architecture model, architectural pattern, design decision; and use GMF (Graphical Modeling Framework) to provide visual design ability; and use Eclipse-OCL to parse and execute generated constraints.

IV. ILLUSTRATIVE EXAMPLE

A. A Simple On-line Voting System

In this section, we use a simple case to illustrate our approach. This is a project selected from a software architecture course for graduate students. Requirements of this on-line voting system includes: (1) provide a user interface for each kind of user; (2) when login, user can view each candidate’s information, (3) user can vote for a selected candidate; (4) each user can only vote at most three times; and (5) this system should have high portability.

B. Using the Proposed Approach

One group of students’ architecture is presented in Fig.6. In this architecture, there are three adopted architectural patterns. The first one is three-layer pattern; and within the middle layer, there is the second pattern: pipe-and-filter; and the third one is model-view-controller which is scattered in three layers. In total, there are eight components and seven connectors.

![Fig. 6. Architecture of On-line Voting System](Image)

In the help of ABC Tool, this group of students used our proposed approach and finished this project. By reviewing this project, we find that ABC Tool detected 2 structural problems in their architecture design and 2 implementation problems during the runtime stage. That is to say, there are 4 potential architecture erosion. We list them in Table.2.

<table>
<thead>
<tr>
<th>Architecture Design</th>
<th>Run-time Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three-Layer</td>
<td>1) “User Interface” to “Vote Record”</td>
</tr>
<tr>
<td></td>
<td>2) “User Interface” to “Candidate Information”</td>
</tr>
<tr>
<td>Pipe-Filter</td>
<td>null</td>
</tr>
<tr>
<td>Model-View-Controller</td>
<td>null</td>
</tr>
</tbody>
</table>

Table.2. Detected Architecture Erosion

For example, during the architecture design stage, architect connected “User Interface” to “Vote Record” which broke the constraint of multi-layer pattern; the tool also detected an improper design that connected “User Interface” to “Candidate Information”, which violated the constraint of model-view-controller pattern.

V. RELATED WORK

A. Software Architectural Pattern

There are three major approaches used for modeling architectural patterns, including ADL(Architecture Description Language), UML and formal Method.

Most of ADLs are introduced by academia and aim at representing software architectures in general [22]. Some of them directly support architectural pattern description and also leverage formal or semi-formal approaches for the formalization of pattern specifications. For example, ACME [23] uses first-order logic language, Armani, to formally specify architectural pattern. Likewise, Wright ADL uses CSP and ArchWare uses (π-calculus), etc.

UML aims at providing a generic modeling language. There are also some researchers tried to use UML to describe software architectures [24] and architectural patterns [25].

Some researchers proposed to use pure formal methods to describe architectural pattern. For example, [26] uses graph grammars to formal specify pattern. Similarly, [27] also uses some formal methods to analysis architectural pattern. Another comprehensive work [18] presents a semi-formal way using architectural primitives to model architectural patterns. Their work provided a solid foundation for our work.

B. Software Architecture Design Decision

In recent years, there are lots of valuable work in design decision area. Such as Archium, AREL, AQUA, ArchDesigner, ABC/DD, and so on. The most related one is SEURAT [28] that supports the software architects in selecting architectural patterns, design patterns, and idioms for use in architecture. It also captures the rationale for each pattern and the rationale and store them in the pattern library. Compared to our work, although SEURAT records the decisions of adopted pattern and corresponding rationale, it did not provide a visual modeling environment and did not support verification.
VI. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an architectural pattern-based approach to capture important architecture design decisions. By predefining a collection of architectural patterns, architects can directly select and apply them in their architecture design. Meanwhile our ABC tool can capture these important design decisions, i.e., adopted architectural patterns, and generate a set of OCL code to specify the constraints imposed by those patterns. Then, we can detect architecture erosion in both conceptual design and runtime by executing the generated OCL code.

This paper presented our preliminary work. So far, we have already identified a number of areas that call for further improvements and some of them have become our ongoing work.

The first limitation is that we do not support variants of architectural pattern. Some patterns have several variants to adapt to a specific problem. For example, in the multi-layer pattern, sometimes components in a high layer do need to use components in a very low layer. Such intentional violation of layers constraints is called "layer bridging". Similarly, there are many patterns have variants, so we need to improve this problems.

The second limitation is that we do not provide an extension mechanism for architects to add some new architectural patterns. All the predefined patterns in our tool are hard-coded. This problem limits the extendibility of our approach. Since our tool support is based on the Eclipse platform, we plan to take advantages of the meta-modeling capabilities of EMF to support this.

The third limitation is our approach is unavailable to support specifying behavior aspect of architectural pattern. We plan to extend this work to detect critical behavioral design decisions and their trade-offs.

As part of our future work, we also plan to evaluate the feasibility of our approach with quantitative experiment in a more complex system in real world.

ACKNOWLEDGMENT

This effort is sponsored by the National Basic Research Program of China (973) under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, and the National Natural Science Foundation of China under Grant No. 61073020. The authors want to express their gratitude to the anonymous reviewers for their valuable comments that helped to improve this work.

REFERENCES