An Approach to Documenting and Evolving Architectural Design Decisions

Meiru Che
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712, USA
meiruche@utexas.edu
https://webspace.utexas.edu/~mc36838

Abstract—Software architecture is considered as a set of architectural design decisions (ADDs). Capturing and representing ADDs during the architecting process is necessary for reducing architectural knowledge evaporation. Moreover, managing the evolution of ADDs helps to maintain consistency between requirements and the deployed system. In this work, we create the Triple View Model (TVM) as a general architecture framework for documenting ADDs. The TVM clarifies the notion of ADDs in three different views and covers key features of the architecting process. Based on the TVM, we propose a scenario-based method (SceMethod) to manage the documentation and the evolution of ADDs. Furthermore, we also develop a UML metamodel that incorporates evolution-centered characteristics to manage evolutionary architectural knowledge. We conduct a case study to validate the applicability and the effectiveness of our model and method. In our future work, we plan to investigate how to support ADD documentation and evolution in geographically separated software development (GSD).

I. INTRODUCTION

Software architecture plays a foundational role in achieving system functional and non-functional requirements. The architecting process provides a high-level framework to support designing, developing, testing, and maintaining software systems. The traditional concept of software architecture focuses on components and connectors, as Perry/Wolf proposed in [21]. Although the achievement by recognizing components and connectors is significant in research and industry, some problems still remain in software architecture theory and practice. As the most critical aspects of the problems for researchers and practitioners, architectural knowledge representation and knowledge evaporation have major influence on complexity and cost of system evolution, communication among stakeholders, as well as software architecture reuse.

Perry and Wolf considered the selection of elements and their form to be architectural design decisions (ADDs), and the justification for these decisions to be found in the rationale. It was not until 2004, with Bosch’s paper [4] at the European Workshop on Software Architecture, that software architecture has generally come to be considered as a set of ADDs. This specific focus on ADDs led to a broader focus on architectural knowledge [19]. Capturing and representing ADDs helps to organize architectural knowledge and reduce its evaporation, thereby providing a better control on many fundamental architectural drift and erosion problems in the software life cycle [21]. In the research related to our work, the focus has been on the development of models and tools to capture, represent, and share ADDs [5], [17], [27]. However, most of the existing methods document ADDs with different emphases, and moreover, they seldom support architecture evolution and knowledge evolution very well in practice [6], which is also critical for architectural knowledge management and needs more attention in research and industry [20].

Our work is that ADDs can be explicitly documented by a scenario-based approach, which covers three views of an architecture that form the basis of our work, to record architectural knowledge and to manage decision evolution for reducing knowledge evaporation. Thus, our work addresses both the documentation and the evolution needs for ADDs. Furthermore, we also make an extension to address these needs in geographically separated software development (GSD). Specifically, We define a Triple view Model (TVM) as an architecture framework that helps to clarify the notion of ADDs in three different views. Based on the TVM, we propose a scenario-based method (SceMethod) for capturing and documenting ADDs that enables us to manage architectural knowledge explicitly. We also develop a UML metamodel to manage ADD evolution in a fine-grained way, in which several evolution-centered characteristics are incorporated to ensure the evolution of architectural knowledge can be captured and tracked properly. In addition, we are going to focus on supporting ADD management in GSD, i.e., the architectural knowledge management is not localized and centralized but geographically distributed. This part of work is currently in progress.

We believe our research offers useful suggestions and insights for architectural knowledge management, and will also provide benefits and incentives for researchers and practitioners in improving methodologies on documenting and evolving architectural knowledge.

II. RESEARCH QUESTIONS

We identify the following four main research questions that we aim to answer in our research:

RQ1: What prescriptions and issues in an architecting process should be considered as ADDs? How can we collect them to form an ADD set?
RQ2: How does one derive and document the ADDs in a real software project context?
RQ3: How does one manage ADD evolution effectively and explicitly with minimum evolutionary knowledge evaporation?
RQ4: What are the typical ADD management paradigms in the context of GSD, and how does one document and evolve ADDs in GSD projects?

III. APPROACH

In this section, we will give a sketch of our proposed solutions and future research plans.

A. Triple View Model

In order to capture the ADD set, we proposed the Triple View Model (TVM) to clarify the notion of ADDs and to cover key features in an architecting process [7].

The TVM is defined by three views: the element view, the constraint view, and the intent view. This is analogous to Perry/Wolf model’s elements, form, and rationale but with expanded content and specific representations [21]. Each view in the TVM is a subset of ADDs, and the three views together constitute an entire ADD set. Specifically, the three views cover three different aspects when creating an architecture, i.e., “what”, “how”, and “why”, as shown in Fig. 1. The three aspects aim to specify design decisions on “what” elements should be selected in an architecture, “how” these elements combine and interact with each other, and “why” a certain decision is made.

The detailed contents in each view of the TVM are illustrated in Fig. 2.

In the element view, the ADDs describe “what” elements should be selected in an architecting process. We define computation elements, data elements, and connector elements in this view. Computation elements represent processes, services, as well as interfaces in a software system. Data elements indicate data accessed by computation elements. Both computation elements and data elements are regarded as components in software architecture, and connector elements are (at minimum) communication channels (that is, mechanisms to capture interactions) between those components in the architecture.

In the constraint view, the ADDs are defined as behavior, properties, and relationships. They describe constraints on system operations and are typically derived from requirement specifications. Specifically, behavior illustrates what a system should do and what it should not do in general. It specifies prescriptions and proscriptions based on requirement specifications. Properties are defined as constraints on a single element in the element view, and relationships are constraints on interactions and configurations among different elements.

The ADDs in the intent view are composed of rationale and best-practices in an architecting process. Rationale, which includes alternatives, motivations, trade-offs, justifications and reasons, is generated when analyzing and justifying every decision that is made. Best-practices are styles and patterns we choose for system architecture and design. The ADDs in the intent view mainly exist as tacit knowledge [26].

B. Scenario-Based Documentation and Evolution Method

The TVM is the foundation of ADD documentation and evolution. Based on the TVM, we proposed the scenario-based ADD documentation and evolution method (SceMethod) [7].

In the SceMethod, we aim to obtain and specify the element view, constraint view, and intent view through end-user scenarios, which are represented by Message Sequence Charts (MSCs) [22]. Figure 3 illustrates the SceMethod process. At the beginning of the architectural design process, we obtain initial ADD results. Later on, as the requirements change, the ADDs are evolved and refined according to the newly requirements. By documenting all the possible ADDs and evolving these decisions with changing requirements, the SceMethod effectively helps us to make architectural knowledge explicit and to reduce architectural knowledge evaporation.

Basically, we have the following four steps in the SceMethod to derive ADDs in a software project. For the sake of brevity, we will not discuss the detailed process of each step, but just give a brief introduction. We have the full illustration in [9].

1) Initialization: Before applying the TVM to end-user scenarios, the requirements of the software system are elicited, then we use MSCs to describe both the positive and negative scenarios. An MSC is composed of agent instances, interaction messages, and the timelines of the agents.

2) From MSC Syntax to Element View: We derive the element view directly from the syntax of MSCs. Specifically, each agent instance is taken as a computation element, and from the interaction messages between the source and target agent instances, we can extract data elements accessed by computation elements. Connector elements serve as communication channels between computation elements. Therefore, the element view is derived as follows:
Computation Elements = \{Agent Instances\}
Data Elements = \{Interaction Messages\}
Connector Elements = \{Channels between Agents\}

3) From MSC Semantics to Constraint View: Based on the semantics of MSCs, we analyze behavior, properties, and relationships of the goal system to document ADDs in the constraint view. The ADDs on the behavior of the system are documented as:

\[
\begin{align*}
\text{Behavior} & = \{\text{Prescriptions; Proscriptions}\} \\
\text{Prescriptions} & = \{\text{Positive Scenarios}\} \\
\text{Proscriptions} & = \{\text{Negative Scenarios; Exceptions}\}
\end{align*}
\]

In addition, we use three factors to define properties, and we adopt simple path expressions to illustrate the interacted events in the MSCs to specify relationships:

\[
\begin{align*}
\text{Properties} & = \{\text{Receive; Issue; Check}\} \\
\text{Relationships} & = \{\text{Event Traces by Path Expressions}\}
\end{align*}
\]

4) Intent View Documentation: Since decision making strategies are usually behind stakeholders’ thoughts, the intent view cannot be derived directly from MSCs, which make it difficult to define a formal specification for documenting the intent view. The best way to make the intent explicit is to record decision making strategies as the architecting process moves forward. Specifically, answering each question that occurs to the stakeholders in the architecting and designing phase is helpful to constitute the ADDs in the intent view. Besides, architectural styles, architectural patterns and design patterns that we apply as best-practices should also be recorded as design decisions in the intent view.

\[
\begin{align*}
\text{Rationale} & = \{\text{Answers to The Intent-Related Questions}\} \\
\text{Best-Practices} & = \{\text{Architectural Styles/Patterns; Design Patterns}\}
\end{align*}
\]

C. Evolution-Centered UML Metamodel

Furthermore, we developed a UML metamodel for the TVM. Each view in the TVM is specified by classes with a bunch of attributes describing the ADD information. The UML metamodel provides evolution-related attributes, helps to establish traceable evolution chains, and supports multiple ways to document version-specific rationale during the architecting process. The most important benefit from the metamodel is that it enables us to make ADD evolutionary knowledge explicit, therefore all the stakeholders can share consistent architectural knowledge with minimum knowledge evaporation. We discussed the UML metamodel in details in our previous work [8].

D. ADD Management in GSD

This part is our ongoing work currently. Based on different organizational approaches of global software projects, we are going to explore and analyze three typical ADD strategies and paradigms for managing architectural knowledge documentation and evolution in a GSD context. In addition, we aim to conduct field studies in geographically separated software projects to compare and analyze different ADD paradigms.

IV. Evaluation

Our TVM and SceMethod have been validated in a case study on an industrial project provided by the Italian electrical company ENEL [1], to derive ADDs as an architecture and to drive the following design, development and testing processes. Based on the requirement specifications of the power plant monitoring system in this project, we established end-user scenarios to describe the functionality of the system, and derived the ADDs in the element, constraint, and intent views that covered significant architecture prescriptions and issues.

We applied the UML metamodel to this case study as well, and provided a thorough analysis and discussion on the overall results for ADD documentation and evolution. In our future work, we plan to conduct field studies in geographically separated development projects to compare and analyze ADD management paradigms, thereby providing insights into architectural knowledge management in the GSD context.

V. Related Work

The key concepts of the traditional view on software architecture are components and connectors [3], [21]. In a number of current approaches, software architecture has been seen as a set of ADDs [4], [16], [25]. The architectural decisions in the software architecting process are an increasing focus of researchers and practitioners [13], [18], and are also considered to be a part of architectural knowledge [19]. In [12], a systematic review for architectural knowledge is
presented, and different definitions on architectural knowledge and how they are relevant to each other are discussed as well.

Guidelines for documenting software architecture have been provided in [11] and [15]. However, those documentation approaches do not explicitly capture ADDs in the architecting process. Recently, many models and tools have been proposed for capturing, managing, and sharing ADDs.

Tyrees template [27] provides a simple document describing key architectural decisions that establishes a concrete direction for design and implementation, and also clarifies the rationale for different stakeholders. In [19], an ontology of ADDs and their relationships have been described. This ontology then can be used to construct architectural knowledge of a software system. ADDSS [5] is a web-based tool for documenting ADDs. It establishes the backward and forward traceability between requirements, decisions, and architectures. Archiuim [17] is a Java tool, including a compiler and a run-time environment, for supporting, capturing, tracing and managing ADDs. It also provides visualization for design decisions by using a dependency graph, which is easy for stakeholders to evaluate and to track the decisions. Other models and tools such as AREL [24] and PAKME [2] are also proposed for managing architectural knowledge.

In addition, the research on managing the evolution of ADDs has been focused in the software architecture area as well. In [14], an approach for assisting architects in reasoning about architectural evolution paths has been described, and the concept of evolution style is defined in it. Some other techniques as discussed in [10] and [28] introduce different approaches for capturing architectural evolution and selecting architectural evolution alternatives.

A detailed comparison of these existing models and tools has been done in [23]. We can see that most of the existing methods document ADDs with different emphases, and moreover, they seldom support architecture evolution and knowledge evolution very well in practice [6]. Our work is that ADDs can be explicitly documented by a scenario-based approach, which covers three views of an architecture that form the basis of our work, to record architectural knowledge and to manage decision evolution for reducing knowledge evaporation, thereby addressing both the documentation and the evolution needs for ADDs.

VI. EXPECTED CONTRIBUTIONS

This research is expected to make the following contributions:

1. Define a general framework of ADDs. The what-how-why triple view clarifies the notion for documenting ADDs.
2. Provide a scenario-based method as an effective way to derive ADDs and keep architectural knowledge explicit and consistent during architecture evolution.
3. Bring a fine-grained definition in a UML metamodel to capture ADD evolutionary knowledge.
4. Give researchers and practitioners further insights into the ADD management in geographically separated software development.

ACKNOWLEDGMENT

This work was carried out as the PhD program, supervised by Dr. Dewayne E. Perry at The University of Texas at Austin, and was supported in part by the NSF under Grant Nos. IIS-0438976 and CCF-0820251.

REFERENCES