Copyright
by
Meiru Che
2011

The Thesis Committee for Meiru Che
Certifies that this is the approved version of the following thesis:

Scenario-Based Architectural Design Decisions Documentation and Evolution

APPROVED BY

SUPERVISING COMMITTEE:

Dewayne E. Perry, Supervisor

Sarfraz Khurshid

Scenario-Based Architectural Design Decisions Documentation and Evolution

by

Meiru Che, B.S.; M.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2011

Dedicated to my husband and our parents,

for their love and support.

Acknowledgments

I would like to express my deep gratitude to my advisor Dr. Dewayne E. Perry for his constant support and guidance in my research and in shaping this thesis. Special thanks to Dr. Sarfraz Khurshid for sparing his time in evaluating this thesis. I would also like to express my heartfelt thanks to all the members of the ESEL group for all the helpful discussions we have had. Finally, I wish to thank my family for their love and support for me.

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967 and CCF-0820251.

Scenario-Based Architectural Design Decisions Documentation and Evolution

Meiru Che, M.S.E
The University of Texas at Austin, 2011

Supervisor: Dewayne E. Perry

Software architecture is considered as a set of architectural design decisions. Capturing and representing architectural design decisions during the architecting process is necessary for reducing architectural knowledge evaporation. Moreover, managing the evolution of architectural design decisions helps to maintain consistency between requirements and the deployed system. In this thesis, we create the Triple View Model (TVM) as a general architecture framework for documenting architectural design decisions. The TVM clarifies the notion of architectural design decisions in three different views and covers key features of the architecting process. Based on the TVM, we propose a scenario-based methodology (SceMethod) to manage the documentation and the evolution of architectural design decisions. We also conduct a case study on an industrial project to validate the applicability and the effectiveness of the TVM and the SceMethod. The results show they provide complete documentation on architectural design decisions for creating a system architecture, and well support architecture evolution with changing requirements.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **ix**

List of Figures **x**

Chapter 1. Introduction **1**
- 1.1 Contributions 3
- 1.2 Organization 3

Chapter 2. Triple View Model **4**
- 2.1 Overview 4
- 2.2 Model Details 5
- 2.3 Advantages of the Triple View Model 8

Chapter 3. Scenario-based Methodology **9**
- 3.1 Overview 9
- 3.2 Methodology Details 10
  - 3.2.1 Initialization 11
  - 3.2.2 Element View Derivation 12
  - 3.2.3 Constraint View Derivation 13
  - 3.2.4 Intent View Derivation 14

Chapter 4. Case Study **17**
- 4.1 Background 17
- 4.2 Research Questions 18
- 4.3 End-user Scenarios 19

4.4 Results 19
4.4.1 Element View 19
4.4.2 Constraint View 19
4.4.3 Intent View 25
4.5 Analysis 26
4.6 Discussion 30
4.6.1 Practicality 30
4.6.2 Scalability 31
4.6.3 Limitations 31

Chapter 5. Related Work **33**

Chapter 6. Extension for Future Work **36**
6.1 Basic Idea 36
6.2 Software Ecosystems Characteristics 37
6.3 Open Challenges 40

Chapter 7. Conclusion **42**

Bibliography **44**

List of Tables

4.1 The Element View Results 22
4.2 The Properties Results For The Constraint View 24
4.3 Questions For Establishing The Intent View 26

List of Figures

2.1 Triple View Model Framework 5
2.2 Triple View Model and Software Architecture 6
2.3 Triple View Model for Architectural Design Decisions 6

3.1 The SceMethod Process 10
3.2 An MSC Example 11

4.1 MSC Specifications of the Power Plant Monitoring System (positive scenarios) 20
4.2 MSC Specifications of the Power Plant Monitoring System (negative scenarios) 21
4.3 The Structure Diagram of The Power Plant Monitoring System 25

Chapter 1

Introduction

Software architecture plays an important role in achieving functional and non-functional requirements. The architecting process provides a high-level framework to support designing, developing, testing, and maintaining software systems after deployment. The traditional concept of software architecture focuses on components and connectors, as Perry/Wolf proposed in [26]. 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, and software architecture reuse.

Perry and Wolf considered the selection of elements and their form to be architectural design decisions, 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 finally come to be considered as a set of architectural design decisions.

This specific focus on architectural design decisions led to a broader focus on architectural knowledge [23]. Capturing and representing architectural design decisions helps to organize architectural knowledge and reduce its evaporation, thus providing a better control on many fundamental architectural drift and erosion problems in the software life cycle. In the research related to our work, the focus has been on the development of models and tools to capture, manage, and share architectural design decisions [32], [9], [20]. A brief comparison and analysis of the existing models and tools has been conducted in [28]. However, there is still no agreed notion on what should be considered as an architectural design decision during an architecting process. Besides, current models and tools do not support architecture evolution very well, which is also critical for architectural knowledge management and needs more attention in research and industry [24].

To address this need, we propose the Triple View Model (TVM) as a general architecture framework of architectural design decisions. The TVM divides architectural design decisions set into three different views, i.e., the element view, the constraint view, and the intent view. These three views specify architectural design decisions by three aspects, “what”, “how”, and “why”, and all the architectural design decisions are regarded as a software architecture. In addition, based on the TVM, we present a scenario-based methodology (SceMethod) for architectural design decisions documentation and evolution, which enables us to manage architectural knowledge effectively. We subsequently conduct a case study to validate our TVM and SceMethod.

1.1 Contributions

We make the following three contributions in this thesis:

1) The Triple View Model (TVM) - A general framework of architectural design decisions. The "what" - "how" - "why" triple view clarifies the notion when documenting architectural design decisions;

2) The scenario-based methodology (SceMethod) - A scenario-based approach to architectural design decisions documentation and evolution. It provides an effective way to derive architectural design decisions and keep architectural knowledge complete and consistent during architecture evolution;

3) A substantial case study - A validation for the TVM and the SceMethod on an industrial project. The results demonstrate the applicability and the effectiveness of the TVM and the SceMethod.

1.2 Organization

The rest of this thesis is organized as follows. Section 2 describes the overview of the TVM, and then discusses the TVM in detail. Section 3 presents the scenario-based method of architectural design decisions documentation and evolution. In section 4, we conduct a case study to validate the TVM and the SceMethod in an industrial project, and analyze the research questions based on the study results. Section 5 discusses related work on architectural design decision models and architecture evolution. Section 6 discusses the ideas for future work, and we conclude the thesis in section 7.

Chapter 2

Triple View Model

This chapter first presents the overview of the Triple View Model (TVM), and then describes the contents of the TVM in detail. It finishes by discussing the advantages of the TVM.

2.1 Overview

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 [26]. Each view in the TVM is a subset of architectural design decisions, and the three views constitute an entire architectural design decisions set. Specifically, the three views mean three different aspects when creating an architecture, i.e., "what", "how", and "why", as shown in Figure 2.1. The three aspects aim to cover 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.

During the architecting process in the software life cycle, architects are the main role operating architectural design decisions. However, architectural

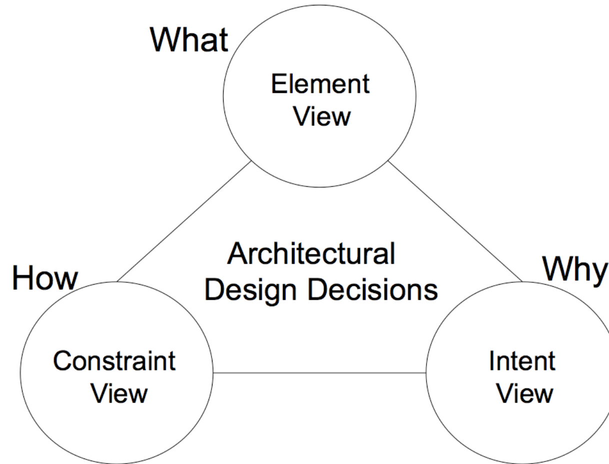

Figure 2.1: Triple View Model Framework

decisions may also be brought forward by programmers, project managers, or customers in real software project environment. In any case, the TVM provides a right selection of architectural design decisions, and it is applicable for all stakeholders. Moreover, the TVM suggests a systematical way to include complete architectural decisions for creating an architecture. Figure 2.2 shows the relations among architectural design decisions, the TVM and software architecture in a system.

2.2 Model Details

In this section, we discuss the detailed contents of each view in the TVM, which are illustrated in Figure 2.3.

In the element view, the architectural design decisions describe "what" elements should be selected in an architecting process. We define computation elements, data elements, and connector elements in this view. Computation el-

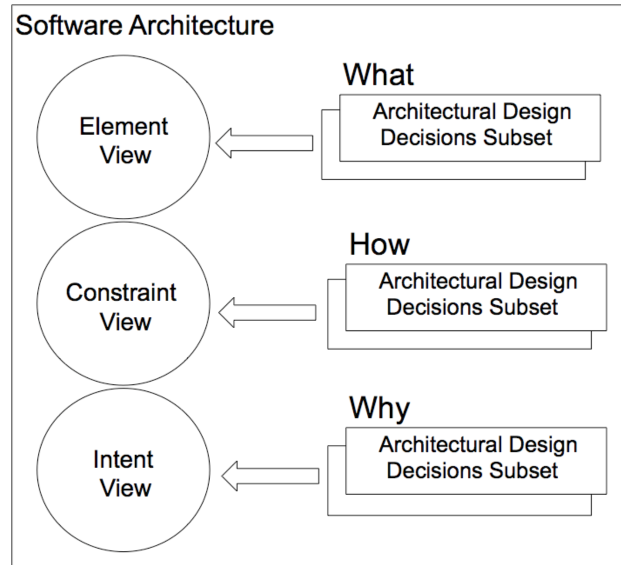

Figure 2.2: Triple View Model and Software Architecture

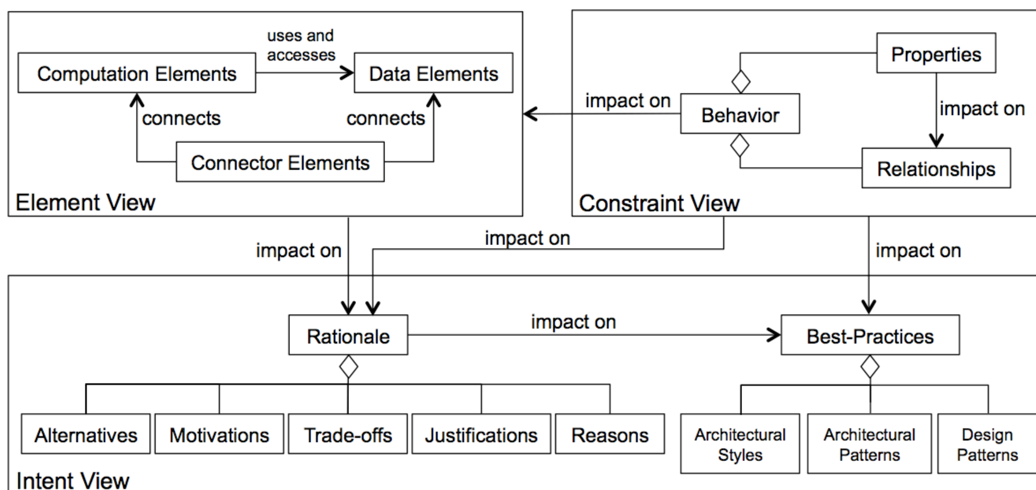

Figure 2.3: Triple View Model for Architectural Design Decisions

ements represent processes, services, and 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 communication channels between those components in the architecture. Note that the architectural design decisions in the element view consist of traditional architecture concepts, which are mainly represented by components and connectors.

In the constraint view, the architectural design decisions 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, and influences the design decisions in the element view. Properties are defined as constraints on a single element in the element view, and relationships mean interactions and configurations among different elements.

The architectural design decisions in the intent view are composed of rationale and best-practices in the 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 architectural decisions in the intent view mainly exist as tacit knowledge [31], and we need to document them during the decision making process, so that stakeholders can clearly understand these tacit architectural knowledge during

the architecting process. What's more, the consistent communication among different stakeholders effectively decreases architectural knowledge evaporation.

2.3 Advantages of the Triple View Model

The Triple View Model provides us a fundamental framework for architectural design decisions and covers key features of the architecting process. It has the following advantages:

First, the TVM captures architectural design decisions not only on components, connectors, and their relationships, but also on intent behind each design decision. It is essentially consistent with the traditional concept of software architecture, and helps researchers and practitioners grasp both the fundamental concepts and the decision making strategies in an architecting process;

Second, the TVM enables us to establish a complete set of architectural knowledge, which provides clear directions for communication among different stakeholders in the software development life cycle;

Third, the TVM supports scenario-based architectural design decisions documentation and evolution, and finally supports software architecture evolution.

Chapter 3

Scenario-based Methodology

In this chapter, we propose the scenario-based architectural design decisions documentation and evolution method (SceMethod). We first provide the overview of the SceMethod, and then discuss the methodology step by step to illustrate how to manage the documentation and the evolution of architectural design decisions.

3.1 Overview

The TVM is the foundation of architectural design decisions documentation and evolution. 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). Most functional requirements can be represented by end-user scenarios through MSCs; while non-functional requirements and quality attributes probably cannot be directly shown in the scenarios. However, in the end, all non-functional properties can be reified functionally into architecture design decisions, so that we still can manage non-functional properties in the SceMethod. Figure 3.1 illustrates the SceMethod process. We can see that for the first time we apply this method,

we obtain initial architectural design decisions results. Later on, as the requirements change, the architectural decisions are evolved and refined according to the newly requirements. By documenting all the possible architectural design decisions and evolving these decisions with changing requirements, the SceMethod effectively makes architectural knowledge explicit and reduces architectural knowledge evaporation.

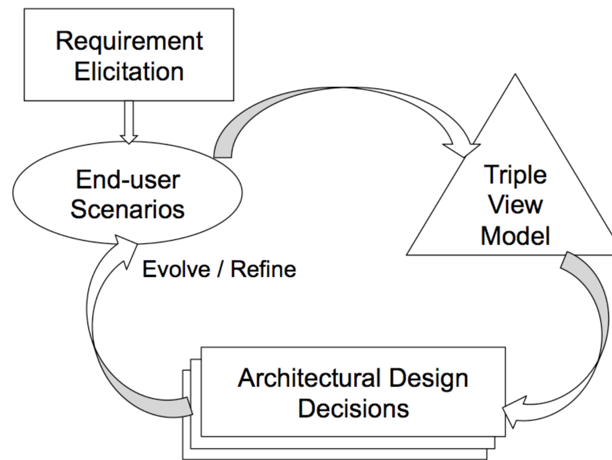

Figure 3.1: The SceMethod Process

3.2 Methodology Details

In the following sections, we discuss each step of the SceMethod in detail. Basically, the SceMethod includes an initialization step which uses MSCs to specify scenarios, and the other three steps each deriving one single view in the TVM.

3.2.1 Initialization

Before applying the TVM to end-user scenarios, the requirements of the software system are elicited, and then we use MSCs to describe both the positive and negative scenarios. MSC is used for representing end-user scenarios [27], and it is a widespread notation for describing scenarios as its UML counterpart, sequence diagrams. Specifically, an MSC is composed of vertical lines, horizontal arrows, and agent instances. Figure 3.2 is a simple example of an MSC [27].

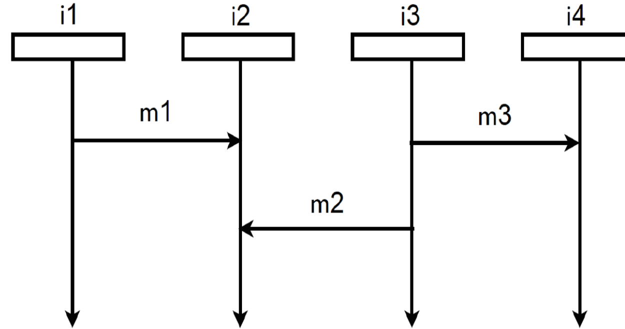

Figure 3.2: An MSC Example

The vertical line associated with the agent instance specifies the timeline of the corresponding agent. The horizontal arrow shows the interaction message between the source and the target agent instances. In Figure 3.2, we can see that i1, i2, i3, and i4 are agent instances, and each of them has a timeline. m1, m2, and m3 are three interaction messages among the four agent instances. Based on the end-user scenarios represented by MSCs, we initially derive the architectural design decisions as defined in the TVM. If the scenarios change afterward, we then track the evolution of the decisions and

refine them based on the changing requirement specifications. The following three steps illustrate the complete SceMethod process.

3.2.2 Element View Derivation

As we mentioned previously, the element view captures architectural design decisions on components and connectors we need in the architecting process. Since an MSC is associated with several agent instances, we can derive the element view directly from the syntax of MSCs.

Specifically, each agent instance is taken as a computation element, which includes its services or interfaces according to requirement specifications. Besides, from the interaction messages between the source and target agent instances, we can extract data elements that 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}

From the syntax of MSCs, the element view is initially documented. When new scenarios are introduced by end-users, the element view is then evolved and refined based on updated MSCs.

3.2.3 Constraint View Derivation

Based on the semantics of MSCs, we analyze behavior, properties, and relationships of the goal system, in order to document architectural design decisions in the constraint view.

In terms of behavior, we focus on general functionality of the system that is specified by the end-user scenarios, i.e., the prescriptions and the proscriptions. Typically, in the end-user scenarios, positive scenarios describe the desirable behavior of the system, while negative scenarios describe the undesirable behavior. Therefore, we can tell what the system should do from positive scenarios, and what should not do from negative scenarios as well as exceptions handled in the MSCs. Through this information, the architectural design decisions on the behavior of the system are documented by the following steps:

Behavior = {Prescriptions; Proscriptions}

Prescriptions = {Positive Scenarios}

Proscriptions = {Negative Scenarios; Exceptions}

Properties in the constraint view mean the constraints on a single element. We use “Receive”, “Issue”, and “Check” factors to define properties.

Properties = {Receive; Issue; Check}

“Receive” and “Issue” factors identify the responsibility of each element. For a computation element, “Receive” factor indicates the data which inputs to the element, and “Issue” factor means the data which outputs from the element. Both of them are achieved according to the message interactions

in the MSCs. If the element is a data element or a connector element, the "Receive" and "Issue" factors are specified as the corresponding computation elements directly operating the data element or connected by the connector element. "Check" factor is the precondition and the postcondition for an element according to requirement specifications. Generally, properties capture architectural decisions for a single element, through which we are able to grasp the responsibility of the element and the requirement constraints on the element.

Relationships are architectural design decisions on interactions and configurations among different elements. In order to find out the interactions among agent instances, we use simple path expressions to illustrate the interacted events in the MSCs.

Relationships = {Event Traces by Path Expressions}

The event traces provide us with general information about the interaction among agent instances. Based on the event traces results, the couplings and the structure of the components are obtained. Additionally, interactions and configurations among different elements provide a blueprint for us to choose architectural styles and patterns for subsequent architecting and designing process.

3.2.4 Intent View Derivation

Documenting the intent, i.e., decision making strategy, is necessary for communicating clearly among different stakeholders and keeping architectural knowledge complete in the software development life cycle. Since decision mak-

ing strategies are usually behind architects and other stakeholders' thoughts, the intent view cannot be derived and evolved directly from MSCs as the element and constraint view, 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 architectural design decisions in the intent view. For instance, we may document the motivations why we choose some elements as computation elements while others as connector elements, and the reasons that we put a certain property on an element, etc. Basically, rationale evolves together with the element view and the constraint view. When the decisions in the element and constraint view change, the documented rationale is to be updated as well in order to keep the architectural knowledge up-to-date.

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. At the same time, the justifications, alternatives, and trade-offs generated when selecting a certain best-practice during the decision making process are documented in the rationale as well.

In conclusion, the intent view are documented in two aspects:

Rationale = {Answers or Solutions to The Intent-Related Questions}

Best-Practices = {Architectural Styles; Architectural Patterns; Design

Patterns}

The intent view is as important as the element and constraint view, and is critical for architectural knowledge management. Therefore, when we update the element view and the constraint view according to the changing requirements, it is necessary to update the intent view as well.

Chapter 4

Case Study

In order to evaluate the applicability and the effectiveness of the TVM and the SceMethod, we conduct a case study on an industrial project. This chapter first presents the background of our case study, and then describes research questions, end-user scenarios, results, analysis, and discussion respectively.

4.1 Background

Our TVM and SceMethod have been validated in a substantial case study on an industrial project provided by the Italian electrical company ENEL [1]. In this project, an information system is designed to manage ENEL's thermal power plant operations. The purpose of the project aims to improve power plant efficiency, to reduce operation and maintenance costs, and to avoid forced outages [33]. Therefore, a power plant monitoring system is to be established with functions such as data acquisition from the field through sensors, fault detection in the power plant, and alarm raising in case of fault occurred. The main requirements of the system are gathered from [11], [13], [14].

Perry and Brandozzi have presented a method that transforms goal oriented requirement specifications into architectural prescriptions [6], [7]. The power plant monitoring system has already been applied in a case study by using Perry/Brandozzi's method [18]. We conducted the case study on the same real world project. On the one hand, we assessed the applicability of the TVM and the SceMethod for a real industrial project; on the other hand, we further evaluated the effectiveness of the TVM and the SceMethod by comparing our results with those in the previous case study which used Perry/Brandozzi's method.

4.2 Research Questions

The TVM and the SceMethod provide a general architecture framework and a complete process to support the documentation and evolution of architectural design decisions. This leads to the following research questions:

RQ1: Are the TVM and the SceMethod feasible when applied to real scenarios in an industrial project context?

RQ2: How well do the architectural design decisions derived from the SceMethod cover the main architectural specifications and issues?

RQ3: How well do the derived results on architectural design decisions support architecture evolution?

We conduct a case study to address these questions. In the following sections, we describe our end-user scenarios, results, analysis, and discussion

respectively.

4.3 End-user Scenarios

Based on the requirement specifications of the power plant monitoring system, we established end-user scenarios to cover the functionality of the system, including all the positive scenarios and some of the negative scenarios. Figure 4.1 and Figure 4.2 show the MSC specifications for the positive and negative scenarios of the power plant monitoring system.

4.4 Results

Taking the MSC specifications as the input, we followed the SceMethod to derive the architectural design decisions of the power plant monitoring system.

4.4.1 Element View

From the syntax of the MSCs in Figure 4.1 and Figure 4.2, all the agent instances are considered as the computation elements, and the information transmitted by the interaction messages are the data elements. We defined four connector elements as the channels between the source and target computation elements. Table 4.1 shows the element view of the power plant monitoring system.

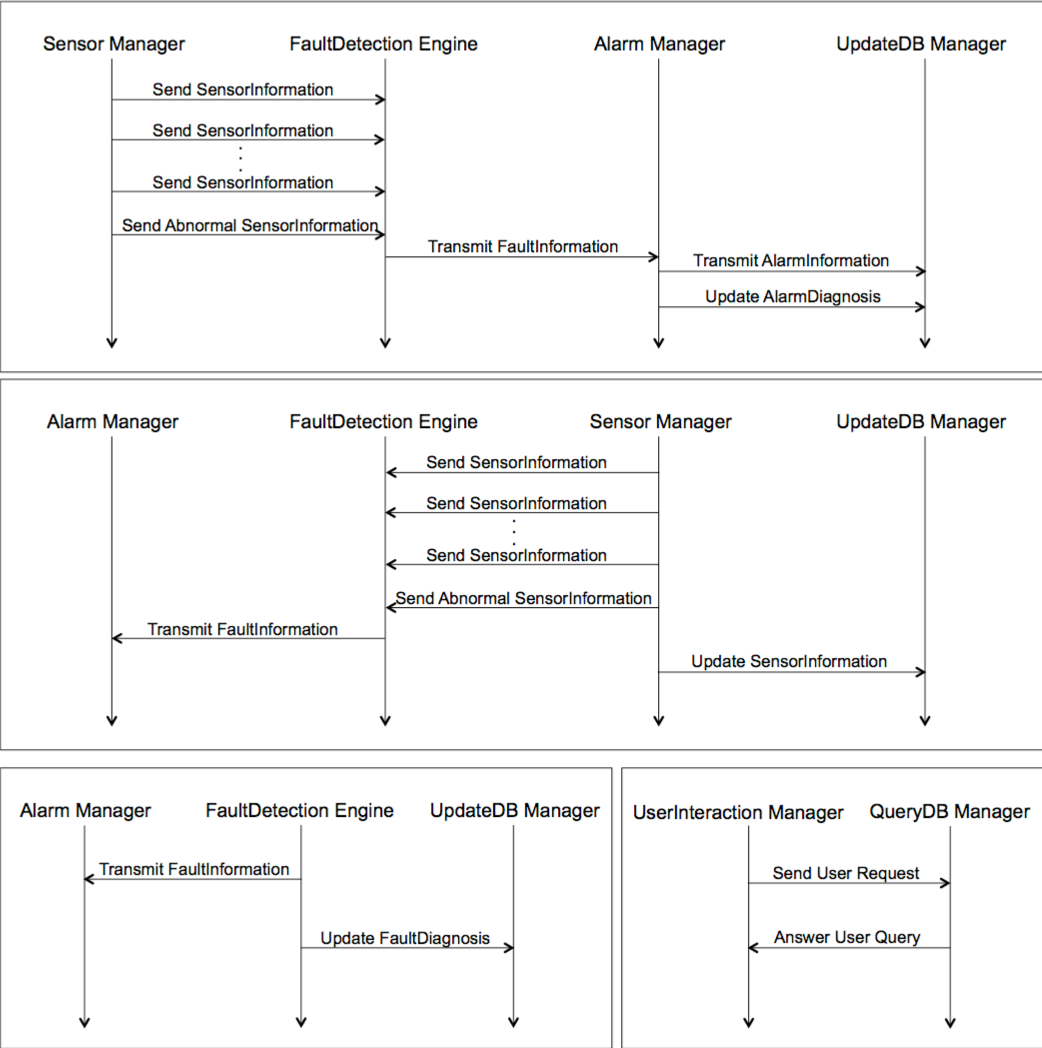

Figure 4.1: MSC Specifications of the Power Plant Monitoring System (positive scenarios)

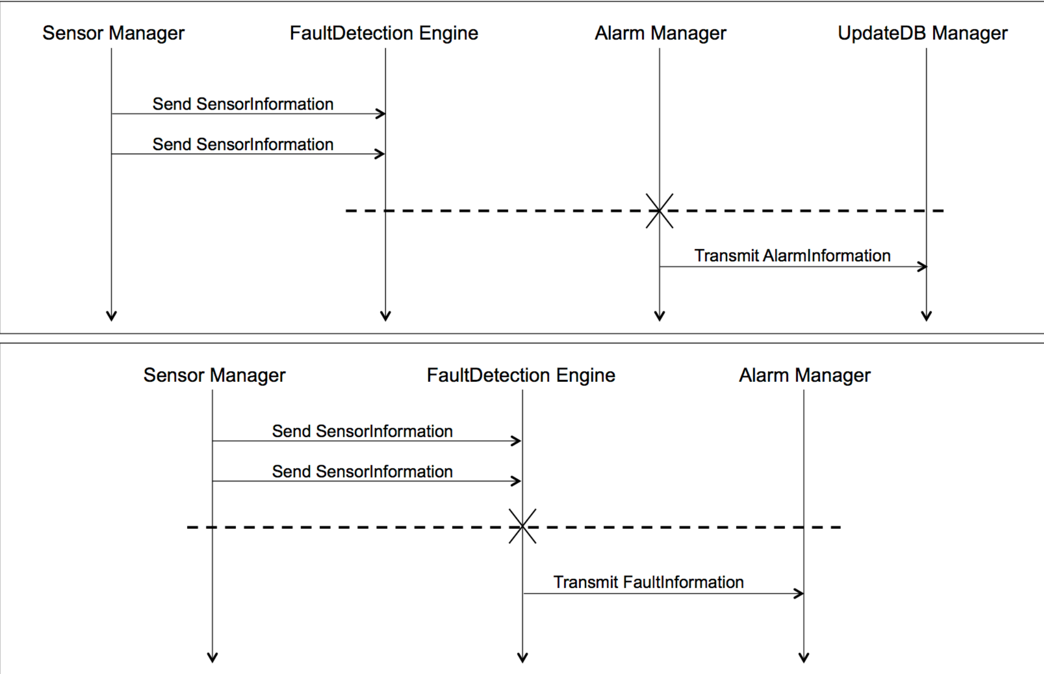

Figure 4.2: MSC Specifications of the Power Plant Monitoring System (negative scenarios)

Table 4.1: The Element View Results

Category	Element
Computation Elements	Sensor Manager
	FaultDetection Engine
	Alarm Manager
	UpdateDB Manager
	UserInteraction Manager
	QueryDB Manager
Data Elements	Sensor Information
	Fault Information
	Alarm Information
	Alarm Diagnosis
	Fault Diagnosis
	User Request
	Query Answer
Connector Elements	Sensor Connector
	FaultDetectionAlarm Connector
	UpdateDB Connector
	QueryDB Connector

4.4.2 Constraint View

From the semantics of the MSCs, we derived architectural design decisions on behavior, properties, and relationships of the power plant monitoring system. First of all, we focused on the behavior of the system. The positive and the negative scenarios tell the system behavior, and each conclusion we draw from the end-user scenarios can be seen as an architectural design decision on system behavior. Such as “when the Alarm Manager receives fault information, it should send alarm information to the UpdateDB Manager to update the database” and “If the FaultDetection Engine does not receive abnormal sensor information, it should not release fault information”. The architectural design decisions relevant to the system behavior provide us general functionality of the power plant monitoring system, based on which we find out the detailed system architecture through further analysis.

Secondly, we documented the properties of each element in the element view. The results are shown in Table 4.2. From these results, the responsibility of each element enables us to extract the requirement constraints (precondition and postcondition) that we need to comply with in the later architecting and designing process.

As for relationships among different elements, we obtained each event trace from the MSC specifications of the system. One example of the event trace is:

Table 4.2: The Properties Results For The Constraint View

Elements	Receive	Issue	Check
Sensor Manager (S_M)	Field Data	S_I	Data Correctness
FaultDetection Engine (FD_E)	S_I	F_I, F_D	Sanity, Consistency
UpdateDB Manager (UDB_M)	A_I, A_D, S_I, F_D	-	-
UserInteraction Manager (UI_M)	User Operations	U_R	-
QueryDB Manager (QDB_M)	U_R	Q_A	-
Sensor Information (S_I)	S_M	FD_E	Sanity, Consistency
Fault Information (F_I)	FD_E	A_M	Fault Detected
Alarm Information (A_I)	A_M	UDB_M	Fault Detected
Alarm Diagnosis (A_D)	A_M	UDB_M	Alarm Transmitted
Fault Diagnosis (F_D)	FD_E	UDB_M	Fault Detected
User Request (U_R)	UI_M	QDB_M	-
Query Answer (Q_A)	QDB_M	UI_M	-
Sensor Connector (S_C)	S_M	FD_E	Data Correctness
FaultDetectionAlarm Connector (FDA_C)	FD_E	A_M	Sanity, Consistency
UpdateDB Connector (UDB_C)	S_M, FD_E, A_M	UDB_M	Secure, TimeConstraint=2s
QueryDB Connector (QDB_C)	UI_M	QDB_M	TimeConstraint=5s

S_M : send abnormal sensor info

→ FD_E : transmit fault info

→ A_M : transmit alarm info

→ UDB_M

Based on all the event traces from the end-user scenarios, we captured the coupling relationship among the computation elements, data elements, and connector elements. Structure diagram is the best way to show how each element related with others to establish the complete architecture. We illustrated the structure diagram of the power plant monitoring system in Figure 4.3, which is generated from the event traces.

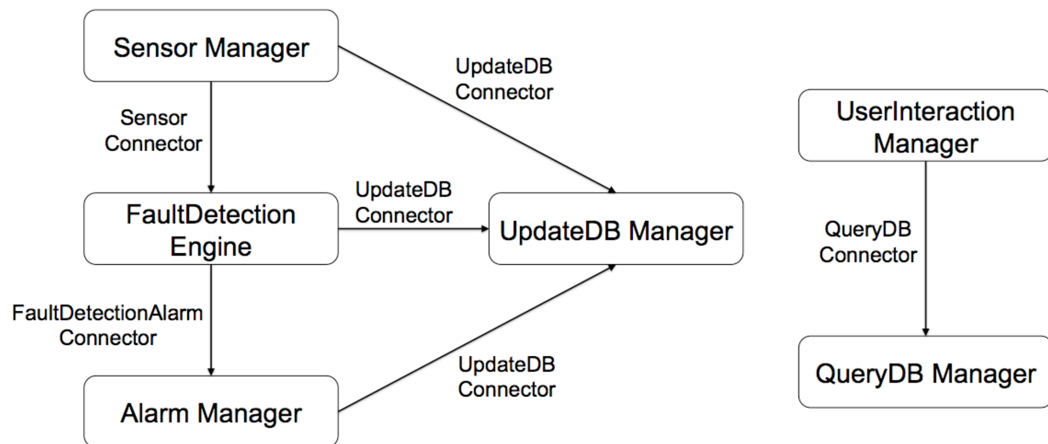

Figure 4.3: The Structure Diagram of The Power Plant Monitoring System

4.4.3 Intent View

Since the intent view reflects the thoughts behind stakeholders' head during the architecting process, as mentioned previously, we documented the

answers to the questions that concerned with the decision making process as architectural design decisions.

We did not specify all the possible decisions in the intent view. We only illustrated some questions as examples here, which are shown in Table 4.3. Answers to these questions provide us with the intent during the architecting process.

Table 4.3: Questions For Establishing The Intent View

Rationale	(**Motivation**) What is the motivation to establish the monitoring system?
	(**Alternatives**) How can we get the six computation elements?
	(**Reasons**) Why do we need the computation element "FaultDetection Engine"?
	(**Trade-offs**) What is the trade-off between using "Sensor Manager" or not?
	(**Justifications**) How to justify "Alarm Manager" works according to the requirements?
	⋮
Best-Practices	(**Architectural styles**) What kind of architectural style we can use to establish the system?
	(**Architectural patterns**) Is the layers architectural pattern applicable to the system?
	(**Design patterns**) Is there any design pattern we can adopt to design the system?
	⋮

4.5 Analysis

RQ1: Are the TVM and the SceMethod feasible when applied to real scenarios in an industrial project context?

The power plant monitoring system is an industrial project that supported by the Italian company ENEL. We note that after we have described the end-user scenarios based on the requirement specifications of the system, it is easy to apply the TVM and the SceMethod to those scenarios to derive most of the architectural design decisions. Basically, the end-user scenarios specified by MSCs enable us to obtain the element view, the constraint view, and the intent view respectively according to the SceMethod.

RQ2: How well do the architectural design decisions derived from the SceMethod cover the main architectural specifications and issues?

Table 4.1 shows all the components and connectors that we need to establish the power plant monitoring system. Comparing with the previous case study by using Perry/Brandozzi's method on the same system, we find that the elements generated from the SceMethod have covered all the process components, data components, and connectors from Perry/Brandozzi's method [18]. However, there is a little difference that we have one more computation element, i.e., the Sensor Manager, in our element view. Because by providing more computation elements, we can make the architecture more flexible, which helps to support detailed functionality and is also easier for us to manage the coupling and the evolution of the architecture. Table 4.2

indicates that the properties enable us to clarify the responsibility of each element and the requirement constraints that need to be considered in the future designing process. In addition, in order to establish a whole blueprint of the goal system, we generate Figure 4.3 based on the relationships among all the computation and connector elements, which is similar as the box diagram in the architecture results using Perry/Brandozzi's method [18]. Note that the architectural decisions derived from the SceMethod have covered all the architecture prescriptions from Perry/Brandozzi's method, and in our case study, the main issues on the components, connectors, and their relationships have been achieved as well when deriving architectural design decisions. Furthermore, we captured all the possible intent-related design decisions, which are then used to record and track the architectural knowledge and the decision making process during the architecting phase. On the contrary, the intent-related decisions were not mentioned in Perry/Brandozzi's method.

RQ3: How well do the derived results on architectural design decisions support architecture evolution?

The architecture derivation process is basically an evolutionary process. Since architecture is regarded as a set of architectural design decisions, we primarily analyze the evolution of architectural design decisions to manage architecture evolution. The initial architectural design decisions results largely cover the functional requirements of the power plant monitoring system, from which we can obtain the architecture blueprint of the system. During the evolutionary change, the architectural decisions in the elements, the constraints,

and the intent view should be tracked and updated with the changing scenarios and requirements. Here, we take the constraint view evolution as an example. For the constraint view, non-functional requirements influence the properties of the elements, and they may be changed after the components, the connectors, and the structure diagram of the system are derived. Specifically, as the architecting process proceeds, some quality attributes, e.g., reliability requirements, are more crucial for the whole system, and adding these quality requirements will make the system more realistic. For instance, we have basic requirement constraints between the FaultDetection Engine and the Alarm Manager in the initial architecture, and some new reliability requirements are added to the system afterward. One requirement may be "once a fault is detected by the FaultDetection Engine, the alarm should be raised within 5 seconds". When this new limitation is included in the requirement specifications, we need to find out how it affects the current design decisions results. Based on the TVM, we find that the element view does not change, since there is no change on the syntax of the end-user scenarios. However, the constraint view is to be updated, because the "Check" factor of the property for the FaultDetectionAlarm connector should comply with the new requirement specification, i.e., we need to add "TimeConstraint=5s" to the "Check" factor. Most of the time, the intent view evolves together if the element view or the constraint view changes. Hence we also need to document the reason or the justification in the intent view, in order to specify why the time constraint should be within 5 seconds for the FaultDectection Alarm connector.

Generally, the architecture evolution process is based on the initial architectural design decisions results. When new requirements or new decisions via end-user scenarios arrive, we apply the SceMethod to the changing information to evolve the initial decisions. The SceMethod ensures that the architecture evolution results are consistent with the changing requirement specifications, and keeps architectural knowledge complete in the changing environment.

4.6 Discussion

4.6.1 Practicality

The TVM and the SceMethod, which are applied during the architecting and designing process, enable us to capture architectural design decisions and manage their evolution. As the software life cycle proceeds, the architectural design decisions results are widely employed throughout the entire software development process. Specifically, the documentation on architectural design decisions intuitively reflects development artifacts, such as the decisions in the element view, which trigger the implementation of the particular classes in the development phase. Furthermore, the constraint view brings benefits to system testing and system configuration, since the decisions on properties and relationships enable us to define effective test cases and system configuration framework. The architectural knowledge is also important for training and project management by providing efficient understanding among different stakeholders in the software development life cycle.

By applying the TVM and the SceMethod, the architectural design decisions are employed in most of the software development phases, and finally architectural knowledge is well incorporated in various levels of the software development process.

4.6.2 Scalability

In the case study, we applied the TVM and the SceMethod to the power plant monitoring system and it worked well. As the system become more complex, for instance, more requirements need to be considered, our method can be applied incrementally. Each time we obtain new requirements, we describe them as scenarios by MSCs, and then follow the process of the SceMethod to derive the newly architectural design decisions. Our method right now is not quite applicable to distributed system, because the decision-collection mechanism in the SceMethod does not support for distributed environment. We try to improve this by providing tool support as integrating the SceMethod into configuration management tools, in order to better support the application and the management of architectural decisions for complex systems.

4.6.3 Limitations

One limitation of the TVM and the SceMethod is lack of automatic traceability from architectural design decisions to requirement specifications. The automatic traceability between requirement and architectural knowledge will be more efficient when considering large-scale software systems, which have

larger architectural design decisions set and more difficult to trace by hand. Therefore, tool support of the TVM and the SceMethod is also necessary to manage the traceability. Moreover, it may be useful to include a status for each decision to support the traceability. Another limitation is that current architectural design decisions results do not show the relations among each decision, and thus cannot provide in-depth architectural knowledge information. We aim to overcome this limitation by creating a network of the design decisions, through which we are capable of looking into further relationships of each decision, such as the cause and effect influence among them.

Chapter 5

Related Work

The key concepts of the traditional view on software architecture are components and connectors [26], [3]. Nowadays, software architecture has been seen as a set of architectural design decisions [4], [19], [30]. The architectural decisions in the software architecting process are increasingly focused by researchers and practitioners [16], [22], and architectural design decisions are also considered to be a part of architectural knowledge [23]. In [15], 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 has been provided in [12], [17], however, those documentation approaches do not explicitly capture architectural design decisions in the architecting process. Recently, many models and tools have been proposed for capturing, managing, and sharing architectural design decisions.

Tyree's template [32] provides a simple document describing key architectural decisions, which establishes a concrete direction for design and implementation, and also clarifies the rationale for different stakeholders. In

[23], an ontology of architectural design decisions and their relationships have been described. This ontology then can be used to construct architectural knowledge of a software system. ADDSS [9] is a web-based tool for documenting architectural design decisions. It establishes the backward and forward traceability between requirements, decisions, and architectures. Archium [20] is a Java tool, including a complier and a run-time environment, for supporting architectural design decisions capturing, tracing, and managing. It also provides visualization for design decisions by using a dependency graph, which is easy for stakeholder to evaluate and track the decisions. Other models and tools such as AREL [29] and PAKME [2] are also proposed for managing architectural knowledge.

A detailed comparison of these existing models and tools has been done in [28]. Since each model has its own strong and weak points, it is still difficult for researchers and practitioners to choose which one is more suitable for their architecting process, and the existing models are hard to support architecture evolution very well [10]. Perry and Grisham have focused on architecture and design intent in [25], and our work in this thesis tries to further generalize the concept of the intent and architectural decisions in software architecture and its evolution. Our TVM intends to provide a general architecture framework to clarify the notion of architectural design decisions, and the triple views perfectly cover the key features in software architecture. In addition, the SceMethod based on the TVM gives a simple and consistent way to manage the documentation and the evolution of architectural design decisions, which

is effective in operating and maintaining the architecting process in a changing software development context.

Chapter 6

Extension for Future Work

6.1 Basic Idea

A recent phenomenon in the evolution of software development strategies is that of encouraging external software developers to become involved in software development. These third parties make their contributions to software development and software organizations realize intrinsic benefits. This significant shift in traditional software development process has resulted in a new software development paradigm called "software ecosystems". The adoption of the software ecosystem approaches establishes a new area in software engineering research and practice. Basically, in a software ecosystem, software organizations have broken their organization boundaries, and different parties collaborate under a common architecture and within a social networking context to achieve innovation. Therefore, the traditional closed software development has changed to open software development.

The current approach to managing architectural design decisions within a software organization for single product development may not be applicable for software ecosystems. The popularity of software ecosystems forces researchers and practitioners to reconsider how to manage architectural knowl-

edge in open software development, since architectural design decisions should be shared not only within the organization but also with external parties. Thus, a number of challenges of managing architectural design decisions in a software ecosystem platform will arise, and it is important to find a way for effectively managing architectural knowledge in order to adapt the increasing openness and interoperability in the software community. So far, little work has been done in this area to the best of our knowledge.

Hence, the research question that will need to be addressed is: How to manage architectural design decisions in software ecosystems to adapt to collaboration and openness in software development. In order to manage architectural design decisions in software ecosystems, models and tools should be capable of capturing and representing decisions not only in an organization's architecting process, but also among those external parties in the social community. Here, we analyze the characteristics of software ecosystems in order to obtain a deeper insight into architectural knowledge for a software ecosystem.

6.2 Software Ecosystems Characteristics

A software ecosystem is defined as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them [21]. Compared with the traditional software engineering process, software ecosystems have the following characteristics:

- **A social community**. In a software ecosystem, third parties are encouraged to contribute to an organization's product development, which establishes a social network that includes not only the team within the organization but also external developers, sharing technologies, skills, knowledge and even issues in the network. This further accelerates social interactions among the organization and the external parties, and forms a software social community.

- **Extensive business innovation**. Innovations are always used to illustrate the capability of an organization to be creative in product development [8]. In a software ecosystem, both the employees in the organization and the third parties have opportunities to provide innovative ideas to solve business problems, which extends the organization's innovative strategy and supports both reactive and proactive business innovations [8].

- **Architecture platform commonality and variability**. The concept of a software ecosystem focuses on multiple product development achieved by sharing a common architecture platform in open software development. However, software organizations also need to provide stable interfaces through the architecture platform to external developers, without disabling the operation of externally developed applications on top of the platform [5]. The concern of the architecture in a software ecosystem is to manage its commonality and variability to suit different business entities.

- **Diverse resources management**. In a software ecosystem, both employees in an organization and external developers share community resources which include not only technical resources in the development but also tacit knowledge behind the thoughts of all parties. Thus, resources management is required to deal with the diverse resources distributed in multiple development and multiple stakeholders, which in uences the decision-making processes and the corresponding architectural knowledge.

Due to the aforementioned characteristics, it is more difficult to document architectural design decisions in software ecosystems than in single product development. Basically, some key aspects of architectural design decisions in open software development should be identified.

For single product development, we have proposed the element view, the constraint view, and the intent view constitute a complete architectural design decisions set. When applied to software ecosystems, these three views are still able to document the most fundamental design decisions. However, the openness and the sociability of a software ecosystem bring us new challenges of further capturing architectural decisions influenced by a software social community. We argue that a new architectural design decisions set for software ecosystems should be established, including basic architecture elements, properties, and relationships that form a common architecture platform, and also decision-making strategies in the social community that support multiple de-

velopment and communication. Additionally, new strategies to ensure consistent communication should be developed for sharing architectural knowledge in a software ecosystem.

6.3 Open Challenges

We summarize major challenges of developing new technologies and tools for managing architectural design decisions in software ecosystems.

- **Comprehensive definition**. Aiming to identify and manage effectively architectural knowledge, a definition of what should be considered as architectural design decisions in a software ecosystem is firstly required. The existing definition for architectural design decisions may not be suffcient to meet software ecosystem requirements.
- **Multi-level communication**. Sharing and communicating architectural design decisions within an organization, between an organization and external developers, and among third parties are all necessary in a software ecosystem. Therefore, how to keep architectural knowledge consistent in a complex distributed and communicating environment should be addressed.
- **Completeness**. Our work on the Triple View Model (TVM) helps to document complete architectural design decisions in single product development. However, for complete architectural knowledge representation

in a software ecosystem approach, an adequate model and tool support are still needed.

- **Knowledge gain and evolution**. Since different parties contribute to multiple product development, models and tools for architectural knowledge should address scalability issues as the amount of decisions increases. This further accelerates the evolution of architectural design decisions, which could be another potential issue in software ecosystem approaches.

- **Traceability**. Efficient automatic traceability between system drivers (such as requirements, business and market needs) and architectural design decisions is necessary for a large architectural design decisions set in software ecosystems, and requires extensive research on knowledge traceability.

Chapter 7

Conclusion

A recent strand of software architecture research is that software architecture is considered as a set of architectural design decisions. Architectural design decisions are also defined as a part of architectural knowledge, and are necessary to be documented and managed in order to control fundamental problems in the software life cycle.

In this thesis, we discussed the documentation and evolution of architectural design decisions. We proposed the Triple View Model (TVM) as a general architecture framework, which includes an element view, a constraint view, and an intent view to indicate "what"-"how"-"why" features for architectural design decisions. Based on the TVM, we presented a scenario-based methodology (SceMethod) for architectural design decisions documentation and evolution. In the SceMethod, we obtained and specified the element view, the constraint view, and the intent view through end-user scenarios, which are represented by Message Sequence Charts (MSCs). When applying this method for the first time, we obtained initial architectural design decisions results. Later on, as requirements change, the initial architectural decisions are evolved and refined according to the newly requirements.

We also conducted a case study on an industrial-strength project to validate the applicability and the effectiveness of the TVM and the SceMethod. The results show they provide complete documentation on architectural design decisions for creating a system architecture, and well support architecture evolution with changing requirements.

Bibliography

[1] ENEL page. http://www.enel.com/en-GB/.

[2] M. A. Babar and I. Gorton. A tool for managing software architecture knowledge. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, SHARK-ADI '07, pages 11–17, Washington, DC, USA, 2007. IEEE Computer Society.

[3] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[4] J. Bosch. Software architecture: The next step. In F. Oquendo, B. Warboys, and R. Morrison, editors, *EWSA*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2004.

[5] J. Bosch. Architecture challenges for software ecosystems. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 93–95, New York, NY, USA, 2010. ACM.

[6] M. Brandozzi. Transforming goal oriented requirements specifications into architectural prescriptions. In *Proceedings STRAW'01, ICSE 2001*, pages 54–61, 2001.

[7] M. Brandozzi and D. E. Perry. Architectural prescriptions for dependable systems. In *ICSE 2002 Workshop on Architecting Dependable Systems*, 2002.

[8] P. R. J. Campbell and F. Ahmed. A three-dimensional view of software ecosystems. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 81–84, New York, NY, USA, 2010. ACM.

[9] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes*, 31, September 2006.

[10] R. Capilla, F. Nava, and A. Tang. Attributes for characterizing the evolution of architectural design decisions. *Software Evolvability, IEEE International Workshop on*, 0:15–22, 2007.

[11] E. Ciapessoni, P. Mirandola, A. Coen-Porisini, D. Mandrioli, and A. Morzenti. From formal models to formally based methods: an industrial experience. *ACM Trans. Softw. Eng. Methodol.*, 8:79–113, January 1999.

[12] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2002.

[13] A. Coen-porisini and D. Mandrioli. Using trio for designing a corba-based application. *Concurrency and Computation: Practice and Experience*,

12:981–1015, 2000.

[14] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli. A formal approach for designing corba-based applications. *ACM Trans. Softw. Eng. Methodol.*, 12:107–151, April 2003.

[15] R. C. de Boer and R. Farenhorst. In search of 'architectural knowledge'. In *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, SHARK '08, pages 71–78, New York, NY, USA, 2008. ACM.

[16] J. C. DueÃśas and R. Capilla. The decision view of software architecture. In *European Workshop on Software Architecture*, pages 222–230, 2005.

[17] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[18] D. Jani, D. Vanderveken, and D. E. Perry. Deriving architecture specifications from kaos specifications: A research case study. In R. Morrison and F. Oquendo, editors, *EWSA*, volume 3527 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2005.

[19] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[20] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, WICSA '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society.

[21] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 187 –190, may 2009.

[22] P. Kruchten, R. Capilla, and J. C. Dueñas. The decision view's role in software architecture practice. *IEEE Softw.*, 26:36–42, March 2009.

[23] P. Kruchten, P. Lago, and H. V. Vliet. Building up and reasoning about architectural knowledge. In *Quality of Software Architectures*, pages 43–58, 2006.

[24] D. E. Perry. Issues in architecture evolution: Using design intent in maintenance and controlling dynamic evolution. In *Proceedings of the 2nd European conference on Software Architecture*, ECSA '08, pages 1–1, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] D. E. Perry and P. S. Grisham. Architecture and design intent in component & cots based systems. In *Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, pages 155–, Washington, DC, USA, 2006. IEEE Computer Society.

[26] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, October 1992.

[27] D. M. A. Reniers. Message sequence chart: Syntax and semantics. Technical report, Faculty of Mathematics and Computing, 1998.

[28] M. Shahin, P. Liang, and M.-R. Khayyambashi. Architectural design decision: Existing models and tools. In *WICSA/ECSA*, pages 293–296. IEEE, 2009.

[29] A. Tang, Y. Jin, and J. Han. A rationale-based architecture model for design traceability and reasoning. *J. Syst. Softw.*, 80:918–934, June 2007.

[30] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[31] D. Tofan. Tacit architectural knowledge. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 9–11, New York, NY, USA, 2010. ACM.

[32] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Softw.*, 22:19–27, March 2005.

[33] D. Vanderveken and A. V. Lamsweerde. Deriving architectural descriptions from goal-oriented requirements models. Technical report, 2004.