Managing Architectural Design Decisions Documentation and Evolution

Meiru Che, Dewayne E. Perry

***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 paper, 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 methodology (SceMethod) to manage the documentation and evolution of ADDs. Furthermore, we also develop a UML metamodel that incorporates evolution-centered characteristics to manage ADDs evolution. We conduct a case study on an industrial project to validate the applicability and the effectiveness of the TVM, the SceMethod and the UML metamodel. The results show they provide complete documentation on ADDs for creating system architecture, and well support architecture evolution with changing requirements.

***Keywords*—**Architectural design decisions, Architecture documentation, Architecture evolution, Architectural knowledge, Scenario

I. 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 [1]. Although the achievement by recognizing components and connectors is significant in re-search 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 (ADDs), and the justification for these decisions to be found in the rationale. It was not until 2004, with Bosch's paper [2] at the European Workshop on Software Architecture, that software architecture has finally come to be considered as a set of ADDs. This specific focus on ADDs led to a broader focus on architectural knowledge [3]. Capturing and representing ADDs 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 ADDs [4]–[6]. A brief comparison and analysis of the existing models and tools has been conducted in [7]. 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 [8].

To address this need, we propose the Triple View Model (TVM) as a general architecture framework of ADDs. The TVM divides ADDs set into three different views, i.e., the element view, the constraint view, and the intent view. These three views specify ADDs by three aspects, "what", "how", and "why", and all the ADDs are regarded as a software architecture. In addition, based on the TVM, we propose a scenario-based methodology (SceMethod) for ADDs documentation and evolution, which enables us to manage architectural knowledge effectively. Furthermore, we also develop a UML metamodel for the TVM in order to manage architecture knowledge evolution. The core idea of the UML metamodel is to ensure that the evolution of ADDs can be captured and tracked properly, thus all the stakeholders can share the evolutionary architectural knowledge without evaporation. Several evolution-centered characteristics are incorporated into the metamodel to achieve this goal. We subsequently conduct a substantial case study to validate our TVM, SceMethod and UML metamodel.

We make the following three contributions:

1) The TVM - A general framework of ADDs. The "what" - "how" - "why" triple view clarifies the notion when documenting ADDs;

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

Manuscript received March 25, 2012. This work was supported in part by the NSF under Grant Nos. IIS-0438967 and CCF-0820251.

Meiru Che is with department of Electrical and Computer Engineering, the University of Texas at Austin, Austin, TX 78712 USA (e-mail: meiruche@utexas.edu).

Dewayne E. Perry is with department of Electrical and Computer Engineering, the University of Texas at Austin, Austin, TX 78712 USA (e-mail: perry@mail.utexas.edu).

3) The UML metamodel - A fine-grained definition for the TVM, which designs each ADD category in the TVM as a UML class with multiple attributes to describe the decision information. The evolutionary characteristics in the UML metamodel helps to capture architectural knowledge for managing the architectural design decision evolution;

4) The substantial case study - A validation for the TVM, the SceMethod, and the UML metamodel on an industrial project. The results demonstrate the applicability and the effectiveness of them.

II. TRIPLE VIEW MODEL

The Triple View Model provides a fundamental framework of ADDs and covers key features of the architecting process [9]. It has the following advantages:

First, the TVM captures ADDs 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 ADDs documentation and evolution, and finally supports software architecture evolution.

A. Framework

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. Each view in the TVM is a subset of ADDs, and the three views constitute an entire ADDs set. Specifically, the three views mean three different aspects when creating an architecture, i.e., "what", "how", and "why", as shown in Fig. 1. The three aspects aim to cover design decisions on "what" elements should be selected in software architecture, "how" these elements combine and influence each other, and "why" a certain decision is made.

During the architecting process in the software life cycle, architects are the main role operating ADDs. However, programmers, project managers, or customers in the real software project environment may be brought forward architectural decisions as well. In any case, the TVM provides a right selection of ADDs, and it is applicable for all stakeholders. Moreover, the TVM suggests a systematical way to include complete architectural decisions for creating software architecture. Fig. 2 shows the relations among ADDs, the TVM and software architecture in a system.

B. Model

Here, we discuss the detailed contents of each view in the Triple View Model, which are illustrated in Fig. 3.

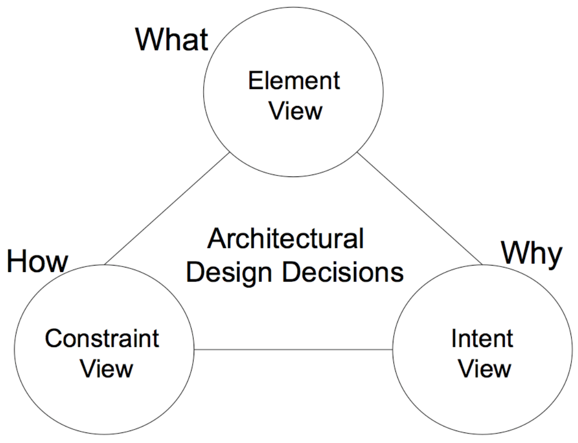


Fig. 1 Triple View Model Framework

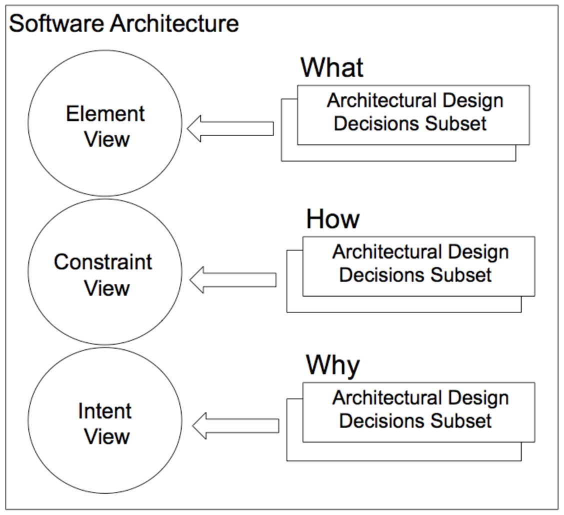

Software Architecture
What
Element View
Architectural Design Decisions Subset
How
Constraint View
Architectural Design Decisions Subset
Why
Intent View
Architectural Design Decisions Subset

Fig. 2 Triple View Model and Software Architecture

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, 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 ADDs in the element view consist of traditional architecture concepts, which are mainly represented by components and connectors.

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, 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.

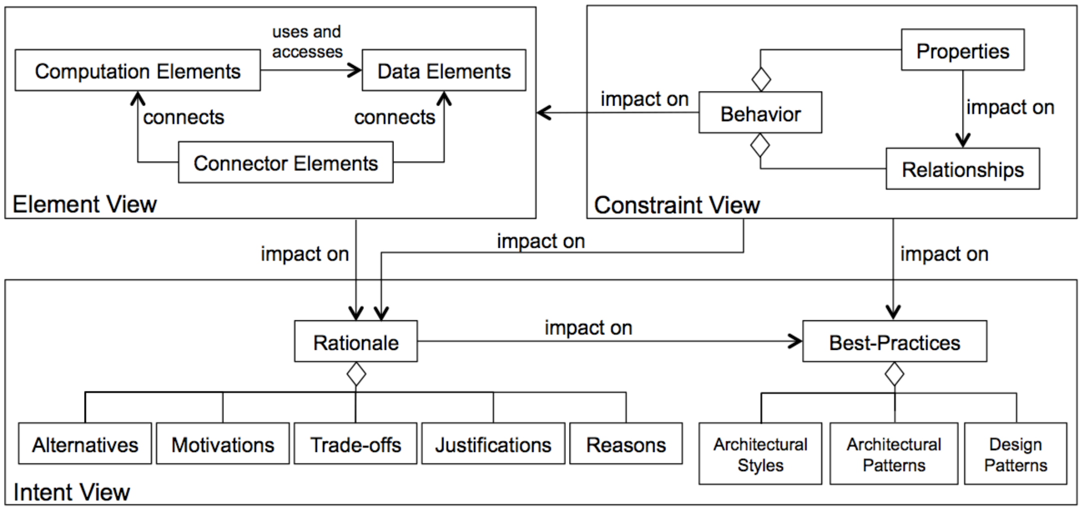


Fig. 3 Triple View Model for Architectural Design Decisions

The ADDs 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 [10], 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.

III. Scenario-Based Documentation and Evolution Method

In this section, we propose the scenario-based ADDs documentation and evolution method (SceMethod).

The TVM is the foundation of ADDs 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 of the 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. Fig. 4 illustrates the SceMethod process. We can see that for the first time we apply this method, we obtain initial ADDs results. Later on, as the requirements change, the architectural decisions 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 makes architectural knowledge explicit and reduces architectural knowledge evaporation.

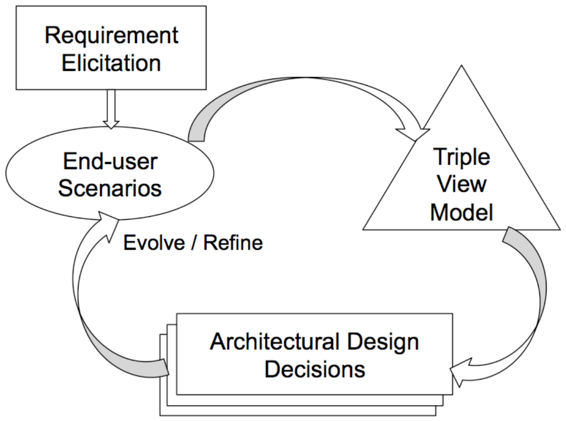


Fig. 4 The SceMethod Process

A. 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 [11], 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. Fig. 5 is a simple example of an MSC [11].

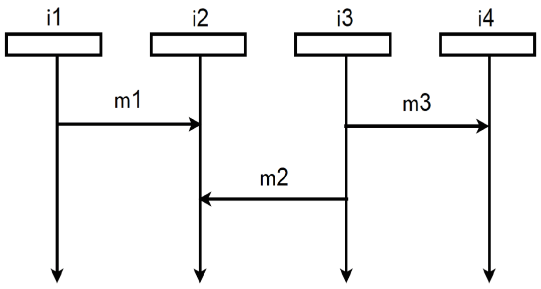


Fig. 5 An MSC example

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 Fig. 5, 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 ADDs 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.

B. From MSC Syntax to Element View

As we mentioned previously, the element view captures ADDs 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 end-users introduce new scenarios, the element view is then evolved and refined based on updated MSCs.

C. From MSC Semantics to Constraint View

Based on the semantics of MSCs, we analyze behavior, properties, and relationships of the goal system, in order to document ADDs 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 following steps document ADDs on the behavior of the system:

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 retrieved 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 pre-condition and the post-condition 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 ADDs 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 con- figurations among different elements provide a blueprint for us to choose architectural styles and patterns for subsequent architecting and designing process.

D. Intent View Documentation

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 making 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 ADDs 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.

IV. Evolution-Centered UML Metamodel

A. Metamodel

Based on the research work where we focused on managing the documentation and the evolution of ADDs, we develop a UML metamodel of our Triple View Model. The UML metamodel provides more detailed evolution-centered characteristics, which enable us to manage the evolution of architectural knowledge [12].

Figure 6, 7, and 8 illustrate the UML metamodel. In Figure 6, we can see that computation elements, data elements and connector elements in the element view are specified as classes where each of them has a bunch of attributes to describe its information. *Behavior* and *Relationship* classes describe the ADDs in the constraint view, and the architectural decisions for the "properties" of each element are merged into the corresponding element class as attributes. In Figure 7 and 8, ADDs on "Rationale" and "Best-Practices" are described as specific classes that extended the general *Rationale* and *Best-Practices* class.

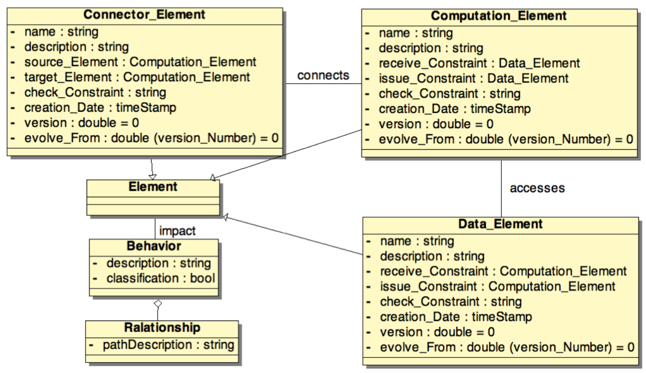

Connector_Element
- name : string
- description : string
- source_Element : Computation_Element
- target_Element : Computation_Element
- check_Constraint : string
- creation_Date : timeStamp
- version : double = 0
- evolve_From : double (version_Number) = 0
connects
Computation_Element
- name : string
- description : string
- receive_Constraint : Data_Element
- issue_Constraint : Data_Element
- check_Constraint : string
- creation_Date : timeStamp
- version : double = 0
- evolve_From : double (version_Number) = 0
accesses
Element
impact
Behavior
- description : string
- classification : bool
Ralationship
- pathDescription : string
Data_Element
- name : string
- description : string
- receive_Constraint : Computation_Element
- issue_Constraint : Computation_Element
- check_Constraint : string
- creation_Date : timeStamp
- version : double = 0
- evolve_From : double (version_Number) = 0

Fig. 6 Metamodel for the Element and the Constraint View

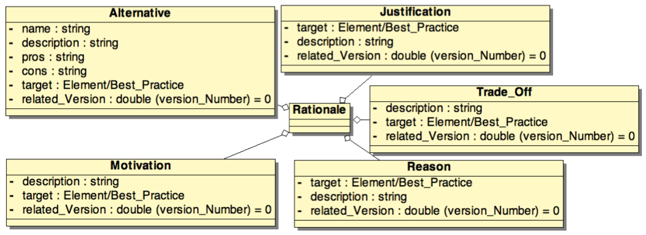

Alternative
- name : string
- description : string
- pros : string
- cons : string
- target : Element/Best_Practice
- related_Version : double (version_Number) = 0
Justification
- target : Element/Best_Practice
- description : string
- related_Version : double (version_Number) = 0
Rationale
Trade_Off
- description : string
- target : Element/Best_Practice
- related_Version : double (version_Number) = 0
Motivation
- description : string
- target : Element/Best_Practice
- related_Version : double (version_Number) = 0
Reason
- target : Element/Best_Practice
- description : string
- related_Version : double (version_Number) = 0

Fig. 7 Metamodel for Rationale in the Intent View

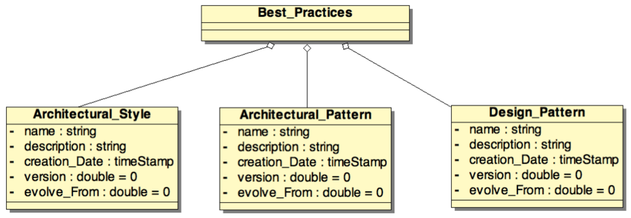


Fig. 8 Metamodel for Best-Practices in the Intent View

B. Evolution-Centered Characteristics

The metamodel aims to manage the evolution of ADDs. It has the following evolution-centered characteristics that enable us to make ADDs evolution explicitly, so that all the stakeholders can share architectural knowledge, specifically, decreasing the evaporation of the evolutionary knowledge in the software architecting process.

1) Evolution-Related Attributes

We define several evolution-related attributes to describe the ADDs classes in the metamodel. We use *creation_Date* to record the specific time stamp when a certain ADD is made. *Version* is used to specify a version number assigned to each ADD, which helps to manage multiple copies of a certain ADD during the evolutionary process. Moreover, we aim to record the evolution history by an attribute called *evolve_From*. When a new ADD is made that is evolved from an existing one, the *evolve_From* attribute is used to indicate the version of the previous ADD based on which the newly ADD is evolving. Another evolution-related attribute we propose is the *related_Version* for the Rationale classes, which is used to specify the version number of an ADD the rationale describes.

2) Traceable Evolution Chain

Besides the aforementioned evolution-related attributes, the UML metamodel provides a complete evolution chain for every ADD's evolutionary change, which enables us to keep tracking the evolution history of the ADDs set. Specifically, the *evolve_From* attribute provides a bridge to establish the evolution chain for ADDs. Through the evolution chain, the architect and other stakeholders are able to trace the changing information of a certain ADD, and they can share consistent architectural knowledge. Additionally, a traceable evolution chain keeps all the evolution history explicitly and hence significantly reducing the evaporation of the evolutionary architectural knowledge during the software development process.

3) *Version-Specific Rationale*

We can see that in all the rationale classes we have an attribute called *related_Version,* which is used to record the specific version number of an element or a best-practice that a rationale describes. The version-specific rationale classes provide us multiple ways of managing the evolutionary knowledge of ADDs by either tracking the rationale for a target ADD for its multiple versions or tracking the rationale for a certain version of an ADD. Thus, the tacit knowledge can be obtained according to the specific requirement on ADDs evolution.

V. CASE STUDY: VALIDATION IN A POWER PLANT MONITORING SYSTEM

Our TVM and SceMethod have been validated in a substantial case study on an industrial project provided by the Italian electrical company ENEL [13]. 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 [14]. 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 [15]–[17].

Perry and Brandozzi have presented a method that transforms the goal oriented requirement specifications into architectural prescriptions [18], [19]. The power plant monitoring system has already been applied in a case study by using Perry/Brandozzi's method [20]. 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 that used Perry/Brandozzi's method.

A. *Research Questions*

The TVM and the SceMethod provide a general architecture framework and a complete process to support the documentation and evolution of ADDs. 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 ADDs derived from the SceMethod cover the main architectural specifications and issues?

RQ3: How well do the derived results on ADDs support architecture evolution?

We conducted a case study to address these questions. We describe our end-user scenarios, results, analysis, and discussion respectively. In addition, we also applied the UML metamodel to this real project.

B. *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. Fig. 9 and Fig. 10 show the MSC specifications for the positive and negative scenarios of the power plant monitoring system.

C. *Results*

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

1) *Element View*

From the syntax of the MSCs in Fig. 9 and Fig. 10, 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 1 shows the element view of the power plant monitoring system.

Table. 1 The Element View Results

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

2) *Constraint View*

From the semantics of the MSCs, we derived ADDs 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 ADD 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 ADDs 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.

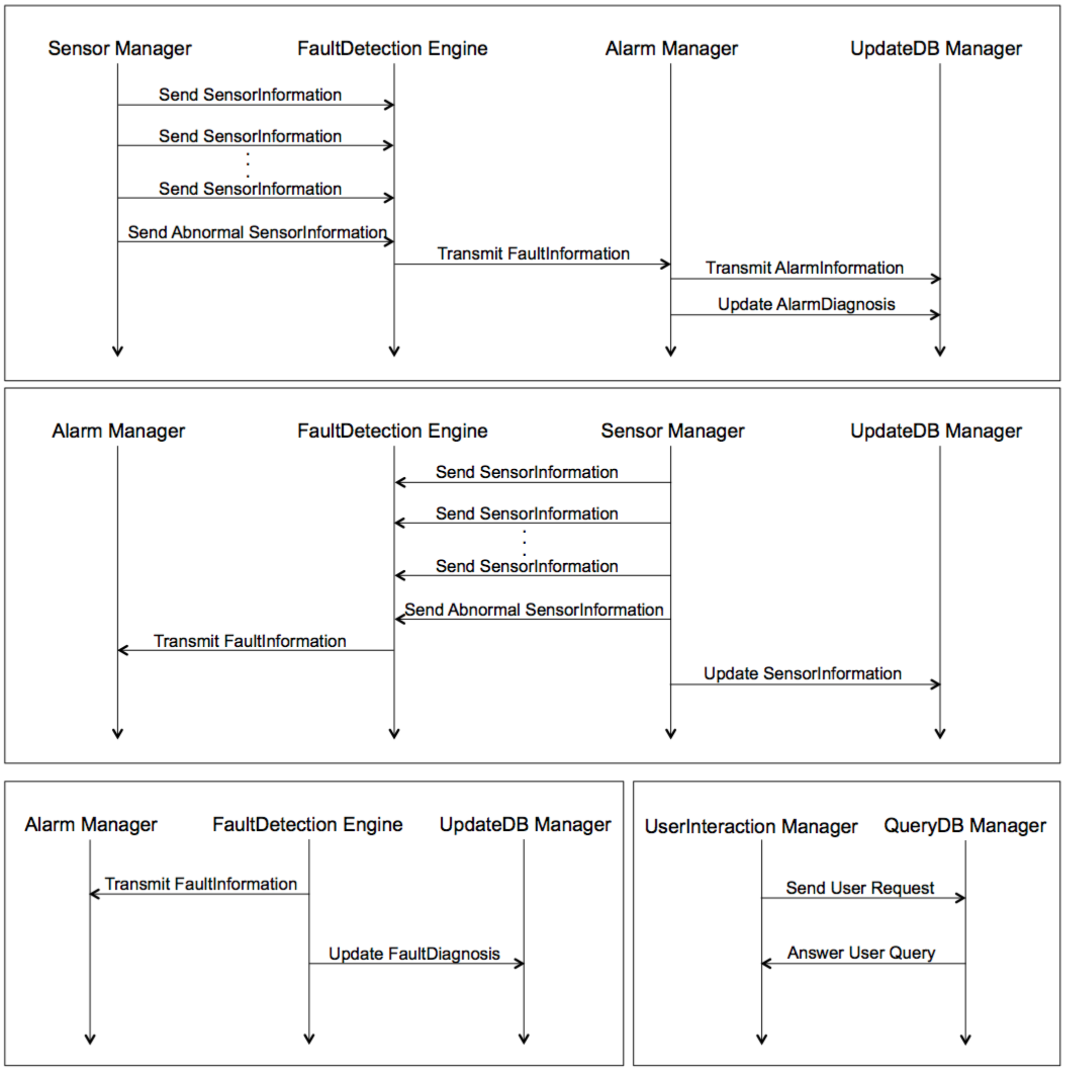


Fig. 9 MSC Specifications of the Power Plant Monitoring System (positive scenarios)

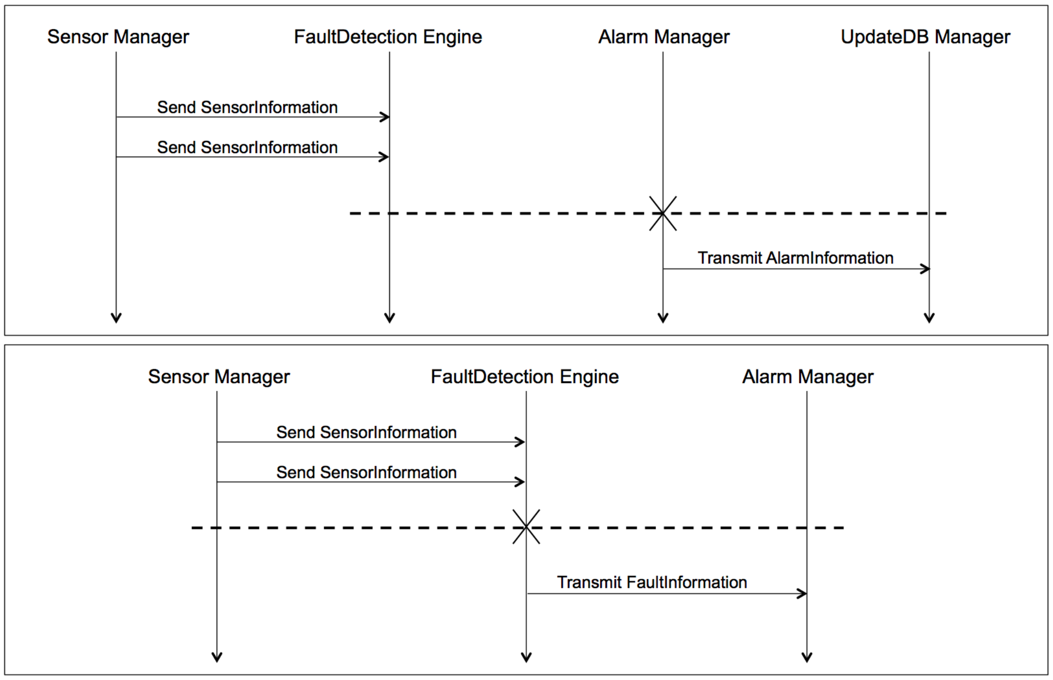


Fig. 10 MSC Specifications of the Power Plant Monitoring System (negative scenarios)

Secondly, we documented the properties of each element in the element view. The results are shown in Table 2. From these results, the responsibility of each element enables us to extract the requirement constraints (pre-condition and post-condition) that we need to comply with in the later architecting and designing process.

Table. 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
Alarm Manager (A_M)	F_I	A_I, A_D	Fault Detected
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

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:

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 Fig. 11, which is generated from the event traces.

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 ADDs.

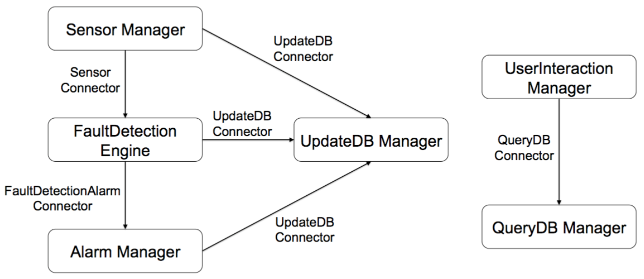


Fig. 11 Structure Diagram of The Power Plant Monitoring System

For the sake of brevity, 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 3. Answers to these questions provide us with the intent during the architecting process.

Table 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?
	…

D. *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 ADDs. 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 ADDs derived from the SceMethod cover the main architectural specifications and issues?
Table 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 [20]. 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 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 Fig. 11 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 [20]. 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 ADDs. 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 ADDs support architecture evolution?

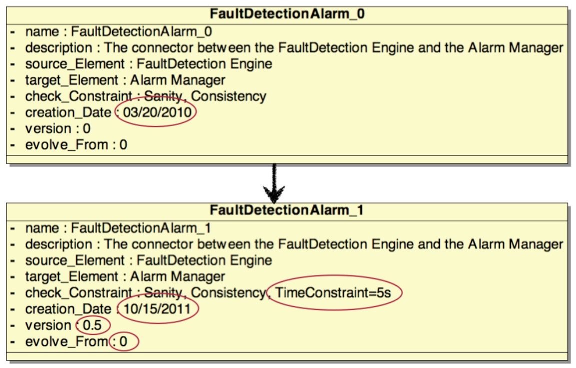
The architecture derivation process is basically an evolutionary process. Since architecture is regarded as a set of ADDs, we primarily analyze the evolution of ADDs to manage architecture evolution. The initial ADDs results largely cover the functional requirements of the power plant monitoring system, 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 *FaultDectectionAlarm Connector*.

Generally, the architecture evolution process is based on the initial ADDs 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.

E. Applying UML Metamodel

When applying our UML metamodel as a fine-grained way, we further defined all of these ADDs as objects for the corresponding UML classes. We finally derived seventeen objects for the elements, three objects for the architectural/design style and patterns, and about fifty objects for the rationale during the entire architecting process. The contents in Table 2 are transferred as the values of attributes in the element objects.

When we first applied the UML metamodel, we got the initial version of each object in every class. During the evolutionary change, the ADDs in the element, the constraint, and the intent view will be changed as well, and should be tracked and updated explicitly. With the UML metamodel, we do not need to update evolutionary changes from the Triple View Model, but only add new objects for specifying the corresponding changes. For example, "once a fault is detected by the *FaultDetection Engine*, the alarm should be raised within 5 seconds", which is a new limitation included in the requirement specifications. Based on this newly added requirement, we need to introduce a new object for the *FaultDetectionAlarm Connector* element that evolves from the previous one. The evolutionary change is shown in Fig. 12.



FaultDetectionAlarm_0
- name : FaultDetectionAlarm_0
- description : The connector between the FaultDetection Engine and the Alarm Manager
- source_Element : FaultDetection Engine
- target_Element : Alarm Manager
- check_Constraint : Sanity, Consistency
- creation_Date : 03/20/2010
- version : 0
- evolve_From : 0

FaultDetectionAlarm_1
- name : FaultDetectionAlarm_1
- description : The connector between the FaultDetection Engine and the Alarm Manager
- source_Element : FaultDetection Engine
- target_Element : Alarm Manager
- check_Constraint : Sanity, Consistency, TimeConstraint=5s
- creation_Date : 10/15/2011
- version : 0.5
- evolve_From : 0

Fig. 12 An Evolutionary Change

In Fig. 12, we can see that a new object *FaultDetectionAlarm 1* is added based on the changing requirement. The value of the attribute *check_Constraint* has been changed, and the *creation_Date* and the *version* respectively specified the new creation date and the version number of this object. We trace the evolution from the attribute *evolve_From,* and the evolution chain provides us the snapshot of the evolutionary history. Most of the time, the ADDs in the intent view evolve as well if the decisions in element view or the constraint view change. Hence we also need to document the reason and the justification as ADD objects, in order to specify the rationale for the time constraint of the *FaultDectectionAlarm Connector*. In this way, we explicitly record how the *FaultDetectionAlarm Connector* evolves in the architecting process and the rationale behind the new decision during the decision making process.

Due to the space limitations, we will not specify all the possible decisions in the architectural evolution. Using the UML metamodel we propose, we can effectively manage the evolutionary change of ADDs. The stakeholders in the software development process can share consistent architectural knowledge on ADDs evolution without knowledge evaporation

F. Discussion

1) Practicality

The TVM and the SceMethod, which are applied during the architecting and designing process, enable us to capture ADDs and manage their evolution. As the software life cycle proceeds, the ADDs results are widely employed throughout the entire software development process. Specifically, the documentation on ADDs 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 ADDs are employed in most of the software development phases, and finally architectural knowledge is well incorporated in various levels of the software development process.

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 ADDs. 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 management of architectural decisions for complex systems.

3) Limitations

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

VI. RELATED WORK

The key concepts of the traditional view on software architecture are components and connectors [1], [21]. Nowadays, software architecture has been seen as a set of ADDs [2], [22], [23]. The architectural decisions in the software architecting process are increasingly focused by researchers and practitioners [24]-[26], and ADDs are also considered to be a part of architectural knowledge [3]. In [27], 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 [28], [29], 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.

Tyree's template [4] 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 [3], 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. Archium [6] is a Java tool, including a complier and a run-time environment, for supporting ADDs 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 [30], PAKME [31], and RIAP [32], and other approaches in [33] and [34] are also proposed for managing architectural knowledge.

The research on managing the evolution of ADDs has been focused in the software architecture area as well. A number of models and tools have been proposed for ADDs evolution management. In [35], an approach for assisting architects in reasoning architectural evolution paths has been described, and the concept of evolution style is defined in it. Some other techniques as discussed in [36] and [37] introduce different approaches for capturing architectural evolution and selection architectural evolution alternatives.

A detailed comparison of these existing models and tools has been done in [7]. 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 [38]. Perry and Grisham have focused on architecture and design intent in [39], and our work in this paper 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 ADDs, 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 ADDs, which is effective in operating and maintaining the architecting process in a changing software development context. The UML metamodel incorporates evolution-centered characteristics to manage ADDs evolution, and helps to capture and trace the evolution of ADDs explicitly, thus reducing the evaporation of architectural knowledge that results from decisions evolution.

VII. Conclusions and Future Work

A recent strand of software architecture research is that software architecture is considered as a set of ADDs. ADDs 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 paper, we discuss the documentation and evolution of ADDs. We propose 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 ADDs. Based on the TVM, we propose a scenario-based method (SceMethod) for ADDs documentation and evolution. In addition, we develop a UML metamodel that incorporates key evolution-centered characteristics to manage the evolution of ADDs. The goal of the UML metamodel is to ensure that the architectural knowledge on the evolutionary changes of ADDs can be recorded and traced in a systematic way, in order to reduce architectural knowledge evaporation during the architecting process. The case study on an industrial-strength project validates the applicability and the effectiveness of the TVM, the SceMethod, and the UML metamodel.

In our future work, we plan to provide tool support for the TVM and the SceMethod. We also need to evaluate our UML metamodel in multiple large-scale software projects and complex systems.

References

[1] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes,* vol. 17, no. 4, 1992, pp. 40–52.
[2] J. Bosch, "Software architecture: The next step," *Software Architecture,* 2004, pp. 194–199.
[3] P. Kruchten, P. Lago, and H. van Vliet, "Building up and reasoning about architectural knowledge," *Quality of Software Architectures,* 2006, pp. 43–58.
[4] J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecture," *IEEE Software,* vol. 22, 2005, pp. 19–27.
[5] R. Capilla, F. Nava, S. Pe ́rez, and J. C. Due nas, "A web-based tool for managing architectural design decisions," *SIGSOFT Softw. Eng. Notes,* vol. 31, no. 5, 2006, p. 4.
[6] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer, "Tool support for architectural decisions," in *WICSA,* 2007, p. 4.
[7] M. Shahin, P. Liang, and M. R. Khayyambashi, "Architectural design decision: Existing models and tools," in *WICSA/ECSA,* 2009, pp. 293–296.
[8] D. E. Perry, "Issues in architecture evolution: Using design intent in maintenance and controlling dynamic evolution," in *ECSA,* 2008, pp. 1-1.
[9] M. Che and D. E. Perry, Scenario-based architectural design decisions documentation and evolution. In *ECBS,* 2011, pages 216–225.
[10] D. Tofan, "Tacit architectural knowledge," in *ECSA,* 2010, pp. 9–11.
[11] D. M. A. Reniers, "Message sequence chart: Syntax and semantics," Faculty of Mathematics and Computing, Tech. Rep., 1998.
[12] M. Che and D. E. Perry, "Evolution-Centered Architectural Design Decisions Management", *in the 11th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS '12)*, 2012, pp. 131-136.
[13] http://www.enel.com/en-GB/.
[14] D. Vanderveken, "Deriving architectural descriptions from goal-oriented requirements," Ph.D. dissertation, Universit catholique de Louvain, 2004.
[15] 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.,* vol. 8, no. 1, 1999, pp. 79–113.
[16] A. Coen-Porisini and D. Mandrioli, "Using trio for designing a corba-based application," *Concurrency - Practice and Experience,* vol. 12, no. 10, 2000, pp. 981–1015.
[17] M. Pradella, M. Rossi, D. Mandrioli, and A. Coen-Porisini, "A formal approach for designing corba based applications," in *ICSE,* 2000, pp. 188–197.
[18] M. Brandozzi, "Transforming goal oriented requirements specifications into architectural prescriptions," in *Proceedings STRAW 01, ICSE 2001,* 2001, pp. 54–61.
[19] M. Brandozzi and D. E. Perry, "Architectural prescriptions for dependable systems," in *ICSE 2002 Workshop on Architecting Dependable Systems,* 2002.
[20] D. Jani, D. Vanderveken, and D. E. Perry, "Deriving architecture specifications from kaos specifications: A research case study," in *EWSA,* 2005, pp. 185–202.
[21] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice.* Addison-Wesley Professional; 2nd edition, 2003.
[22] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *WICSA,* 2005, pp. 109–120.

[23] R. N. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons; New edition, 2009.

[24] J. C. Duen˜as and R. Capilla, "The decision view of software architecture," *Software Architecture,* 2005, pp. 222–230.

[25] P. Kruchten, R. Capilla, and J. C. Duenas, "The decision view's role in software architecture practice," *IEEE Software,* vol. 26, 2009, pp. 36–42.

[26] A. Ghazarian, "A domain-specific architectural foundation for engineering of numerical software systems," *in WSEAS transactions on systems*, vol. 10, no. 7, 2011, pp. 193-208.

[27] R. C. de Boer and R. Farenhorst, "In search of 'architectural knowledge'," in *SHARK,* 2008, pp. 71–78.

[28] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional, 2002.

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

[30] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *J. Syst. Softw.,* vol. 80, no. 6, 2007, pp. 918–934.

[31] M. A. Babar and I. Gorton, "A tool for managing software architecture knowledge," in *SHARK-ADI,* 2007, pp. 11.

[32] T. Al-Rousan, S. Sulaiman, and R. A. Salam, "Supporting architectural design decisions through risk identification architecture pattern (RIAP) Model," *in WSEAS transactions on information science and applications*, vol. 6, no. 4, 2009, pp. 611-620.

[33] A. A.A Saed, W. M.N. Kadir, and A. Yousif, "A Prediction Approach to Support Alternative Design Decision for Component-Based System Development," *in the 11th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS '12)*, 2012, pp. 85-91.

[34] Z. Li, X. Wang, and X. Yu, "Orthogonal software architecture design for radar data processing system with object-oriented component and COM interface," *in WSEAS transactions on computers*, vol. 10, no. 2, 2011, pp. 61-70.

[35] D. Garlan, J. M. Barnes, B. R. Schmerl, and O. Celiku, "Evolution styles: Foundations and tool support for software architecture evolution". In *WICSA/ECSA,* 2009, pp. 131–140.

[36] S.Ciraci, H.So¨zer, and M.Aksit, "Guiding architects in selecting architectural evolution alternatives". In *ECSA,* 2011, pp. 252–260.

[37] A. Zalewski, S. Kijas, and D. Sokolowska. "Capturing architecture evolution with maps of architectural decisions 2.0", in *ECSA,* 2011, pp. 83–96.

[38] R. Capilla, F. Nava, and A. Tang, "Attributes for characterizing the evolution of architectural design decisions," *in the 3rd International IEEE Workshop on Software Evolvability*, 2007, pp. 15-22.

[39] D. E. Perry and P. S. Grisham, "Architecture and design intent in component & cots based systems," in *ICCBSS,* 2006, pp. 155.