Exploring Architectural Design Decision Management Paradigms for Global Software Development

Meiru Che, Dewayne E. Perry
Department of Electrical & Computer Engineering
The University of Texas at Austin
Austin, Texas, USA
meiruche@utexas.edu, perry@mail.utexas.edu

***Abstract*—Global software development (GSD) is an increasing trend in the field of software engineering. It can be considered as coordinated activities of software development that are geographically and temporally distributed. The management of architectural knowledge, specifically, architectural design decisions (ADDs), becomes important in GSD due to the geographical, temporal, and cultural challenges in global environment. However, little work has be done on capturing, sharing, and evolving ADDs in a GSD context. Based on our previous work on ADD management in localized software development (LSD), we extend our study to explore ADD management paradigms for GSD in this paper. We propose three ADD management strategies for the distributed development environment, and according to global software project structures, we explore and analyze three typical ADD management paradigms that can be widely adopted in GSD. We aim to provide a fundamental framework on managing ADD documentation and evolution in GSD, and offer good insights into sharing and coordinating ADDs in a global setting.**

***Keywords*-architectural design decisions; global software development; architectural knowledge; documentation; evolution**

I. Introduction

Global software development (GSD) is an increasing focus in the field of software engineering. It can be considered as the coordinated activities of software development that are not localized and centralized but geographically and temporally distributed [14]. In GSD, software teams work together at geographically separated locations to accomplish software projects. Thus, global teams face challenges associated with the coordination of their work due to different locations, time zones, languages, and cultures. In order to cope with different challenges in the globalization of software development, communication, as well as coordination, a number of approaches have been proposed in different domains of GSD [1]. However, little attention has been paid to software architecting processes and software architectural knowledge management in the context of GSD. Similar to localized software projects, software architecting and architectural knowledge are important to support designing, developing, testing, and evolving software. We note, however, that in the global development of large complex systems, architecture plays an even more critical role in the structure of the project [13]. Therefore, managing and coordinating architectural knowledge such as architectural design decisions (ADDs) is a significant and also relatively new research problem in the context of GSD.

Perry and Wolf considered the selection of elements and their form to be ADDs, and the justification for these decisions to be found in the rationale [20]. It was not until 2004, with Boschs paper [5] at the European Workshop on Software Architecture, that software architecture has generally come to be considered as a set of ADDs. This specific focus on ADDs led to a broader focus on architectural knowledge [19]. Capturing and representing ADDs helps to organize architectural knowledge and reduce its evaporation, thus providing a better control on many fundamental architectural drift and erosion problems [20] in the software life cycle. In a globally distributed software environment, the documenting and sharing of ADDs can serve to support the complex collaboration and coordination needs of software projects. With the increasing trends of further globalization of software development, managing ADDs in GSD becomes a much more critical task than in a localized environment.

In our previous work on ADD management, we had an overall goal of providing a systematic approach that supports ADD documentation and evolution in a localized software development (LSD) context. Based on this, we intend to focus on involving ADD management in a global development environment in this paper. Since little work has been done on ADD documentation and evolution in GSD, we are going to discuss several ADD management strategies for multi-site software projects, and then explore the typical paradigms for ADD management in global software projects. We aim to provide a fundamental framework for managing ADDs in the context of GSD, and offer insights into architectural knowledge documentation and evolution for researchers and practitioners in the field of software architecture.

II. Localized ADD Management Approach

This section briefly introduces our previous work on ADD documentation and evolution in a localized software project context. We give an overview of the basic approach to

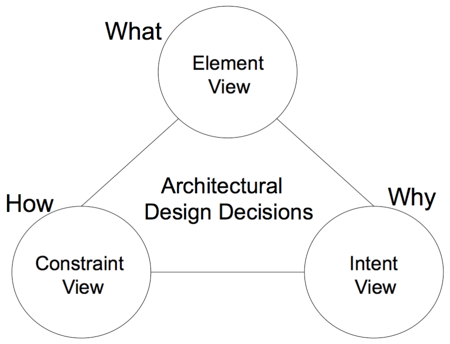

Figure 1. Triple View Model Framework

managing ADDs, which provides a foundation for exploring ADD paradigms in GSD.

A. *Triple View Model*

To capture and document the ADD set in a software project, we propose the Triple View Model (TVM) to clarify the notion of ADDs and to cover key features in an architecting process [7].

The TVM is defined by three views: the element view, the constraint view, and the intent view. This is analogous to Perry/Wolf models elements, form, and rationale but with expanded content and specific representations [20]. Each view in the TVM is a subset of ADDs, and the three views together constitute an entire ADD 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 an architecture, "how" these elements combine and interact with each other, and "why" a certain decision is made. The detailed contents of each view in the TVM are illustrated in Fig. 2.

In the element view, the ADDs describe "what" elements should be selected in an architecting process. We define computation elements, data elements, and connector elements in this view. Computation elements represent processes, services, 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 (at minimum) communication channels (that is, mechanisms to capture interactions) between those components in the architecture.

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

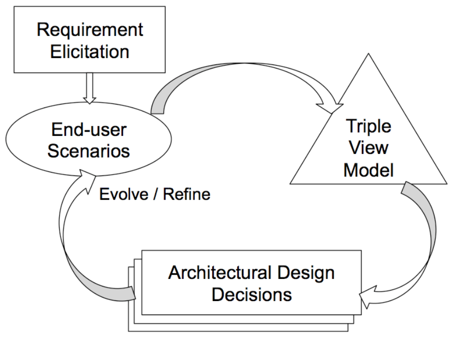

Figure 3. The Process of the Scenario-Based Method

The ADDs in the intent view are composed of rationale and best-practices in an architecting process. Rationale, which includes alternatives, motivations, trade-offs, justifications and reasons, is generated when analyzing and justifying every decision that is made. Best-practices are styles and patterns we choose for system architecture and design. The architectural decisions in the intent view mainly exist as tacit knowledge [24].

B. *Scenario-based ADD Documentation and Evolution*

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

In the SceMethod, we aim to obtain and specify the element view, constraint view, and intent view through end-user scenarios, which are represented by Message Sequence Charts (MSCs) [21]. Figure 3 illustrates the SceMethod process. At the beginning of the architectural design process, we obtain initial ADD results. Later on, as the requirements change, the ADDs are evolved and refined according to the new or updated requirements. By documenting all the possible ADDs and evolving these decisions with changing requirements, the SceMethod effectively makes ADDs explicit and reduces architectural knowledge evaporation.

Basically, we have the following four steps in the SceMethod to derive ADDs in a software project. For the sake of brevity, we will not discuss the detailed process of each step, but just give a brief introduction. We have the full description in [8].

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

2) From MSC Syntax to Element View: We derive the element view directly from the syntax of MSCs. Specifically, each agent instance is taken as a computation element, and from the interaction messages between the source and target agent instances, we can extract data elements accessed by computation elements. Connector elements serve as communication channels between computation elements. Therefore, the element view is derived as follows:

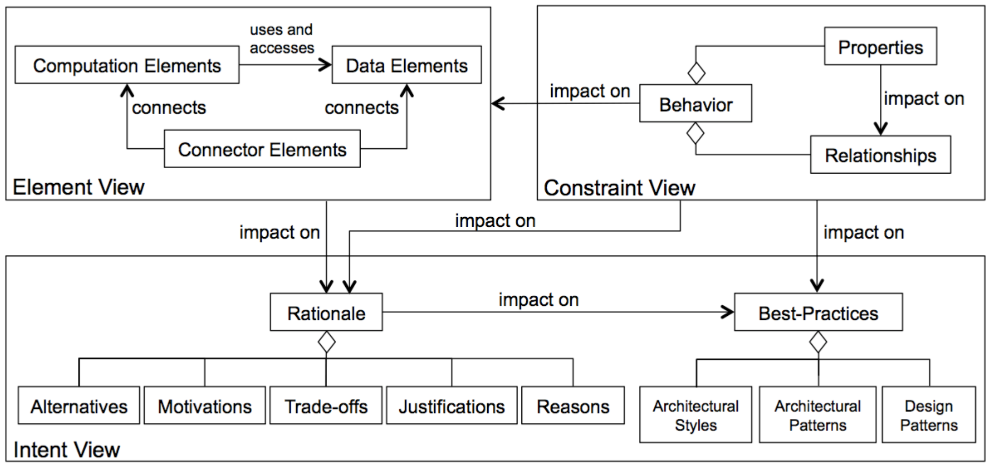

Figure 2. Triple View Model for Architectural Design Decisions

Computation Elements = {*Agent Instances*}
Data Elements = {*Interaction Messages*}
Connector Elements = {*Channels between Agents*}

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

Behavior = {*Prescriptions; Proscriptions*}
Prescriptions = {*Positive Scenarios*}
Proscriptions = {*Negative Scenarios; Exceptions*}

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

Properties = {*Receive; Issue; Check*}
Relationships = {*Event Traces by Path Expressions*}

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

Rationale = {*Answers to The Intent-Related Questions*}
Best-Practices = {*Architectural/Design Styles and Patterns*}

III. Multi-site ADD Management Strategy

As mentioned previously, the TVM and the SceMethod are the foundation of architectural knowledge management in GSD. However, managing ADDs in GSD becomes more difficult and complex than in LSD. On the one hand, the capturing and the documenting on ADDs are not just within a centralized environment, but considered for multi-site teams distributed geographically and temporally. On the other hand, the communication and the exchange of ADDs have a significant impact on the coordination of the distributed teams, and further, influence the subsequent analysis, design and implementation of global projects.

In order to support ADD management in GSD projects, we propose three different strategies for managing the documentation and the evolution of ADDs in a distributed context, and discuss how distributed sites coordinate with each other to share and maintain consistent architectural knowledge. The three strategies for multi-site ADD management are client-server strategy, hybrid strategy, and incremental strategy respectively. Table I describes the detailed ADD management mechanism for each strategy.

In the *client-server strategy*, one site in the global software teams is considered as the headquarters, and it is responsible for the entire process of ADD documentation and evolution

Table I
Multi-site ADD Management Strategies

Strategy	**ADD Management Mechanism**
Client-Server Strategy	Centralized ADD documentation on the headquarters site; Centralized ADD evolution on the headquarters site; Central repository is set up to store ADD knowledge information from the headquarters site; Central repository is accessed by all the sites.
Hybrid Strategy	Individual ADD documentation on each local site; Individual ADD evolution on each local site; Central repository is set up to store ADD knowledge information from each local site; Central repository is accessed by all the sites.
Incremental Strategy	Individual ADD documentation on site 1; Individual ADD evolution on site 1; ADD knowledge on site 1 is transferred to site 2; Individual ADD documentation on site 2; Individual ADD evolution on site 2; ADD knowledge on site 2 is transferred to site 3; . . ADD knowledge on site n-1 is transferred to site n; Individual ADD documentation on site n; Individual ADD evolution on site n.

in the global software project. Therefore, all the tasks on ADD documentation and evolution are conducted in the headquarters, which is similar to a localized software project context. In addition, a central repository is set up in the headquarters site to record and store the up-to-date ADDs, so that all the other sites can access the repository to share and reuse the ADDs through the global context. We term this strategy the client-server strategy because architectural knowledge resides in a central repository (as the server) and is accessed by all the distributed sites (as the clients).

In the *hybrid strategy*, every individual site manages architectural knowledge in a localized context, i.e., each site in the GSD project documents and evolves its own ADDs that are derived from its local architecting process. However, a central repository is also set up in one of the GSD sites for storing and sharing architectural knowledge throughout all the global teams. The repository is accessed by all the sites in the global project, and by this means, different sites can share and reuse ADD knowledge, or even reapply ADD knowledge in a different context. We term this strategy the hybrid strategy because it combines both the local ADD management and the global architectural knowledge sharing and reusing.

In the *incremental strategy*, the ADD management mechanism is analogous to the incremental development, i.e., on site 1, it manages the local ADD documentation and evolution process, and stores all the architectural knowledge in its local site. When site 1 derives all the ADD knowledge, it transfers the ADDs to site 2. Site 2 manages its local ADD documentation and evolution as well, and moreover, it combines the ADD information from site 1 into a larger ADD set. Similarly, when site 2 derives all the ADD knowledge, it transfers the ADDs to site 3, which follows the same way as site 1 and 2. In this strategy, each site captures ADDs in a certain context of the global project, and the final goal is that we are able to have a fully complete ADD set through the incremental documenting and evolving process.

IV. ADD Management Paradigms in GSD

In this section, we aim to explore the typical ADD management paradigms in GSD. First, we introduce three main software project structures adopted in a global development context, then we discuss the corresponding paradigm that is specific to each project structure.

A. Global Software Project Structures

Since global software projects very often have to deal with large and complex software systems, and development activities are performed by geographically different teams, the structure of a global software project plays a significant role in GSD. A good structure provides an effective way to organize the GSD project across multiple development sites, and in the meantime, it also offers a platform for managing the resources on both the development and the organizational activities.

A large number of possible ways to structure the GSD projects have been adopted. The main structures widely used are product-based structure, process-based structure, and release-based structure [3]. In addition, platform-based structure, competence-based structure, and open source structure are also often considered in GSD [3]. In this paper, the focus is on the first three typical structures to address ADD management in GSD.

In a *product-based structure*, a global system is decomposed into different components based on its requirement specification. Different components are then allocated as work items to different global teams. In a *process-based structure*, work items are allocated across different teams in accordance with the development phases of a software project. Specifically, we may allocate requirement, design, development, and test to different sites, and each site focuses on the tasks in the specific phase. As for the *release-based structure*, each site is responsible for a different release of the project, i.e., the first product release is developed on site 1, the second release is developed on site 2, and the third on site 3. In most cases, the releases are overlapped on different sites due to the timing requirement from customers.

B. ADD Management Paradigms

Given the foregoing discussion, we are going to explore and discuss three different paradigms for managing ADDs in GSD, which are specific to the three widely used software project structures.

1) Product-based Paradigm (Product-based Structure / Hybrid Strategy): For product-based structures in GSD, the system is decomposed into components and the components are allocated to distributed sites, thus each site conducts its own architecting process locally focusing on the functionality of the allocated components. During the architecting process in each local site, ADDs can be captured and documented by using the TVM and the SceMethod. We adopt the hybrid strategy to manage ADDs in the GSD projects with product-based structures. Figure 4 illustrates this paradigm in details.

As shown in Fig. 4, each site manages ADD documentation and ADD evolution locally according to the SceMethod and the TVM that we discussed in localized software projects. In addition, one of the global sites is selected as the headquarters and needs to set up a central repository for recording and storing the architectural decisions, which enables the geographically distributed sites to share ADDs in the global context. Each site can access the central repository, check in their local ADDs to the repository, and even read and reuse the ADDs from other sites when necessary. The headquarters site with the central repository coordinates the architectural knowledge in the repository and keep them consistent without conflicts. During the evolutionary process,

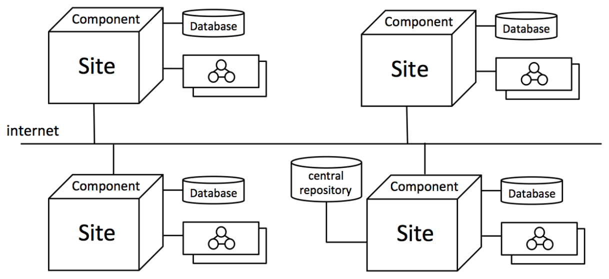

Figure 4. Product-based Paradigm in GSD

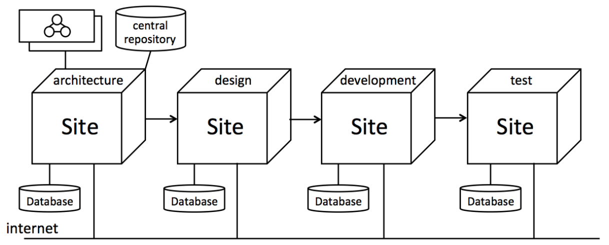

Figure 5. Process-based Paradigm in GSD

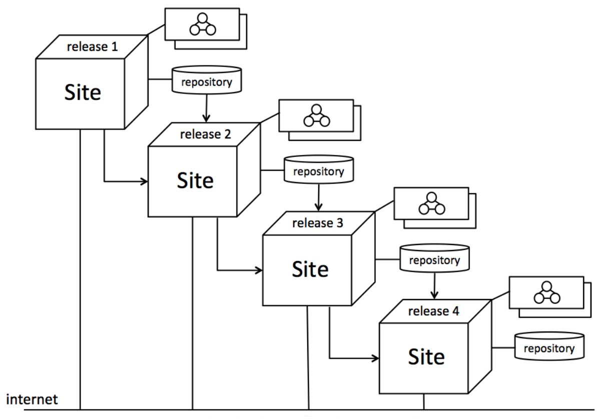

Figure 6. Release-based Paradigm in GSD

the evolved ADDs from each site are also transferred to the central repository. As we discussed for the hybrid strategy, the multiple sites in GSD manage architectural decisions not only within their local sites, but also with a central coordination to share and reuse architectural knowledge.

2) Process-based Paradigm (Process-based Structure / Client-Server Strategy): For the process-based structures in GSD, it is appropriate to use client-server strategy for managing ADDs. The reason is that the architecting process mainly occurs in the architecture phase, and all the other subsequent development phases, i.e., the design, development, and testing phases, are largely considered as the clients who access the ADDs that are derived in the architecture phase. Therefore, the client-server strategy provides us suitable support for GSD projects with process-based structures. We describe this paradigm in Fig. 5.

In Fig. 5, we note that the architecting process is conducted in the site with architecture phase, relying on our TVM and SceMethod to derive the entire ADD set. Moreover, a repository is set up in the same site to manage architectural knowledge documentation and evolution. This repository is also regarded as a central repository among the global teams, and all the other sites access the repository for sharing and reusing ADDs in the specific development phases. In some cases, the subsequent development phases, such as design phase, may also come up with new ADDs as the process proceeds. However, we do not deal with this kind of exceptions for now, but only explore the general paradigms that are normally used in GSD. More implications and exceptions will be addressed in our future work.

3) Release-based Paradigm (Release-based Structure / Incremental Strategy): The third paradigm that we are going to explore and analyze is for GSD projects with the release-based structures. In release-based structures, different product releases are allocated to different sites, so that each site handles all the development activities for the assigned release. It is obvious that in this paradigm each site derives its ADD set locally, and maintain ADD documentation and evolution in its local repository. Note that we do not need to create a central repository as the previous two paradigms, but only use each local repository for the architectural knowledge management.

As illustrated in Fig. 6, each repository plays an important role in establishing a bridge to transfer architectural knowledge, which complies with the mechanism in our incremental strategy. Typically, the ADDs from the first site are transferred to the second site, so that the architectural knowledge from the first product release can be reused efficiently in the next release. In the release-based structure, the multiple releases contain similar or even the same functionalities and product features, which implies that the ADDs derived from different releases may have similarities as well. By adopting the incremental strategy in this paradigm, each repository can serve as a reused ADD pool, and the latter site is easy to combine, reuse, or even modify the ADDs derived from the former site.

V. Related Work

The key concepts of the traditional view on software architecture are components and connectors [4], [20]. Nowadays, software architecture has been seen as a set of ADDs [16], [23]. The architectural decisions in the software architecting process are increasingly focused by researchers and practitioners [12], [18], and ADDs are also considered to be a part of architectural knowledge [19].

Guidelines for documenting software architecture has been provided in [9], [15], however, those documentation approaches do not explicitly capture ADDs in the architecting process. Recently, many models and tools have been proposed for capturing, managing, and sharing ADDs, most of which are discussed and used within a localized software development context. Tyrees template [25] 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 [19], an ontology of ADDs and their relationships have been described. This ontology then can be used to construct architectural knowledge of a software system. ADDSS [6] is a web-based tool for documenting ADDs. It establishes the backward and forward traceability between requirements, decisions, and architectures. Other models and tools such as Archium [17] and AREL [22] are also proposed for managing ADDs

With the increasing attention paid to GSD, ADD management should be able to effectively applied in a GSD setting as well. However, little work has be done on ADD management in the GSD environment. A few of general architectural knowledge management practices for GSD have been proposed in [10], and the usefulness of these practices are evaluated in [11]. Furthermore, a literature review has been done [2] to explore architectural knowledge in a GSD context, and to synthesize architectural knowledge concepts, practices, tools and challenges that are important in GSD. In [26], six architectural viewpoints are defined to model GSD systems, which are based on a metamodel that has been derived after a thorough domain analysis of GSD literature.

Notably, architectural knowledge, specifically, ADDs, has not been widely discussed and supported in GSD, and the aforementioned approaches do not address in detail how to capture, share, and evolve ADDs in a global software project. In this paper, our goal is to provide a fundamental framework on managing ADDs in the GSD context.

VI. Conclusions and Future Work

With the increasing trend of GSD, the management of ADDs becomes more significant and critical due to the geographical, temporal, and cultural challenges innate to GSD. In this paper, we propose three different strategies for managing ADDs within multiple distributed development sites. Based on this, we explore three typical ADD management paradigms that can be widely used in GSD, and provide a high-level methodology on how to manage the documentation and the evolution of ADDs in the GSD context. In our future work, we plan to perform field studies to evaluate the ADD management paradigms in GSD projects. We also intend to investigate problems and implications for ADD management to provide insights into architectural knowledge management in GSD.

References

[1] First Workshop on Architecture in Global Software Engineering. Helsinki, Finland, August 2011. Http://www.cs.bilkent.edu.tr/AGSE-2011/.

[2] N. Ali, S. Beecham, and I. Mistrik. Architectural knowledge management in global software development: A review. In *ICGSE'10*, pages 347–352, 2010.

[3] A. Avritzer, D. Paulish, and Y. Cai. Coordination implications of software architecture in a global software development project. In *WICSA '08*, pages 107–116, 2008.

[4] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, Boston, MA, USA, 1998.

[5] J. Bosch. Software architecture: The next step. In *EWSA'04*, pages 194–199, 2004.

[6] 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.

[7] M. Che and D. E. Perry. Scenario-based architectural design decisions documentation and evolution. In *ECBS'11*, pages 216–225, 2011.

[8] M. Che and D. E. Perry. Managing architectural design decisions documentation and evolution. *International Journal Of Computers*, 6:137–148, 2012.

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

[10] V. Clerc. Towards architectural knowledge management practices for global software development. In *SHARK'08*, pages 23–28, 2008.

[11] V. Clerc, P. Lago, and H. v. Vliet. The usefulness of architectural knowledge management practices in gsd. In *ICGSE'09*, pages 73–82, 2009.

[12] J. C. Dueñas and R. Capilla. The decision view of software architecture. In *EWSA'05*, pages 222–230, 2005.

[13] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The geography of coordination: dealing with distance in r&d work. In *GROUP '99*, pages 306–315, 1999.

[14] J. D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *FOSE*, pages 188–198, 2007.

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

[16] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, pages 109–120, 2005.

[17] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *WICSA*, page 4, 2007.

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

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

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

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

[22] 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.

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

[24] D. Tofan. Tacit architectural knowledge. In *ECSA'10*, pages 9–11, 2010.

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

[26] B. M. Yildiz and B. Tekinerdogan. Architectural viewpoints for global software development. In *ICGSE*, pages 9–16, 2011.