Linking Quality Attributes and Constraints with Architectural Decisions

David Ameller and Xavier Franch
Universitat Politècnica de Catalunya
Barcelona, Spain
{dameller, franch}@essi.upc.edu

June 22, 2012

Abstract

Quality attributes and constraints are among the principal drivers in architectural decision making processes. Quality attributes are improved or damaged by architectural decisions, while constraints directly include or exclude parts of the architecture (e.g., logical components or technologies). We may determine the impact of an architectural decision in the software quality, or which parts of the architecture are affected by a constraint, but the hard problem is to know if we are respecting the quality requirements (requirements over the quality attributes) and the imposed constraints with all the architectural decisions made. Currently, the most usual approach is that architects use their own experience to produce software architectures that comply with the expected quality requirements and imposed constraints, but at the end, especially for crucial decisions, the architect has to deal with complex tradeoffs between quality attributes and juggle with possible incompatibilities raised by the imposed constraints. To facilitate this task and make architect's decision making processes more reliable and effective, in this paper we present the Quark method to guide the architects in the software architecture design. Quark relies on a specialized ontology, Arteon, which is in charge of managing the architectural knowledge. The decisional part of Arteon is also presented in this paper.

1 Introduction

In the last decade, software architecture has become one of the most active research areas in software engineering. As a significant trend in this community, many researchers have stated that architectural decisions are the core of software architecture [25, 19]. Under this view, software architecture has evolved from a simple structural representation to a decision-centric viewpoint [19]. Along this way, several methodological approaches for architecture design and analysis had been proposed, e.g., ADD, TOGAF, ATAM, CBAM. These are heavyweight methods that offer a significant gain in the reliability of the architecture design process but require large-scale projects to achieve a balance between what the method offers and the effort that supposes for the architects to use it. This balance is hard to achieve when projects are low- or medium-scale. Lighter methods that apply for not so large projects are required [23].

In this context, we present Quark (Quality in Architectural Knowledge), a decision-centric lightweight method for architecture design. Quark builds upon the management and reuse of Architectural Knowledge (AK) [20] represented as an ontology called Arteon. Arteon includes knowledge about architectural decisions, including their rationale and their link to quality attributes and constraints. In this paper we present Quark and the decisional part of Arteon.

The rest of this paper is divided in the following sections: the Quark method (Section 2), the representation of AK using Arteon (Section 3), an example of use of Quark and Arteon (Section 4), analysis of the presented work (Section 5), and finally conclusions and future work (Sections 6 and 7).

2 The Quark Method

Quark aims at providing means to facilitate and making more reliable architects' decisions with regard to quality attributes. The method starts from the software requirements, and ends with a set of architectural decisions and the overall evaluation of the quality attributes.

The design of Quark has been driven from a critical observation gathered from two empirical studies we have recently conducted[1]: software architects may be receptive to new design methods as far as they still keep the control on the final decisions. In other words, architectural decision making should not be an automatic process, but just computer-assisted, therefore still architect-driven.

As a consequence, the architect plays the central role in Quark. There are two tasks where the architect role takes special relevance. First, the architect iteratively defines the quality requirements and constraints relevant to the software architecture. Second, the architect chooses the most convenient architectural decisions and decides when the decisions made are sufficient to end the process.

In the same direction, Quark does not pretend to be intrusive. It notifies about possible incompatibilities and possible actions to solve them, but the method does not require resolving an incompatibility to continue with the design, it is up to the architect to decide.

The Quark method delivers an iterative process divided in four differentiated activities (see Figure 1): first, specification of the quality requirements and the imposed constraints; second, inference of architectural decisions according to existing AK; third, decision making; and fourth, architectural refinement (if necessary). Whenever the solution is refined, activities 1-3 should be repeated. In the following subsections we give details on each activity.

[1]Not available as publication yet.

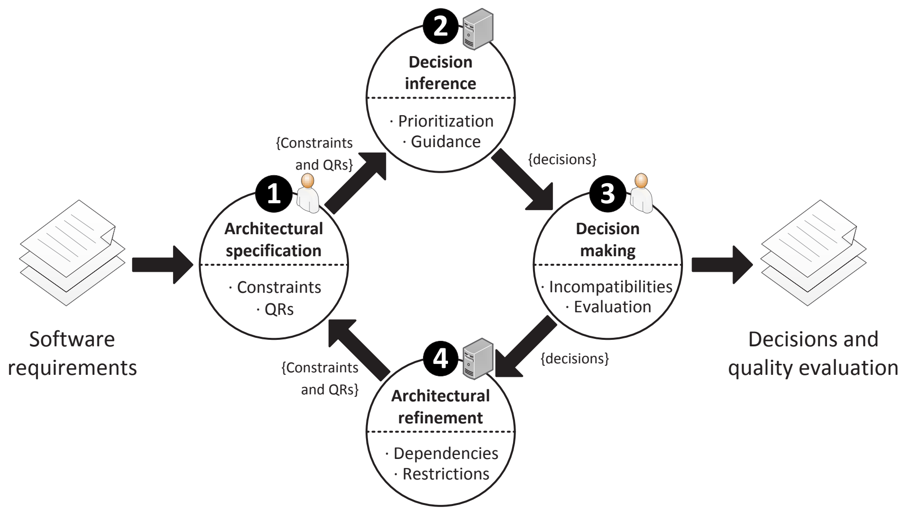

Figure 1: Quark overview.

2.1 Architectural Specification

In the first activity, the architect specifies the quality requirements and constraints that will drive the architecture design. They represent the architect interpretation of the software requirements that are relevant for the architecture design. For example, a quality requirement could be "performance should be high" (in other words, more a goal than a requirement) or something more concrete as "loan processing response time should not be higher than two seconds 95% of the times". Constraints are typically referring to technologies, e.g., "the database management system (DBMS) must be MySQL 5", but may also refer to architectural principles, patterns or styles, as in "the architectural style must be Service-Oriented Architecture (SOA)". More details on how this specification can be done are given in Section 3.

Due to Quark's incremental nature, the specification of these quality requirements and constraints does not need to be complete. The architect has freedom to decide if s/he wants to start from a very short specification and then make the architecture grow in each refinement or if s/he wants to provide a more complete specification and see if the expected quality calculated by the method matches the expected quality requirements, and then refine till the architecture complies with the requirements.

2.2 Decision Inference

In the second activity, the Quark method uses the AK available to generate a list of decisions. Since the expected amount of decisions is large, the decisions should be ordered by priority using some criteria (e.g., the decisions that satisfy more constraints and better comply with the stated quality requirements are top priority).

Decisions need to be informative. This means that, beyond their name, decisions must include information about: why the decision was offered?, what is the impact in the overall architecture quality?, and what other implications involve making the decision? More complete descriptions of decisions are available in the literature (e.g., [25]).

For example, for the decision “data replication” we could answer the above questions as follows: “the decision to have data replication is offered because there is a requirement about having high performance”, “by making this decision, the overall performance will increase but will affect negatively to the maintenance, and can damage the accuracy”, “also, by selecting this decision, the used DBMS is required to be able to operate with data replication.”

2.3 Decision Making

In the third activity, the architect decides which decisions wants to apply from the ones obtained in the previous activity. When the architect makes a decision, two things may happen: First, there could be incompatibilities with previous decisions (e.g., we are selecting the “data replication” decision, but we already selected a DBMS that does not support data replication), and second, there could be one or more quality requirements that do not hold by the decisions made (e.g., the decisions made indicate that maintainability will be damaged while there is a quality requirement that says that maintainability is very important for this project).

In both cases, the architect will be informed about which decisions are the most conflictive, but at the end s/he will decide if the actual set of decisions is satisfactory or not (e.g., there may be external reasons, beyond requirement satisfaction, that have higher priority).

When the architect ends the decision making, s/he has the opportunity to conclude the process by accepting the current set of decisions and their impact in the quality attributes. As mentioned in [25], we understand the software architecture as a set of architectural decisions. Alternatively, the architect may choose to make a new iteration. A new iteration starts in the refinement activity.

2.4 Architectural Refinement

The objective of the refinement activity is to make actions that will resolve the detected issues in the next iteration. We identified three possible outcomes from the decision making activity: incompatibilities, dependencies, and suggestions for quality requirements. Incompatibilities may be translated into conditions over the attributes of the architectural elements (see Section 3.4). For example, there must be an attribute that indicates if the DBMS supports data replication. Dependencies occur when some decision requires other parts in the architecture. For example, when the architect decides to use SOA, several related decisions are needed: service implementation (e.g., SOAP, REST), service granularity

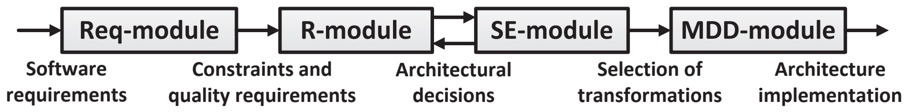
Req-module
R-module
SE-module
MDD-module
Software requirements
Constraints and quality requirements
Architectural decisions
Selection of transformations
Architecture implementation

Figure 2: Arteon modules.

(e.g., service composition, single service), etc. And last, we may infer that some quality attribute is of special relevance due to the selected decisions, e.g., most of the decisions have positive impact on security. In that case we may suggest to the architect to consider including a quality requirement about security. Doing this we help architects making quality requirements explicit.

Incompatibilities and dependencies can be translated into constraints, while the third outcome will generate new quality requirements. In the specification activity the architect will decide which of these outcomes will be included in the next iteration. At this point, the architect can also add or modify the constraints and quality requirements. For example, the architect may have noticed in the last iteration that one quality requirement in particular is very difficult to meet and decide to soften it.

3 Managing and Reusing Architectural Knowledge

Central to Quark is the management and reuse of the AK that supports the decision making process. Among other alternatives, we have chosen to use ontologies to represent the AK. Ontologies have been successfully used for knowledge representation in other domains (e.g., software engineering, artificial intelligence, semantic web, biomedicine, etc.) and they offer other advantages such as reasoning and learning techniques ready to be applied (e.g., we could add new decisions using case-based reasoning techniques).

We have designed Arteon (Architectural and Technological Ontology) to manage and reuse AK. A first overview of Arteon appears in [3]. In that paper, the focus was on the part of the ontology related to the structural elements (i.e., components and their relationships). In this paper, we focus on a different part of the ontology, the part related to architectural decisions. Arteon follows to the ontology design principles stated by Gruber [11], Guarino [12] and Evernmann [8]: clarity, coherence, extendibility, minimal encoding bias, minimal ontological commitment, identity criterion, basic taxonomy, and cognitive quality.

Most of the concepts that appear in this ontology are adopted from the software architecture community. They are defined carefully, and whenever possible we simply adhere to the most widely-accepted definition (according to the minimal ontological commitment design principle).

One particularity of this ontology is that it has been diagramed using UML class diagrams. The principal reason is that UML class diagrams are optimal as communication means. On the other hand, UML has not been a limitation to express any concept or relationship, thus the minimal encoding bias principle has not been jeopardized. We also found in the literature many approaches that use UML, or UML variations, to diagram the ontologies (e.g., [13, 5]) and there is also the possibility to convert the UML representation of the ontology into OWL [9].

The typical benefits of materializing AK are sharing and reusing this knowledge in different software projects and/or communities of architects. But in this paper we go one step forward. We propose to use AK to guide and facilitate the architects' decision making process and, eventually, bring more reliability to this process by surfacing new alternatives that were not initially considered by the architect. To apply the learning and reasoning techniques necessary to walk this step, we need to be able to formalize this knowledge. In this section we provide some examples of formalizations of this knowledge to show its feasibility.

Arteon is divided into four modules: Req-module, representing software requirements knowledge; R-module, reasoning and decision making knowledge (the module presented in this paper); SE-module, structural elements, views and frameworks knowledge (presented in [3]); and MDD-module, Model-Driven Development (MDD) related knowledge. Although interconnected (see Figure 2), the four modules are loosely coupled and highly cohesive enough to be reused separately. In the particular, the Req-module is related to the R-module through a relationship between software requirements (functional and non-functional) and constraints. The relationship between the R-module and the SE-module is done through a specialization of the Decisional Element concept (see Section 3.1) into a full classification of these elements. The relationship between the SE-module and the MDD-module consists on the selection of the correct MDD transformations to apply. The last link is necessary for our vision in software architecture design [4], which is out of the scope of this paper.

In Figure 3 we show the principal concepts of the R-module and their relationships. The rest of the section provide details of R-module.

3.1 Decisional Element

A Decisional Element is an elemental part of an architecture the architect can decide upon, i.e., the object of decisions. This concept is specialized in the SE-module, so it is left unrefined in the R-Module. The different types of Decisional Element proposed in the SE-module are: architectural styles (e.g., 3-layers), style variations (e.g., 3-layers with data replication), components (e.g., persistence layer), and implementations (e.g., Hibernate). Of course, this is not the unique possible specialization of this concept. Being a modular ontology makes it is easy to design and use a different specialization hierarchy for the Decisional Element, which is aligned with the extendibility principle.

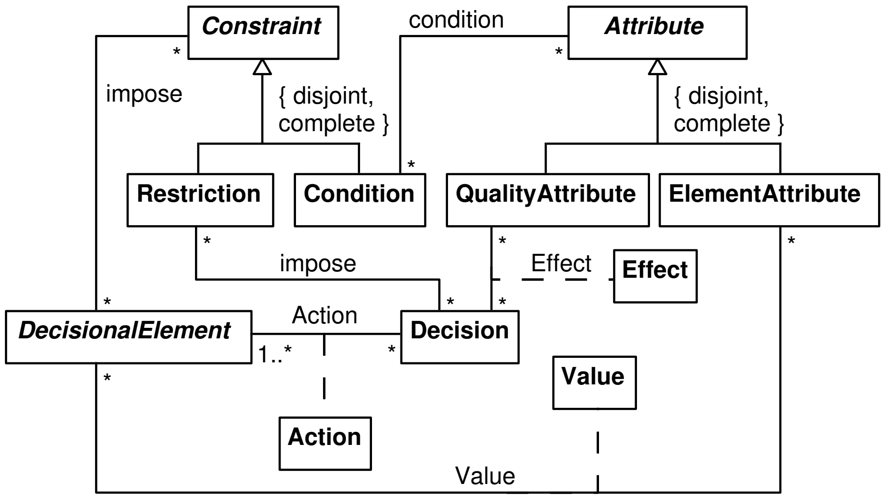

Figure 3: Arteon R-module.

3.2 Decision

According to RUP [18], software architecture is the "selection of the structural elements and their interfaces by which a system is composed, behavior as specified in collaborations among those elements, composition of these structural and behavioral elements into larger subsystem, architectural style that guides this organization". This definition is about making architectural decisions, structural and behavioral, both classified as existence decisions by Krutchen et al. [20].

In Arteon, the decision concept is very similar to the existence decision concept. Decisions are actions over Decisional Elements where the action determines the effect of the decision. Due to the extendibility design principle, we have not closed the ontology to a predefined set of actions, but possible actions could be, for example, the ones proposed in [20]: *use*, the Decisional Element will be in the architecture, and *ban*, the Decisional Element will not be in the architecture.

3.3 Constraint

Constraints can be imposed by software requirements or by decisional elements (the concept of requirement belongs to the Req-module of Arteon). Constraints coming from requirements are normally described in natural language (e.g., "the system shall be developed in C++"), sometimes using templates (e.g., Volere [22]) or a specialized language (e.g., temporal logic, the NFR Framework [6], etc.). Constraints coming from Decisional Elements are formalized as part of the AK (e.g., when the architect uses a technology that is only available for a particular platform, s/he is restricting the architecture to this platform).

```
RestrictionSet → Restriction (LogicOp Restriction)*
Restriction → Action [DecisionalElement]
Action → <use> | <ban>
ConditionSet → Condition (LogicOp Condition)*
Condition → ComparativeCond | ConjuntiveCond
ComparativeCond → [Attribute] CompOp [Value]
ConjunctiveCond → [Attribute] ConjOp [Value]+
LogicOp → <and> | <or>
CompOp → <greater_than> | <lower_than> | <equal_to>
ConjOp → <includes> | <excludes>
```

Figure 4: CFG to formalize constraints.

Independently from the origin of the constraint, we distinguish between two kinds:

Restriction A constraint that directly imposes one or more decisions. For example, "SQL Server" DBMS needs "Windows" operating system.

Condition A constraint that specifies the expected values for some attributes. An example of condition could be that we need to limit the valid licenses in an Open-Source Software (OSS) project: the element attribute "license" must be equal to any of these values "GPL", "LGPL", "BSD", etc..

From the point of view of a reasoning system, constraints are used to reduce the amount of valid solutions, or as it happens in Quark, to give lower priority to the decisions that do not comply with the constraints.

In order to be able to reason with these constraints they must be formalized as evaluable expressions. Again, the ontology does not commit to any particular proposal, but we provide an example expressed as a Context Free Grammar (CFG) [21] (see Figure 4). For simplification, we included extra notation in the CFG: *[concept]* mean one valid instance of the concept and *¡symbol¿* mean a terminal symbol. Also, for simplification, we did not include semantic rules (e.g., "the data type of the value should be the same of the data type of the attribute"). Depending on the expressiveness of the formalization, constraints could contain logic, comparative and conjunctive expressions, but expressiveness impacts negatively on the complexity of the reasoning system.

3.4 Attribute

An Attribute is an "inherent property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means" [17]. In Arteon we differentiate two kinds of attributes:

Element Attribute An attribute of a type of decisional element. For example, AK about technologies will include the name of their license as value for the element attribute "license" whenever this information is considered relevant for making decisions.

Quality Attribute An attribute that characterizes some aspect of the software quality. For example, ISO/IEC 9126-1 [16] defines a hierarchy of quality attributes (named "characteristics" in the standard: functionality, reliability, usability, efficiency, maintainability, and portability).

In this case, we also followed the extendibility principle by leaving the attributes customizable. Initially, we thought to propose a set of attributes, the most generic and independent of domain, but when we tried we found out that domain specific quality models may be more adequate in each situation (e.g., the S-Cube quality model [10] is specific for SOA) and that the element attributes are uncountable, and even worse, the same information can be modeled with different attributes (e.g., for the license information, we may have a boolean attribute, true when is a OSS license and false otherwise, or as before have an attribute with a list of licenses). We finally opted to let the domain expert decide which attributes are more convenient in each case, but we acknowledge that more research is needed in order to make this knowledge reusable between different projects.

4 Example

Since the complete architectural decision making process of a software architecture is too big to be included in a paper, the present example will focus only in one aspect of the architecture, the DBMS. The example is mostly about technologies, but the same idea can be also applied, for example, to the selection of architectural patterns.

Following the Quark method, first the architect should revise the software requirements and identify the ones that are relevant to the architecture. For this example, the requirements are: (R1) the software system shall keep the information about clients and providers, (R2) the software system shall be developed using OSS whenever possible, and (R3) the software system shall have backup systems for reliability.

4.1 Specification Activity

Once software requirements are selected, the software architect should translate them into quality requirements and constraints. From the R1 the architect may deduce that the project is an information system, so a DBMS will be required. R2 sets a constraint on the technologies used to be OSS. R3 sets constraints for backup facilities, and also mentions that reliability is a desired quality attribute. Using the formalization presented in Section 3 the specification will be: *Use* DBMS, "License" *includes* "GPL", "LGPL", "BSD", etc., "Backup facility" *equal* "yes", and "Reliability" *greater than* "average".

4.2 Decision Inference Activity

Next, depending on the AK we have in the knowledge base and using prioritization criteria, an ordered list of decisions will be generated. In this example, the prioritization criteria is to give high priority to decisions that include technologies to implement the DBMS architectural component and the decisions that satisfy more constraints. We propose the following decisions, using as the AK base the information published in the Postgres Online Journal [15]. The decisions are described as mentioned in Section 2.2:

1. The decision to use MySQL 5 is offered because it is OSS. There is no information available about backup facilities in MySQL. MySQL is preferred because it supports more OSS technologies. Using MySQL has neutral impact in reliability because ACID compliance depends on the configuration.

2. The decision to use PostgreSQL 8.3 is offered because it is OSS. There is no information available about back-up facilities in PostgreSQL. There are few OSS technologies with support for PostgreSQL. Using Postgre-SQL improves reliability because it is ACID compliant.

3. The decision to use SQL Server 2005 is offered because it satisfies the backup facility condition. SQL Server is not OSS. There are few OSS technologies with support for SQL Server. SQL Server will require a Windows operating system. Using SQL server improves reliability because it is ACID compliant.

4.3 Decision Making Activity

In the decision making activity, the architect, for example, will decide to use MySQL 5 (the decision with higher priority) as the implementing technology for the DBMS component. But as said before in this paper, the architect may prefer to use PostgreSQL, even it is not the top decision, because s/he is more familiar to it (or any other reason). The important point is that the architect is able to make informed decisions, and, eventually, new decisions that were unknown to her/him are taken into consideration.

4.4 Architectural Refinement Activity

After the decision making activity the architectural design will continue with new iterations, where the decisions to use MySQL will impact, for example, in the selection of other technologies that are compatible with MySQL. This information will appear during the refinement activity as dependencies and incompatibilities.

5 Analysis

Hofmeister et al. [14] compared five software architecture design methods and came out with a general model of architectural design method composed of three activities:

Architectural analysis "serves to define the problems the architecture must solve."

Architectural synthesis "is the core of architecture design. This activity proposes architecture solutions to a set of architectural significant requirements, thus it moves from the problem to the solution space."

Architectural evaluation "ensures that the architectural design decisions made are the right ones."

In Quark architectural analysis is covered with the specification activity, architectural synthesis is covered with the decision inference activity, and architectural evaluation is covered with the decision making activity, but there are two differences between Quark and the general approach of architectural design proposed by Hofmeister. First, the general approach does not consider iterative methods, which is why we have an extra activity in our method. The second difference is that Hofmeister's general approach deals with complete architectural solutions, while Quark works at decisional level. In our exploratory studies we have detected that architects will not trust a support system that generates full architectural solutions without their intervention.

We have compared Arteon with Kruchten's taxonomy of architectural decisions [20], which has three kinds of decisions:

Existence decisions "states that some element / artifact will positively show up, i.e., will exist in the systems' design or implementation."

Property decisions "states an enduring, overarching trait or quality of the system. Property decisions can be design rules or guidelines (when expressed positively) or design constraints (when expressed negatively), as some trait that the system will not exhibit."

Executive decisions "do not relate directly to the design elements or their qualities, but are driven more by the business environment (financial), and affect the development process (methodological), the people (education and training), the organization, and to a large extend the choices of technologies and tools."

Existence decisions, as mentioned in Section 3.2, are represented as the Decision concept of Arteon. The two other kinds of decisions are also represented in the ontology, but not in an evident way. Property decisions are represented in Arteon as the resulting decisions from conditions over quality attributes or element attributes, for example, all the decisions made because of the condition to have OSS license. Executive decisions are represented in Arteon as the resulting decisions imposed by restrictions that come from the software requirements, in particular the requirements unrelated to the software quality, for example, a software requirement says that the DBMS should be Oracle, because the company has a deal with Oracle.

There are other works that propose methods (e.g., [24]) and conceptual models of AK (e.g., [1, 7]), but there is not enough space in the present paper to properly comment on the commonalities and differences between approaches. As a general comment we would say that each one is designed with particular objectives in mind, some focus on management and the actors that take part in architectural design, others are oriented to document decisions, etc.

6 Conclusions

One of the most known Kruchten's statements is "the life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in dark". The lack of knowledge is one of the reasons to produce suboptimal decisions. For example, the architect may not know all the effects of using some technology or architectural pattern: it may need of other components to work correctly (e.g., some of them may be incompatible with other architectural decisions), it may have unexpected effects in the overall evaluation of some quality attributes (e.g., lowers the resource utilization efficiency). Also, the lack of knowledge may cause a worse situation when some alternative is not considered because it is unknown to the architect.

To improve this situation we presented in this paper the Quark method, a lightweight method based in the Arteon ontology to support architects during the architectural decision making.

7 Future Work

Quark can work as standalone method for a decision making support system for architectural design, but we envisioned the conversion of the resulting set of decisions into architectural views and/or models, and then into the actual software architecture implementation [4].

We are implementing an Eclipse-based tool, ArchiTech [2], which will become a proof of concept for the Quark method. This same tool is already capable to manage the AK as it is defined in Arteon. One of the key features of this tool will be the ability to monitor the quality attributes at the very moment that an architectural decision is made.

AK acquisition and maintenance is the "elephant in the room". Neither this nor other works done around AK will become more than toy tools if this issue is not resolved. Our position is that the only way to acquire and maintain such amount of information is making architects active participants. We started conducting several empirical studies based on interviewing software architects to acquire this knowledge, but large networks of knowledge are required (e.g., Stack Overflow had been a successful knowledge base for developers). We should look forward and propose knowledge networks that allow reasoning beyond text searches.

8 Acknowledgments

This work has been partially supported by the Spanish MICINN project TIN2010-19130-C02-02. Thanks to Oriol Collell for his comments.

References

[1] A. Akerman and J. Tyree. Using ontology to support development of software architectures. *IBM Syst. J.*, 45:813–825, 2006.

[2] D. Ameller, O. Collell, and X. Franch. Reconciling the 3-layer architectural style with a plug-in-based architecture: the Eclipse case. In *TOPI (ICSE)*, 2011.

[3] D. Ameller and X. Franch. Ontology-based Architectural Knowledge representation: structural elements module. In *IWSSA (CAiSE)*, 2011.

[4] D. Ameller, X. Franch, and J. Cabot. Dealing with Non-Functional Requirements in Model-Driven Development. In *RE*, 2010.

[5] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168(1-2):70–118, 2005.

[6] L. Chung, B. Nixon, and E. Yu. *Non-functional requirements in software engineering*. Kluwer Academic, 2000.

[7] R. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen. Architectural Knowledge: Getting to the Core. In *QoSA*, 2007.

[8] J. Evermann and J. Fang. Evaluating ontologies: Towards a cognitive measure of quality. *Information Systems*, 35(4):391–403, 2010.

[9] D. Gasevic, D. Djuric, V. Devedzic, and V. Damjanovi. Converting UML to OWL ontologies. In *WWW Alt.*, pages 488–489, 2004.

[10] A. Gehlert and A. Metzger. Quality Reference Model for SBA, 2009.

[11] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43:907–928, 1995.

[12] N. Guarino. Some Ontological Principles for Designing Upper Level Lexical Resources. *CoRR*, cmp-lg/9809002, 1998.

[13] G. Guizzardi, G. Wagner, and H. Herre. On the Foundations of UML as an Ontology Representation Language. In *EKAW*, 2004.

[14] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America. A general model of software architecture design derived from five industrial approaches. *JSS*, 80(1):106–126, 2007.

[15] L. Hsu and R. Obe. Cross Compare of SQL Server, MySQL, and PostgreSQL `http://www.postgresonline.com/journal/archives/51-Cross-Compare-of-SQL-Server,-MySQL,-and-PostgreSQL.html`, 2008.

[16] ISO/IEC. Product quality – Part 1: Quality model, 2001.

[17] ISO/IEC. Software product Quality Requirements and Evaluation (SQuaRE), 2005.

[18] P. Kroll and P. Kruchten. *The rational unified process made easy: a practitioner's guide to the RUP*. Addison-Wesley, 2003.

[19] P. Kruchten, R. Capilla, and J. C. Duenas. The Decision View's Role in Software Architecture Practice. *IEEE Soft.*, 26:36–42, 2009.

[20] P. Kruchten, P. Lago, and H. van Vliet. Building Up and Reasoning About Architectural Knowledge. In *QoSA*, 2006.

[21] A. Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing.* LNCS. Springer, 1980.

[22] J. Robertson and S. Robertson. Volere: Requirements Specification Template. Technical report, Atlantic Systems Guild, 2010.

[23] A. Tang, M. Babar, I. Gorton, and J. Han. A survey of architecture design rationale. *JSS*, 79(12):1792–1804, 2006.

[24] A. Tang, J. Han, and R. Vasa. Software Architecture Design Reasoning: A Case for Improved Methodology Support. *IEEE Soft.*, 26(2):43–49, 2009.

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