Intertwining Relationship Between Requirements, Architecture, and Domain Knowledge

Azadeh Alebrahim, Maritta Heisel

paluno – The Ruhr Institute for Software Technology,
University of Duisburg-Essen, Germany
Email: firstname.lastname@paluno.uni-due.de

***Abstract*—In requirements engineering, properties of the environment and assumptions about it, called *domain knowledge*, need to be captured in addition to exploring the requirements. Despite the recognition of the significance of capturing the required domain knowledge, domain knowledge might be missing, left implicit, or be captured inadequately during the software development process, causing incorrect specifications and software failure. Domain knowledge affects the elicitation and evolution of requirements, the evolution of software architectures, and related design decisions. Conversely, requirements and design decisions affect the elicitation and modification of domain knowledge. In this paper, we propose the iterative capturing and co-developing of domain knowledge with requirements and software architectures. We explicitly discuss the effects of requirements and design decisions on domain knowledge and illustrate this relationship with examples drawn from our research, where we had to go back and forth between requirements, domain knowledge, and design decisions.**

***Keywords*–quality requirements; requirements engineering; domain knowledge; design decisions; software architecture.**

I. Introduction

It is acknowledged that there is an iterative interplay between requirements and software architecture [1]. Requirements cannot be considered in isolation and should be co-developed with architectural descriptions iteratively, known as *Twin Peaks* as proposed by Nuseibeh [2], to support the creation of sound architectures and correct requirements [3].

The system-to-be comprises the software to be built and its surrounding environment such as people, devices, and existing software [4]. The environment consists of the part of the real world into which the software will be integrated. According to Jackson [5], requirements expressing wishes are properties of the environment that are to be guaranteed by the software (*machine* in Jackson's terminology), whereas there is another class of environment properties that are guaranteed by the environment. Such environment properties and also assumptions about the environment are known as *domain knowledge* [4][6]. Domain properties typically correspond to physical laws. They are invariable, no matter how we build the software. Assumptions have to be satisfied by the environment. They are not guaranteed to be true in every case. For example, when we build a traffic light system to prevent accidents, the assumption is that *drivers stop when they see a red traffic light*. Otherwise, a traffic light system cannot prevent accidents.

Specifications describe the machine and not the environment. Domain knowledge supports the refinement of requirements into implementable specifications [7]. Hence, in requirements engineering domain knowledge needs to be captured in addition to exploring the requirements [4][6].

Despite the recognition of the significance of capturing the required domain knowledge, it might be missing, left implicit, or be captured inadequately during the software development process [4]. Several requirements engineering methods exist, e.g., for security. Fabian et al. [8] concluded in their survey about these methods that there is no state of the art considering domain knowledge yet. Hooks and Farry report on a project in which 49% of requirements errors were due to incorrect domain knowledge [9]. In Colombia in December 1995, capturing inadequate assumptions about the environment of the flight guidance software led to the crash of a Boeing 757 [10].

As software systems become larger and more complex, explicitly capturing domain knowledge becomes crucial. The consideration of domain knowledge is particularly essential when talking about quality requirements since quality requirements such as performance and security rely on specific constraints and assumptions for their satisfaction. For instance, performance is concerned with available *resources* (such as CPU and memory) to process the *workload* [11]. Such resources employed by the software-to-be have specific characteristics such as speed and size that might constrain the satisfaction of quality requirements. Hence, one must explicitly define under which constraints and assumptions a quality requirement will be fulfilled.

We are convinced that during the software development process, domain knowledge is not only used in requirements engineering for obtaining adequate specifications, it also has to be captured during the design phase when selecting patterns and mechanisms or when making design decisions. There are new assumptions and requirements associated with each pattern and quality-specific solution, which have to be considered when deciding on this solution. For example, selecting *asymmetric encryption* as a solution for a confidentiality requirement demands other assumptions regarding the keys and their distribution than *symmetric encryption*.

We distinguish between the domain knowledge related to the problem world, required for obtaining correct specifications, and the domain knowledge which is associated with properties and assumptions about the solution world, required for applying a pattern or mechanism properly. We call the former *Problem-Specific Domain Knowledge (PSDK)*, which is part of the problem peak in the twin peaks model [2] and the latter *Solution-Specific Domain Knowledge (SSDK)*, as a part of the solution peak. PSDK and SSDK have to be captured for functional requirements and their corresponding functional solutions as well as for quality requirements and their corresponding quality-specific solutions. In this paper, we focus on PSDK and SSDK related to quality requirements and the corresponding solutions.

As an example for the PSDK, we consider the performance. Performance requirements typically describe the time needed for conducting a task (response time). In order to identify and analyze potential performance problems, the *workload* and the available *resources* as performance-relevant domain knowledge have to be captured explicitly (see Section IV-A). As an example for the SSDK, we consider the *symmetric encryption* as a security-specific solution candidate for achieving a confidentiality requirement. This security solution demands new assumptions regarding the *secret key* that have to be elicited explicitly (see Section IV-B).

Capturing SSDK facilitates future design decisions and architecture evolution since the architect knows the consequences of each design decision through necessary assumptions and requirements and can therefore play “what if” scenarios. Consequently, we consider the SSDK as a set of assumptions and facts that builds the foundation for making design decisions. The SSDK represents one input to the design process.

Domain knowledge affects the elicitation and evolution of requirements as well as the evolution of software architectures and related design decisions. Conversely, modification of initial requirements or taking design decisions might lead to capturing new domain knowledge or modifying the existing one. Therefore, apart from the concurrent and iterative development of requirements and architecture, there is an iterative interplay within the problem peak between the requirements and the PSDK and within the solution peak between the design decisions and SSDK. In this paper, we describe these intertwining relationships and propose to co-develop the domain knowledge together with requirements and software architecture. We illustrate these relationships with examples drawn from our research, where we had to go back and forth within each peak and between two peaks.

The contribution of this paper is 1) emphasizing the importance of domain knowledge, particularly quality-relevant domain knowledge and the need for capturing and reusing it in the software development process in a systematic manner, 2) explicitly considering PSDK as part of the problem peak and SSDK as part of the solution peak, and 3) elaborating the intertwining relationship of domain knowledge with requirements and design decisions.

The rest of this paper is structured as follows: In the following, we discuss the related work in Section II. The smart grid scenario as a running example is introduced in Section III. In Section IV, we first present our idea of co-developing requirements, architecture, and domain knowledge. Then, we give examples on how these artifacts affect each other. We conclude the paper in Section V.

II. Related Work

Beside the common and traditional approaches utilizing requirements for creating the software architecture, there have been increasing efforts regarding the intertwining relationship between requirements and architecture. De Boer and van Vliet [12] review different opinions regarding this relationship between requirements as problem description and software architecture as solution description. They propose a closer collaboration between the two communities to profit from the research results that each community provides.

Ferrari and Madhavji [13] conduct an empirical study to analyze the impact of requirements knowledge and experience on the software architecture. Their findings show that architects having knowledge and experience in requirements engineering perform better in terms of architecture quality.

A number of approaches exist that explore the impact of software architecture and design decisions on requirements engineering [14][15][16][17]. Durdik et al. [14] discuss how the results of design decisions can be used to drive requirement elicitation and prioritization. Koziolek [15] proposes to use the feedback from architecture evaluation and design space exploration for prioritizing quality requirements. An exploratory study has been conducted to analyze to what extent an existing software architecture affects requirements engineering [16]. The authors found four types of architectural effects on requirements decisions, namely *enabler*, *constraint*, *influence*, and *neutral*. Woods and Rozanski [17] report on their experience regarding the relationship between the system requirements and software architecture. They propose to use the architecture design to constrain the requirements to a set which is achievable, to frame the requirements, and to inspire new requirements.

Van Lamsweerde [4] and Jackson [6] underline the importance of eliciting domain knowledge in addition to the elicitation of requirements to obtain correct specifications. This corresponds to capturing the PSDK in this paper. Babar et al. [18] emphasize the significance of capturing architecture knowledge for software development. This corresponds to eliciting the SSDK eplicitly.

However, none of these approaches investigate the intertwining relationship between these artifcats. Moreover, to the best of our knowledge, there is no approach exploring the types of effects of requirements, domain knowledge, and design decisions on each other such as capturing new domain knowledge and requirement or modifiying the existing ones.

III. Smart Grid Example

We illustrate our proposed idea through the example of a smart grid system, based on the protection profile that was issued by the Bundesamt für Sicherheit in der Informationstechnik [19]. To use energy in an optimal way, smart grids make it possible to couple the generation, distribution, storage, and consumption of energy. Smart grids use information and communication technology, which allows for financial, informational, and electrical transactions.

Figure 1 shows the simplified context of a smart grid system based on the protection profile [19]. Gateway, the Target Of Evaluation (TOE) is used for collecting, storing, and providing meter data from one or more smart meters which are responsible for one or more commodities, such as electricity, gas, water, or heat. The Local Metrological Network (LMN) represents the in-house data communication netweork which interconnects the smart meters to the gateway. The term Meter refers to a device which is comparable to

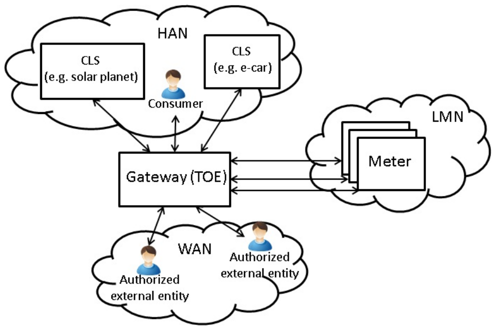

Figure 1. The context of a smart grid system based on [19].

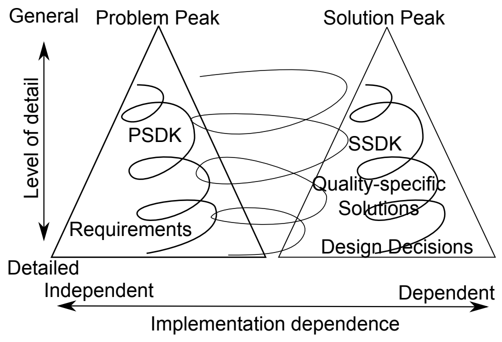

Figure 2. Twin peak model including interrelationships within each peak.

a classical meter with additional functionality. Controllable Local Systems (CLS) are devices of the consumer, such as air condition, solar planet, and intelligent household appliances containing IT-components. They do not belong to the smart metering system. They are in the Home Area Network (HAN) of the consumer. HAN is the in-house data communication netweork interconnecting domestic equipments.

Table I shows an excerpt of terms specific to the smart grid domain taken from the protection profile that are relevant to understand the requirements.

We focus in this paper on the functional requirement *"The smart meter gateway shall submit processed meter data to authorized external entities (RQ4)"*, the security requirements *"Integrity (RQ10)/ Confidentiality (RQ11)/ Authenticity (RQ12) of data transferred in the WAN shall be protected"*, and the performance requirement *"The time to retrieve meter data from the smart meter and publish it through WAN shall be less than 5 seconds (RQ24)"*. We derived these requirements from the protection profile [19] and the report "Requirements of AMI" [20].

IV. Interplay of Requirements, Architecture, and Domain Knowledge

In this section, we describe how requirements, architecture, and domain knowledge affect each other. Figure 2 shows the twin peaks model. It addresses the problem in some linear software development approaches in which on the one hand requirements are elicited, analyzed, and specified in isolation without considering the impact of architecture artifacts. On the other hand design decisions are made without managing the conflicts and making necessary changes in the requirements. The twin peaks model emphasizes the intertwining relationship between requirements and architecture. The spiral between the problem peak and the solution peak in Figure 2 illustrates this relationship between the problem world and the solution world [2].

In this paper, we emphasize the need to capture and specify requirements and PSDK as it affects the elicitation and evolution of requirements. Equally, evolving requirements might have an affect on previously captured PSDK. The spiral in the first peak, the problem peak in Figure 2 represents the synergistic relationship between the requirements and the PSDK. Similarly, architecture artifacts and SSDK exhibit such intertwining relationship as design decisions and trade-offs might require the elicitation of new SSDK or modification of the existing one. Figure 2 shows the relationship between the design decisions and the SSDK by means of the spiral in the solution peak.

In the following, we describe the impact of requirements and PSDK on each other in the problem peak. Then, the impact of design decisions and SSDK on each other is described. We give examples of such effects using the smart grid example.

A. Interplay of Requirements and PSDK

For developing software that achieves its desired quality requirements, additional information (PSDK) for each quality requirement must be explicitly elicited. As an example, we consider the performance. As mentioned in Section I, the *workload* and the available *resources* as performance-relevant domain knowledge have to be elicited and incorporated into existing requirement models explicitly. The workload is described by triggers of the system, representing requests from outside or inside the system. Workload exhibits the characteristics of the system use. It includes the number of concurrent users and their arrival pattern. The arrival pattern can be periodic (e.g., every 10 milliseconds), stochastic (according to a probabilistic distribution), or sporadic (not to capture by periodic or stochastic characterization) [21]. Processing the requests requires resources. Each resource has to be described by its type in the system, such as CPU, memory, and network, its utilization, and its capacity, such as the transmission speed for a network.

Performance-relevant domain knowledge can be gained from performance experts and analysts. In this paper, we do not aim at proposing approaches on how to elicit and model performance-relevant domain knowledge, but at emphasizing the need for eliciting domain knowledge as additional information to the quality requirements and annotating it properly

TABLE I. AN EXCERPT OF RELEVANT TERMS FOR THE SMART GRID

Gateway	represents the central communication unit in a *smart metering system*. It is responsible for collecting, processing, storing, and communicating *meter data*.
Meter data	refers to meter readings measured by the meter regarding consumption or production of a certain commodity.
Smart meter	represents the device that measures the consumption or production of a certain commodity and sends it to the gateway.
Authorized external entity	could be a human or IT unit that communicates with the gateway from outside the gateway boundaries through a *Wide Area Network (WAN)*.
WAN	WAN provides the communication network that interconnects the gateway with the outside world.

TABLE II. INSTANTIATED PSDK TEMPLATE FOR *RQ24* (AND MAPPING TO THE MARTE PROFILE)

Quality: Performance, Quality Requirement: RQ24					
PSDK Template					**Mapping to MARTE**
PSDK Description		**Possible Values**	**Value**	**Range of Value**	**Property**
Number of concurrent users		Natural	50		GaWorkloadEvent. pattern. population
Arrival pattern		ArrivalPattern	closed		GaWorkloadEvent. pattern
Data size		DataSize (bit, Byte, KB, MB, GB)	640 MB		GaStep. msgSize
Memory	capacity	DataSize (bit, Byte, KB, MB, GB)	-		HwMemory. memorySize
	latency	Duration (s, ms, min,hr, day)	-		HwMemory. timing
Network	bandwidth	DataRate (b/s, Kb/s, Mb/s)	2.4 Kb/s		HwMedia. bandWidth
	latency	Duration (s, ms, min,hr, day)	-		HwMedia. packetTime
CPU	speed	Frequency (Hz, kHz, MHz, GHz)	470 MHz		HwProcessor. frequency
	Number of cores	Natural	1		HwProcessor. nbCores

in the requirement models as initially proposed in our previous work [22]. We propose to document the PSDK for the corresponding software qualities as structured templates. We call such templates *PSDK Templates* to be instantiated separately for each type of software quality. The instantiated template should be known to the requirements engineer or performance analyst to analyze whether a particular performance requirement can be satisfied or not. In order for the analyst to be able to determine "whether the meter data can be transferred through WAN within 5 seconds", (s)he needs PSDK involving the number of concurrent users in the system, the bandwidth of the network, the CPU speed and the core numbers, and the data volume which is transferred over the network. We exemplify the instantiation of the PSDK template for the performance requirement *RQ24* according to the information contained in the existing documents for the smart grid application [19][20]. Table II shows the instantiated PSDK template (see column *Value*). The columns *PSDK Description* and *Possible Values* show the domain knowledge to be elicited for performance and its possible values. We may extend the template with the column *Range of Value* showing the possible range of values. Such information can be obtained from documents or involved stakeholders. We make use of this column later on to modify (strengthen or relax) the PSDK as a conflict resolution strategy.

In addition to the template, a suitable notation for integrating quality-relevant domain knowledge in the requirement models can be selected to be used for requirements analysis in a model-based approach. We select the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [23] adopted by OMG consortium that allows us to annotate performance-relevant domain knowledge in a UML-based modeling approach. The column *Property* shows the corresponding stereotypes from the MARTE profile.

We describe the need for eliciting PSDK and documenting it in PSDK templates when we elicit quality requirements by the example of a performance requirement. Requirements might affect the PSDK when they are modified for any reason. In such a case, the PSDK has to be checked for possible modifications.

One important source for changes in the requirements is detecting and resolving conflicts among requirements. Such changes might cause changes in the PSDK [24]. Requirements conflicts can typically be resolved on the requirement level by relaxing the conflicting requirements or on the architecture level by relaxing the corresponding solutions. In some cases, only potential conflicts among requirements can be detected and not the genuine ones [25]. Hence, in such cases the conflict resolution shall preferably be postponed to the architecture level, where more details are available for detecting the genuine requirement conflicts and resolving them. Resolving requirements conflicts by relaxing requirements might lead to the modification of the existing PSDK (e.g., assumptions). Resolution of conflicting requirements on the requirement level is one source for updating existing domain knowledge. One might make a trade-off between a performance requirement and a security requirement by relaxing the performance requirement.

In our previous work [24], we proposed a method for detecting and resolving conflicts among performance and security requirements, as security has mostly a deep impact on the performance of the whole system. The reason is that mechanisms and patterns for satisfying security requirements, such as encryption or Message Authentication Code (MAC) are time-consuming. The general principle of our method for detecting interactions among requirements is using the structure of requirement models to identify trade-off points, where security and performance requirements might interact. After we have identified pairs of conflicting requirements, we have made trade-offs by relaxing one or both conflicting requirements to resolve the conflict. As an example, we consider two requirements *RQ11* and *RQ24* that we identified as conflicting. One option for relaxing the performance requirement is modifying the performance-relevant domain knowledge. Typically, domain knowledge consists of facts (domain properties in [4]) that we cannot change, relax, and negotiate and assumptions that can be changed, relaxed, and negotiated [4]. For resolving the conflict, we might modify (relax or strengthen) the assumptions. For example, the *number of concurrent users* is not a fixed property. It is an assumption and can be modified, when the performance requirement cannot be achieved with this assumption. Hence, we modify *number of concurrent users* by reducing it to a number less than 50. The same holds true for *data size*, which has to be reduced to less than 640 MB, *network bandwidth*, which has to be relaxed to more than 2.4 Kb/s (see the instantiated PSDK in Table II). The rest of the properties are either fixed (can be considered as facts) or irrelevant for the corresponding requirement, or unknown and thus cannot be considered for the modification process. We document such modifications in the column *Range of Value*. Note that changes in the requirements or PSDK should be negotiated with the stakeholders.

B. *Interplay of Design Decisions and SSDK*

Quality-specific patterns and mechanisms for performance such as *load balancer* and *master worker* [26] and for security such as *Role-Based Access Control (RBAC)* and *encryption* [27] represent solution candidates for achieving quality

TABLE III. IMPACT OF SECURITY-SPECIFIC SOLUTION *ASYMMETRIC ENCRYPTION* ON REQUIREMENTS AND SSDK

Security-specific Solution	
Name	Asymmetric Encryption
Brief Description	The plaintext is encrypted using the public key and decrypted using the private key.
Quality Requirement to be achieved	Security (confidentiality)
Positively affected quality requirement(s)	-
Negatively affected quality requirement(s)	Performance
Necessary Conditions	
Quality Requirement □ SSDK □	Integrity of public key during transmission shall be/is preserved.
Quality Requirement □ SSDK □	Confidentiality of private key during transmission shall be/is preserved.
Quality Requirement □ SSDK □	Integrity of private key during transmission shall be/is preserved.
Quality Requirement □ SSDK □	Confidentiality of private key during storage shall be/is preserved.
Quality Requirement □ SSDK □	Integrity of private key during storage shall be/is preserved.
Quality Requirement □ SSDK □	Integrity of public key during storage shall be/is preserved.

TABLE IV. IMPACT OF SECURITY-SPECIFIC SOLUTION *SYMMETRIC ENCRYPTION* ON REQUIREMENTS AND SSDK

Security-specific Solution	
Name	Symmetric Encryption
Brief Description	The plaintext is encrypted and decrypted using the same secret key.
Quality Requirement to be achieved	Security (confidentiality)
Positively affected quality requirement(s)	-
Negatively affected quality requirement(s)	Performance
Necessary Conditions	
Functional Requirement □ SSDK □	Secret key shall be/is distributed.
Quality Requirement □ SSDK □	Confidentiality and integrity of secret key distribution distribution shall be/is preserved.
Quality Requirement □ SSDK □	Confidentiality of secret key during transmission shall be/is preserved
Quality Requirement □ SSDK □	Confidentiality of secret key during storage shall be/is preserved.
Quality Requirement □ SSDK □	Integrity of secret key during storage shall be/is preserved.
Quality Requirement □ SSDK □	Confidentiality of encryption machine shall be/is preserved.
Quality Requirement □ SSDK □	Integrity of encryption machine shall be/is preserved.
Quality Requirement □ SSDK □	Confidentiality of plaintext shall be/is preserved.

requirements. By exploring the solution space for achieving quality requirements, we require to know which assumptions and facts are to be considered, and which new functional and quality requirements are to be elicited when deciding on a particular solution. Generally speaking, all information that can affect the requirements and related domain knowledge has to be documented.

We propose to provide a template consisting of two parts for the analysis of quality-specific solutions and their impacts on requirements and domain knowledge. We analyze the impact of the solution candidate on the problem space (i.e., requirements), and on the SSDK. Such a template helps us later on when selecting a particular solution candidate to keep track on changes in the requirements and domain knowledge. It also supports an unexperienced architect in understanding the impact of design decisions on the entire system, particularly on the achievement of quality requirements.

The first part of the template (see Table III) accommodates information about the quality-specific solution itself, such as name (*Name*), description (*Brief Description*), the quality requirement which will be achieved when selecting this solution (*Quality Requirement to be achieved*), and the quality requirements which are positively or negatively affected by this solution ((*Positively affected Quality Requirement*), (*Negatively affected Quality Requirement*)). For example, improving the security may result in decreasing the performance. Hence, the impact of each security-specific solution on other quality requirements has to be captured in the first part of the template.

The second part of the template captures and documents necessary conditions which have to be addressed when selecting this solution. Necessary conditions have to be addressed either as *Functional Requirement*, *Quality Requirement*, or as *SSDK*. We elicit the necessary conditions as requirements if the software to be built shall achieve them. In contrast, assumptions have to be satisfied by the environment [4]. Hence, the necessary conditions have to be captured as assumptions (one part of the SSDK) if they have to be satisfied by the environment. Assumptions are not guaranteed to be true in every case. For the case that we assume the environment (not the machine) takes the responsibility for meeting them, we capture them as assumptions. This should be negotiated with the stakeholders and documented properly. Tables III and IV show such a template for the security-specific solutions *asymmetric encryption* and *symmetric encryption*.

We describe Table IV in more detail. After capturing the basic information about the security-specific solution *symmetric encryption* in the first part, in the second part we elicit new requirements and capture new assumptions that arise with the solution, such as *secret key shall be/ is distributed*. Eliciting this condition results in thinking about security issues concerned with it, such as *confidentiality and integrity of secret key distribution shall be/is preserved*. If we require that the software we build is responsible for preserving the confidentiality and integrity of the secret key not only during the transmission but also during the storage, we have to capture these as requirements. This is the reason why the necessary conditions are presented as checkboxes to be selected by checking the relevant checkbox as requirement or SSDK.

As mentioned before, one option for resolving interactions among requirements is making trade-offs between corresponding quality-specific solutions. To this end, one or both quality-specific solutions have to be relaxed. Making such design decisions requires eliciting or updating SSDK and requirements associated with the particular solution. For example, selecting a *symmetric encryption* for achieving a confidentiality requirement instead of an *asymmetric encryption* demands other assumptions and requirements with respect to the required keys and key distribution as shown in the corresponding Templates (Tables III and IV). To demonstrate this idea, we consider the *asymmetric encryption* as the initial security-specific solution, which is selected for satisfying the security requirement *RQ11*. *RQ11* is concerned with transmitting meter data through the WAN in a confidential way. *Asymmetric encryption* provides sufficient protection during transmitting meter data through the WAN so that the confidentiality requirement can be achieved. However, by applying our method for detecting interactions among quality requirements in our previous work [24], we detected a conflict with the performance requirement *RQ24*. Hence, *RQ24* cannot be achieved in less than 5 seconds when keeping the security-specific solution *asymmetric encryption* for meeting the security requirement *RQ11*. We have to decide for a strategy to resolve the conflict. We can relax the performance requirement by increasing the response time as one possible resolution strategy. Strengthening or relaxing the PSDK for example by raising the *network bandwidth* or by decreasing the *data size* is possible as well, as described in Section IV-A. Such strategies are at the cost of the performance requirement *RQ24* and can only be used if the security requirement *RQ11* has a higher priority. Here, we assume that the performance requirement *RQ24* has a higher priority. Hence, we have to make a trade-off by relaxing the security-specific solution. This can be achieved by selecting another security-specific solution, which is faster. We decide on *symmetric encryption* instead of *asymmetric encryption*. Symmetric encryption is faster than the asymmetric encryption. It, however, demands other requirements and SSDK. In contrast to the asymmetric encryption, which uses different keys for encrypting and decrypting data, the symmetric encryption uses only one key. Thus, we have to care about the key distribution. Hence, this design decision leads to changes in the requirements as well as in the domain knowledge as shown in Table IV.

V. Conclusion and Future Work

In this paper, we underlined the importance of capturing and documenting domain knowledge, particularly quality-relevant domain knowledge for the problem space as well as for the solution space. More importantly, we described how requirements, design decisions and domain knowledge affect each other. Eliciting and updating requirements causes elicitation and modification of PSDK. Capturing and evolving solutions on the architecture level requires eliciting and modifying SSDK and requirements. Hence, domain knowledge should be captured and developed iteratively and incrementally with requirements and architecture to achieve adequate specifications.

In order to be able to argue that the requirements will be satisfied under specific constraints and assumptions, PSDK should be traceable to the requirements [4]. Moreover, design decisions should be traceable to SSDK and requirements to reflect the changes in design decisions and software architecture to the problem peak. Keeping the changes consistent in requirements, domain knowledge (PSDK and SSDK), and software architecture is challenging. Model-based approaches enables us to provide support by keeping such trace information in the model. In the future, we want to provide traceability links between these artifacts in our models to keep track of the changes emerging in one peak which cause changes in the other peak.

Acknowledgments.

This research was partially supported by the German Research Foundation (DFG) under grant numbers HE3322/4-2. We thank Thein Than Tun for his useful feedback on our work.

References

[1] M. Mirakhorli and J. Cleland-Huang, “Traversing the twin peaks,” IEEE Software, vol. 30, no. 2, 2013, pp. 30–36.

[2] B. Nuseibeh, “Weaving together requirements and architectures,” IEEE Computer, vol. 34, no. 3, 2001, pp. 115–117.

[3] M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam, “Your "What" Is My "How": Iteration and Hierarchy in System Design,” IEEE Software, vol. 30, no. 2, 2013, pp. 54–60.

[4] A. Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, 2009.

[5] M. Jackson, “The meaning of requirements,” Ann. Softw. Eng., vol. 3, Jan. 1997, pp. 5–21.

[6] ——, Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, 2001.

[7] P. Zave and M. Jackson, “Four dark corners of requirements engineering,” ACM Trans. Softw. Eng. Methodol., vol. 6, 1997, pp. 1–30.

[8] B. Fabian, S. Gürses, M. Heisel, T. Santen, and S. Schmidt, “A comparison of security requirements engineering methods,” Requirements Engineering – Special Issue on Security Requirements Engineering, vol. 15, 2010, pp. 7–40.

[9] I. F. Hooks and K. A. F., Customer-centered Products: Creating Successful Products Through Smart Requirements Management. AMACOM, 2001.

[10] F. Modugno, N. Leveson, J. Reese, K. Partridge, and S. Sandys, “Integrated safety analysis of requirements specifications,” Requirements Engineering, 1997, pp. 65–78.

[11] L. Bass, M. Klein, and F. Bachmann, “Quality attributes design primitives,” Software Engineering Institute, Tech. Rep., 2000.

[12] R. C. de Boer and H. van Vliet, “Controversy corner: On the similarity between requirements and architecture,” J. Syst. Softw., vol. 82, no. 3, Mar. 2009, pp. 544–550.

[13] R. Ferrari and N. H. Madhavji, “The Impact of Requirements Knowledge and Experience on Software Architecting: An Empirical Study,” in Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007, p. 16.

[14] Z. Durdik, A. Koziolek, and R. Reussner, “How the understanding of the effects of design decisions informs requirements engineering,” in 2nd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks), 2013, pp. 14–18.

[15] A. Koziolek, “Research preview: Prioritizing quality requirements based on software architecture evaluation feedback,” in Requirements Engineering: Foundation for Software Quality. Springer, 2012, vol. LNCS7195, pp. 52–58.

[16] J. Miller, R. Ferrari, and N. Madhavji, “Architectural effects on requirements decisions: An exploratory study,” in Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2008, pp. 231–240.

[17] E. Woods and N. Rozanski, “How Software Architecture can Frame, Constrain and Inspire System Requirements.” in Relating Software Requirements and Architectures. Springer, 2011, pp. 333–352.

[18] M. Ali Babar, I. Gorton, and R. Jeffery, “Capturing and Using Software Architecture Knowledge for Architecture-Based Software Development,” in Proceedings of the 5th International Conference on Quality Software, ser. QSIC ’05. IEEE Computer Society, 2005, pp. 169–176.

[19] H. Kreutzmann, S. Vollmer, N. Tekampe, and A. Abromeit, “Protection profile for the gateway of a smart metering system,” BSI, Tech. Rep., 2011.

[20] Remero et al., “D1.1 Requ. of AMI,” OPEN meter proj., Tech. Rep., 2009.

[21] L. Bass, P. Clemens, and R. Kazman, Software architecture in practice. Addison-Wesley, 2003.

[22] A. Alebrahim, M. Heisel, and R. Meis, “A structured approach for eliciting, modeling, and using quality-related domain knowledge,” in Proceedings of the 14th International Conference on Computational Science and Its Applications (ICCSA), ser. LNCS. Springer, 2014, vol. 8583, pp. 370–386.

[23] UML Revision Task Force, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2011, http://www.omg.org/spec/MARTE/1.0/PDF [retrieved: 2014-08-07].

[24] A. Alebrahim, C. Choppy, S. Faßbender, and M. Heisel, “Optimizing functional and quality requirements according to stakeholders’ goals,” in System Quality and Software Architecture (SQSA). Elsevier, 2014, pp. 75–120.

[25] A. Alebrahim, S. Faßbender, M. Heisel, and R. Meis, “Problem-Based Requirements Interaction Analysis,” in Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), ser. LNCS, vol. 8396. Springer, 2014, pp. 200–215.

[26] C. Ford, I. Gileadi, S. Purba, and M. Moerman, Patterns for Performance and Operability. Auerbach Publications, 2008.

[27] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, Security patterns: integrating security and systems engineering. John Wiley & Sons, 2005.