Problem-oriented Security Patterns for Requirements Engineering

Azadeh Alebrahim, paluno – The Ruhr Institute for Software Technology, Germany
Maritta Heisel, paluno – The Ruhr Institute for Software Technology, Germany

Security is one essential quality requirement that needs to be addressed during the software development process. While quality requirements such as security are supposed to be the architectural drivers, architecture solutions such as security patterns represent design decisions on the architecture and design levels that in turn might constrain quality requirements significantly. Thus, knowledge which is gained in the solution space, for example from security patterns, should be reflected in the requirements engineering to obtain sound architectures and correct requirements. We propose to reuse security patterns in the requirements engineering in a systematic manner to equip requirement models with security solution approaches early in the software development process. To this end, we propose *problem-oriented security patterns*. Each problem-oriented security pattern consists of a three-part graphical pattern representing the functional problem which describes the functional requirement annotated with a security requirement, the solution to the security requirement, and the composition of them. In addition, we provide a template that captures the affect of applying the security solution on the requirement models.

Categories and Subject Descriptors: I.5.2 [**Pattern Recognition**]: Design Methodology—*pattern analysis*; D.2.11 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*; K.6.5 [**Management of Computing and Information Systems**] Security and Protection

General Terms: Design, Security

Additional Key Words and Phrases: Security patterns, problem frames, requirements engineering

ACM Reference Format:

Alebrahim, A. and Heisel, M. 2014. Problem-oriented Security Patterns for Requirements Engineering jn 2, 3, Article 1 (May 2010), 17 pages.

1. INTRODUCTION

Many software systems fail to achieve their quality objectives due to neglecting quality requirements at the beginning of the software development life cycle [Chung et al. 2000]. In order to obtain a software that achieves not only its required functionality but also the desired quality properties, it is necessary to consider both types of requirements, functional and quality ones, early enough in the software development life cycle. Security is one essential quality requirement to be considered during the software development process.

Architecture solutions provide a means to satisfy quality requirements. While requirements are supposed to be the architectural drivers, architecture solutions represent design decisions on the architecture level that in turn affect the achievement of quality requirements significantly. Decisions made in the design phase could constrain the achievement of initial requirements, and thus could change them. Hence, requirements cannot be considered in isolation and should be co-developed with architectural descriptions iteratively known as *Twin Peaks* as proposed by Nuseibeh [Nuseibeh 2001] to support the creation of sound architectures and correct requirements [Whalen

Author's address: Azadeh Alebrahim, Oststrasse 99, 47057 Duisburg, Germany; email: azadeh.alebrahim@paluno.uni-due.de; Maritta Heisel, Oststrasse 99, 47057 Duisburg, Germany; email: maritta.heisel@paluno.uni-due.de

et al. 2013]. In this paper, we follow the concept of the twin peaks model, in which we refine the security issues at the requirement level by reusing the knowledge located in the solution space to bridge the gap between security problems and security solutions.

Patterns describe solutions for commonly recurring problems in software development. They are defined for different software development phases, such as Jackson's problem frames [Jackson 2001] and Fowler's analysis patterns [Fowler 1996] for the requirements level, architectural styles [Shaw and Garlan 1996] for the architecture level, and design patterns [Gamma et al. 1995] for the design level. Schumacher et al. [Schumacher et al. 2005] provide solutions to security problems represented as patterns that can be applied during the design and implementation phases of software development processes.

In this paper, we represent patterns that can be used during the requirements engineering phase to refine the requirement models. These patterns reuse the knowledge gained in the design phase (solution space) from security patterns to equip the requirement models (problem space) with security solution approaches early in the software development process. That is, we do not only elicit and model security requirements, but also provide solution approaches for these requirements. This supports the seamless transition of requirement descriptions to architectural descriptions.

As a basis for requirements analysis, we use the problem frames approach [Jackson 2001]. It is a requirements engineering method providing a means for understanding, analyzing, and describing the software development problems. The problem frames approach suggests to decompose the overall software problem into simple subproblems. Each subproblem is related to one or more requirements. The solutions of the subproblems will be composed to solve the overall software problem. Subproblems are described using so called *problem diagrams*. We call our proposed patterns *problem-oriented security patterns* as we represent them by means of the problem frames approach, which focuses on understanding the problem.

The benefit of the proposed patterns is manifold. First, it provides guidance for refining security problem descriptions located in the problem space using security patterns located in the solution space. Hence, it enables less experienced software engineers to apply solution approaches early in the requirements engineering phase in a systematic manner. Second, it supports the transformation of the elaborated security requirement models into a particular solution at the design level. Thus, it bridges the gap between security problems and security solutions. Third, it supports the concurrent and iterative development of requirements and architectural descriptions systematically.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the problem frames approach. We introduce our problem-oriented security patterns in Section 3. Section 4 describes how to use the proposed patterns in our problem-oriented software development method. Section 5 presents related work, while Section 6 concludes the paper and points out suggestions for future work.

2. BACKGROUND

This section outlines the basic concepts of problem frames as a requirements engineering approach.

Problem frames are a means to understand, describe, and analyze software development problems. They were proposed by Michael Jackson [Jackson 2001], who describes them as follows:

"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."

A problem frame is described by a *frame diagram*, which basically consists of *domains*, *interfaces* between them, and a *requirement*. The task is to construct a *machine* (i.e., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements. A problem diagram represents an instance of a problem frame. In this paper, we only use the instances of problem frames, namely the problem diagrams.

We use a UML-based enhancement of problem frames, which is extended by a specific UML profile for problem frames (UML4PF) proposed by Hatebur and Heisel [Hatebur and Heisel 2010a]. The software to be developed

(*machine* in problem frames terminology) is represented by the stereotype ≪machine≫. Jackson distinguishes the following domain types:

—Biddable domains that are usually people. A biddable domain is represented by the stereotype ≪Biddable-Domain≫.

—Causal domains that comply with some physical laws. Causal domains are represented by the stereotype ≪CausalDomain≫.

—Lexical domains that are data representations. A lexical domain is represented by the stereotype ≪Lexical-Domain≫.

To establish a connection between two other domains by means of technical devices, a *connection domain* (≪ConnectionDomain≫) may be necessary. Examples are video cameras, sensors, or networks. This kind of modeling allows one to add further domain types, such as ≪DisplayDomain≫ (introduced in [Côté et al. 2008]), being a special case of a causal domain.

In this paper, we use a smart grid application to illustrate the concepts introduced here. 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 (ICT), which allows for financial, informational, and electrical transactions.

We derived the functional and security requirements from the documents such as "Application Case Study: Smart Grid" and "Smart Grid Concrete Scenario" provided by the industrial partners of the EU project NESSoS [1] and the "Protection Profile for the Gateway of a Smart Metering System" [Kreutzmann et al. 2011a; Kreutzmann et al. 2011b] provided by the German Federal Office for Information Security [2]. The protection profile states that "the Gateway is responsible for handling Meter Data. It receives the Meter Data from the Meter(s), processes it, stores it and submits it to external parties". Therefore, we define the requirements *RQ1-RQ3* to receive, process, and store meter data from smart meters. The requirement *RQ4* is concerned with submitting meter data to authorized external entities. Detailed description of the example smart grid is described in [Alebrahim et al. 2014].

Figure 1 illustrates the problem diagram in the context of our smart grid example. It describes the functional requirement *RQ3*, which states that *smart meter gateway shall store meter data from smart meters*. The submachine *StoreMeterData* receives meter data from the domain *TemporaryStorage* and stores it in the domain *MeterData*.

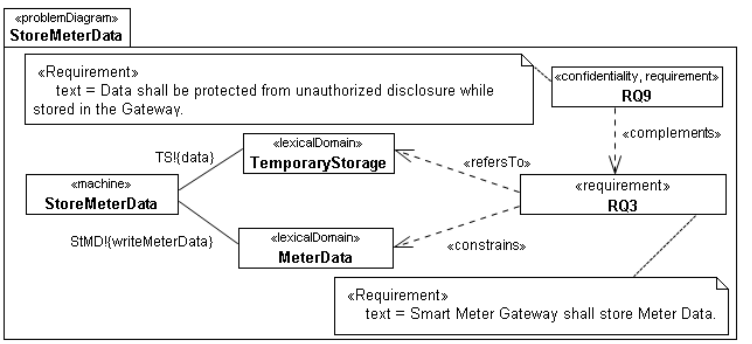

Fig. 1. A problem diagram expressed in UML

[1] http://www.nessos-project.eu/
[2] www.bsi.bund.de

In problem diagrams, *interfaces* connect domains and they contain *shared phenomena*. Shared phenomena may, e.g., be events, operation calls or messages. They are observable by at least two domains, but controlled by only one domain, as indicated by "!". The notation $StMD!\{writeMeterData\}$ (between *StoreMeterData* and *MeterData* in Fig. 1) means that the phenomenon *writeMeterData* is controlled by the machine domain *StoreMeterData*.

When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement expressed by the stereotype $\ll$`requirement`$\gg$ constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype $\ll$`constrains`$\gg$. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype $\ll$`refersTo`$\gg$. The requirement *RQ3* in Fig. 1 constrains the domain *MeterData*. It refers to the domain *TemporaryStorage*.

In order to annotate problem diagrams with quality requirements, we extended the UML profile for problem frames [Alebrahim et al. 2011b]. It enables us to complement functional requirements with quality requirements. A dependency from a quality requirement to a functional one is expressed with the stereotype $\ll$`complements`$\gg$. For annotating security requirements, we use the dependability profile [Hatebur and Heisel 2010b] that provides us with stereotypes such as $\ll$`confidentiality`$\gg$ and $\ll$`integrity`$\gg$. The confidentiality requirement *RQ9* complements the functional requirement *RQ3* in Fig. 1. The confidentiality requirement *RQ9* states that *data shall be protected from unauthorized disclosure while stored in the gateway*.

3. PROBLEM-ORIENTED SECURITY PATTERNS

In this section, we reuse existing security patterns and mechanisms and adapt them in a way that they can be used in the problem-oriented requirements engineering. We call the adapted patterns *problem-oriented security patterns*. The adaptation allows the requirements engineer to integrate such security-specific solutions early in the requirements engineering. Three problem-oriented security patterns are represented in this paper. Further problem-oriented security patterns may be extracted from the existing security patterns and mechanisms by the software engineers to aid the requirements engineers in integrating security solution approaches early in the requirements analysis.

The structure of the problem-oriented security patterns is described in Section 3.1. Sections 3.2, 3.3, and 3.4 describe the three patterns *problem-oriented encryption*, *problem-oriented RBAC*, and *problem-oriented digital signature* in detail.

3.1 Structure of the problem-oriented security patterns

A problem-oriented security pattern consists of a *graphical pattern* and a *template*. In the following, we describe the constituents of the pattern.

Graphical pattern:
The graphical pattern involves the following parts:

Functional Problem Diagram: During the requirements analysis phase, it is essential to describe and understand the problem explicitly. Hence, setting up a *functional problem diagram* is the first step to be performed for describing a specific problem. It captures the structure of the problem explicitly and consists of a generic functional requirement and the involved domains. To describe the security problem related to the functional requirement, we annotate this problem diagram with a specific security requirement, for which we provide a solution approach in the second part. The security requirement is annotated as complement to the functional requirement.

Security Problem Diagram: In the functional problem diagram, we described the functional problem and its related security requirement. The second part of a problem-oriented security pattern is a security problem diagram

that describes the particular solution approach for the security requirement annotated in the first part.

Composition Problem Diagram: The third part is concerned with composing the *functional problem diagram* and *security problem diagram* to obtain a solution for the overall problem. Hence, we provide a *composition problem diagram* that describes how the *functional problem diagram* and the *security problem diagram* can be composed to solve the overall problem. To this end, we make use of *Composition Frames* introduced in [Laney et al. 2004; Alebrahim et al. 2012] as a new kind of problem frames. Composition frames deal with the composition of two machines, each of which is described by a problem diagram in order to address combined requirements [Laney et al. 2004]. We use composition frames to integrate the problem diagram for the selected security solution with the functional problem diagram. A composition frame includes the domains shared between the functional problem and the security solution, and their corresponding machine domains. For the graphical patterns we use the same notation we use for the problem frames as described in Section 2.

As an optional part of the graphical pattern, a sequence diagram might be used to illustrate how the functional machine and the security machine interact with each other to solve the overall problem.

Template:
We provide a template consisting of two parts that documents additional information related to the domains in the problem diagrams. Such information is not observable in the graphical pattern. The first part accommodates information about the security mechanism itself such as name (*Name*), purpose (*Purpose*), description (*Brief Description*), and the quality requirement which will be achieved when applying this pattern (*Quality Requirement to be achieved*). Moreover, a security solution may affect the achievement of other quality requirements. For example, improving the security may result in decreasing the performance. Hence, the impact of each security solution on other quality requirements has to be captured in the first part of the template (*Affected Quality Requirement*). A security pattern not only solves a problem, but also produces new functional and quality problems that have to be addressed either as *Requirements* to be elicited or as *Assumptions* needed to be made in the second part of the template. We elicit new functional and quality problems as requirements if the software to be built shall achieve them. Assumptions have to be satisfied by the environment [Lamsweerde 2009b]. They do not guarantee 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.

We must note that the problem-oriented security patterns do not intend to provide rationales on how a particular security mechanism or pattern is selected. We assume this task has been done in the solution space prior to adopting the patterns and transforming them into the problem space as we make use of existing patterns and mechanisms. In this paper, we facilitate the use of proven security solutions from the design phase in a systematic manner by adopting them for the requirements analysis.

We present three problem-oriented security patterns, namely *symmetric encryption*, *Role-Based Access Control (RBAC)*, and *digital signature* as solutions for achieving confidentiality, integrity, and authenticity. Other security patterns and mechanisms can be transformed into the problem space analogously in order to be used in the problem-oriented software development.

3.2 Problem-oriented symmetric encryption pattern

Symmetric encryption is an important security mechanism to achieve confidentiality. There exists only one *secret key* which is used for encrypting and decrypting a plaintext that has to be kept confidentially. *Asymmetric encryption* is a similar means to achieve confidentiality. It however uses different keys for the encryption and decryption. One advantage of symmetric encryption is that it is faster than asymmetric encryption. The disadvantage is that both

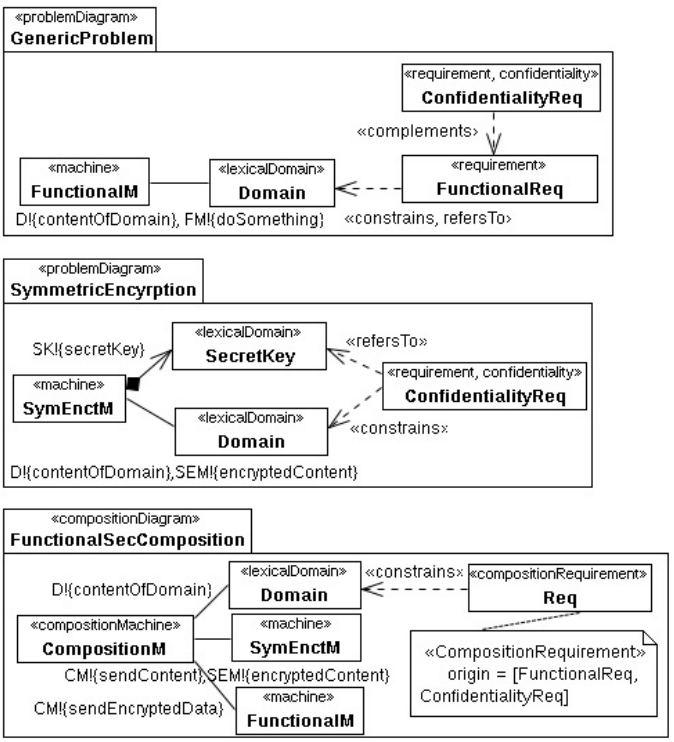

Fig. 2. Problem-oriented symmetric encryption pattern

communication parties must know the same key, which has to be distributed securely or negotiated. In asymmetric encryption, there is no key distribution problem, but a trusted third party is needed that issues the key pairs.

We present the *problem-oriented symmetric encryption pattern* by its corresponding graphical pattern depicted in Fig. 2 and its corresponding template shown in Table I.

Graphical pattern:

Functional Problem Diagram: The graphical pattern first describes the functional problem expressed as the problem diagram *GenericProblem*. It describes the functional requirement *FunctionalReq* and the involved domains. The functional requirement might be “sending the data", “storing data”, . . . to be achieved by the machine *FunctionalM*. Plaintext is expressed as the causal domain *Domain* in the problem diagram. It might be a causal domain such as a harddisc containing plaintext or a lexical domain which itself represents the plaintext. A lexical domain is defined as a special causal domain in UML4PF [Hatebur and Heisel 2010a]. The confidentiality requirement *ConfidentialityReq* is annotated in the problem diagram by complementing the functional requirement. It requires the achievement of the functional requirement in a confidential way. The functional problem diagram is depicted at the top of Fig. 2. Depending on the functional requirement, the problem diagram might contain other domains that are not relevant for the security problem. Hence, they are not represented in the pattern.

Security Problem Diagram: The symmetric encryption as a solution for the confidentiality problem is expressed by the problem diagram *SymmetricEncryption* shown in the middle of Fig. 2. It consists of all domains that are

relevant for the solution. The machine *SymEncM* should achieve the confidentiality requirement *ConfidentialityReq* by encrypting the *Domain* using the *SecretKey* which is part of the machine *SymEncM*.

Table I. Problem-oriented symmetric encryption pattern (template)

Security Solution	
Name	Symmetric Encryption
Purpose	For *Domain* constrained by the requirement *FunctionalReq*
Brief Description	The plaintext is encrypted using the secret key
Quality Requirement to be achieved	Security (confidentiality) *ConfidentialityReq*
Affected Quality Requirement	Performance *PerformanceReq*, availability *AvailabilityReq*
Necessary Conditions	
Requirement ☐ **Assumption** ☐	Secret key shall be/is distributed.
Requirement ☐ **Assumption** ☐	Confidentiality of secret key distribution shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of secret key distribution shall be/is preserved.
Requirement ☐ **Assumption** ☐	Confidentiality of secret key during transmission shall be/is preserved
Requirement ☐ **Assumption** ☐	Confidentiality of secret key during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of secret key during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Confidentiality of encryption machine shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of encryption machine shall be/is preserved.

Composition Problem Diagram: The third part of the graphical pattern shown at the bottom of Fig. 2 is concerned with combining the *functional problem diagram* with the *security problem diagram* to obtain the composed problem diagram *FunctionalSecComposition*. It consists of the new machine *CompositionM*, both machine domains *FunctionalM* and *SymEncM*, and the domains shared by both problem diagrams. In our case, there is only one domain *Domain*. Consider that the lexical domain *SecretKey* is part of the machin *SymEncM* that we made visible as it is of great importance for the encryption mechanism. The machine *CompositionM* is responsible for coordinating the functional machine *FunctionalM* and the solution machine *SymEncM*. The requirement *CompositionReq* shall be achieved by the machine *CompositionM*.

We will see in the following by the description of the related template that we need to capture new assumptions and elicit new requirements regarding the secret key.

Template:
The template shown in Table I represents the additional information corresponding to the graphical part of the problem-oriented pattern *symmetric encryption*. After capturing the basic information 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*. Note that the requirements and assumptions are not fixed. Requirements have to be met by the machine (i.e. software-to-be) and assumptions by the environment. 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 assumption.

Note that the problem-oriented security pattern *symmetric encryption* has to be applied by the sender for encrypting the plaintext. There exists a counterpart to this pattern, namely the problem oriented security pattern

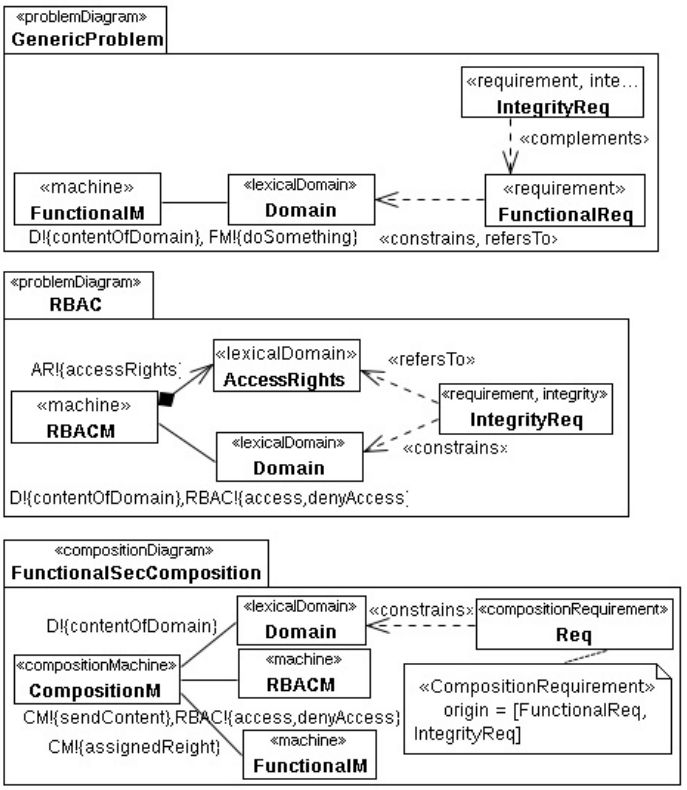

Fig. 3. Problem-oriented RBAC pattern (graphical pattern)

symmetric decryption on the receiver side which is responsible for decrypting the encrypted plaintext. The counterpart pattern is structured in the same way. We do not introduce the counterpart pattern in this paper as it is very similar to the problem-oriented security pattern *symmetric encryption*.

3.3 Problem-oriented RBAC pattern

Since verifying permission is a frequently recurring problem in security relevant systems, it has been treated in several access control patterns for the design phase [Yskout et al. 2006; Schumacher et al. 2005]. Access control patterns define security constraints regarding access to resources. Role-Based Access Control (RBAC) provides access to resources based on functions of people in an environment, known as *roles*, and the kind of permission they have, known as *rights*.

We present the *problem-oriented RBAC pattern* by its corresponding graphical pattern depicted in Fig. 3 and its corresponding template shown in Table II.

Graphical pattern:

Functional Problem Diagram: The first part of the graphical pattern, namely the *functional problem diagram* expressed by the problem diagram *GenericProblem*, is similar to the previous one from the structure point of view. The functional requirement *FunctionalReq* might be “editing data” to be met by the functional machine *FunctionalM*. The functional requirement is complemented by the security requirement *IntegrityReq* demanding

Table II. Problem-oriented RBAC pattern (template)

Security Solution	
Name	RBAC
Purpose	For *Domain* constrained by the requirement *FunctionalReq*
Brief Description	It provides access to data based on defined roles and rights captured as access rights.
Quality Requirement to be achieved	Security (confidentiality and integrity during storage) *ConfidentialityReq*, *IntegrityReq*
Affected Quality Requirement	Performance *PerformanceReq*, availability *AvailabilityReq*
Necessary Conditions	
Requirement ☐ **Assumption** ☐	Integrity of access rights shall be/is preserved.
Requirement ☐ **Assumption** ☐	Confidentiality of data during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of data during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Confidentiality of RBAC machine shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of RBAC machine shall be/is preserved.

"the protection of data against unauthorized access". The functional problem diagram is depicted at the top of Fig. 3.

Security Problem Diagram: The second part provides the domains that are required for applying the *RBAC* pattern expressed by the problem diagram *RBAC* shown in the middle of Fig. 3. The lexical domain *AccessRights* represents *user id*, assigned *role(s)* to it, and assigned *right(s)* to the role(s). It is a part of the machine *RBACM* which is responsible for achieving the integrity requirement.

Composition Problem Diagram: The third part composes the functional machine *FunctionalM* with the security machine *RBACM* by introducing a new machine *CompositionM* that has to meet the requirement *CompositionReq* composed of the requirements *FuncitonalReq* and *IntegrityReq*. It is depicted at the bottom of Fig. 3.

Note that the *problem-oriented RBAC pattern* can be used to achieve a confidentiality requirement as well. In Fig. 3, we only showed the use of the *problem-oriented RBAC pattern* to achieve the integrity requirement *IntegrityReq* in order to keep the figure clear and readable. One can apply the same pattern and only replace the integrity requirement with the confidentiality requirement to achieve confidentiality.

Template:
The template shown in Table II represents the additional information corresponding to the graphical part of the problem-oriented pattern *RBAC*. In addition to the basic information regarding the pattern itself, it contains requirements and assumptions to be selected by the requirements engineer.

3.4 Problem-oriented digital signature pattern

Digital signature is an important means for achieving integrity and authenticity of data. Using the digital signature, the receiver ensures that the data is created by the known sender.

We present the *problem-oriented digital signature pattern* by its corresponding graphical pattern depicted in Fig. 4 and its corresponding template shown in Table III.

Graphical pattern:
The structure of the graphical pattern is similar to the previous patterns. As we described the structure of previous patterns extensively, we do not describe the graphical pattern any more and only refer to Fig. 4.

Note that the *problem-oriented digital signature pattern* can be used to achieve an integrity requirement as well. In Fig. 4, we only showed the use of the *problem-oriented digital signature pattern* to achieve the integrity requirement *AuthenticityReq* in order to keep the figure clear and readable. One can apply the same pattern and

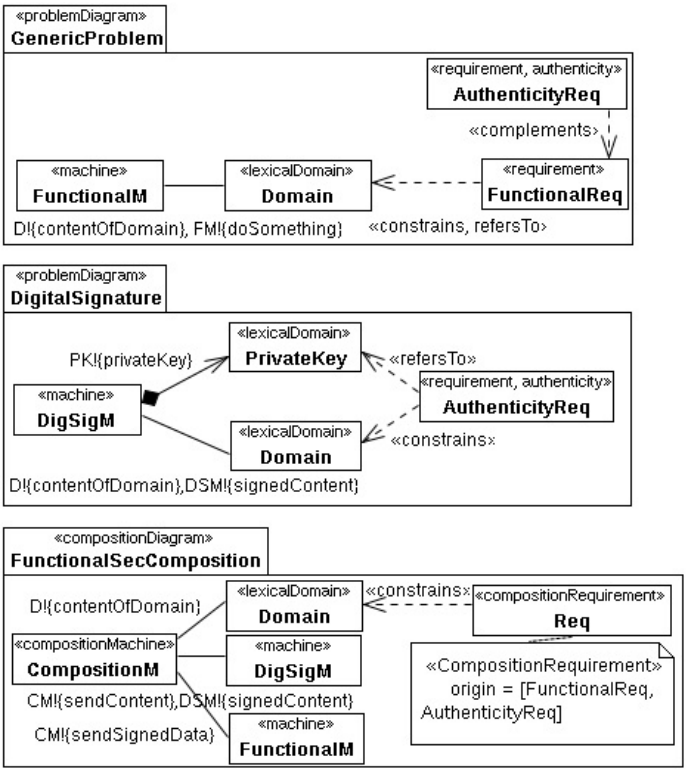

Fig. 4. Problem-oriented digital signature pattern (graphical pattern)

Table III. Problem-oriented digital signature pattern (template)

Security Solution	
Name	Digital Signature
Purpose	For *Domain* constrained by the requirement *FunctionalReq*
Brief Description	Sender produces a signature using the private key and the data.
Quality Requirement to be achieved	Security (integrity and authenticity) *IntegrityReq*, *AuthenticityReq*
Affected Quality Requirement	Performance *PerformanceReq*
Necessary Conditions	
Requirement ☐ **Assumption** ☐	Confidentiality of private key during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of private key during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Confidentiality of signature machine shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of signature machine shall be/is preserved.

only replace the authenticity requirement with the integrity requirement to achieve integrity.

Template:

The template shown in Table III represents the additional information corresponding to the graphical part of the problem-oriented pattern *digital signature*. New requirements and assumptions to be considered are represented in the second part of the template.

Note that we do not provide a structured method to identify new requirements and assumptions as necessary conditions. However, as mentioned earlier in this section, new requirements and assumptions arise due to introducing the security solution (*Security Problem Diagram*). Hence, we can reduce the scope of consideration for identifying new requirements and assumptions to the solution for the security requirement, namely to the *Security Problem Diagram*. In this problem diagram, we only need to consider the lexical domain for example the *SecretKey* for the *problem-oriented symmetric encryption pattern* and its related security machine *SymEncM*. For these two domains, we have to think about new problems that might arise and then capture them as new requirements and/or assumptions.

4. RELATION TO PROBLEM-ORIENTED SOFTWARE DEVELOPMENT

In this section, we describe how the proposed patterns can be applied within our problem-oriented software development method [Alebrahim et al. 2011b; Alebrahim et al. 2011a; Alebrahim et al. 2014]. As we mentioned earlier, problem descriptions and architectural descriptions should be considered as intertwining artifacts influencing each other. We therefore take the Twin Peaks model [Nuseibeh 2001] into account and illustrate our problem-oriented software development method embedded in the context of the Twin Peaks model (see Fig. 5). Our problem-oriented software development method consists of five phases.

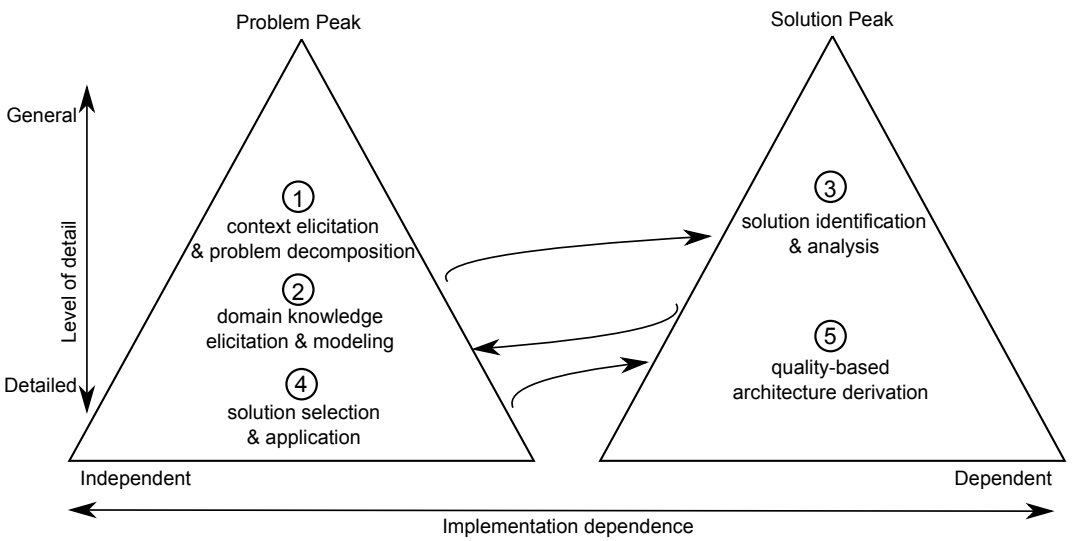

Fig. 5. Overview of our method embedded in the Twin Peaks Model

Phase 1- context elicitation & problem decomposition. This phase (see 1 in Fig. 5) involves understanding the problem and its context, decomposing the overall problem into subproblems, and annotating security requirements. In order to understand the problem, we elicit all domains related to the problem to be solved, their relations to each other and the software to be constructed. Doing this, we obtain a *context diagram* consisting of the machine (software-to-be), related domains in the environment, and interfaces between these domains. Then, we decompose the overall problem into subproblems, which describe a certain functionality, as expressed by a set of related functional requirements. We set up *problem diagrams* representing subproblems to model functional requirements. A problem diagram consists of one submachine of the machine given in the context diagram, the relevant domains, the interfaces between these domains, and a requirement referring to and constraining problem domains.

To analyze and integrate quality requirements in the software development process, quality requirements have to be modeled and integrated as early as possible in the requirement models. Modeling quality requirements and associating them to the functional requirements is achieved by annotating them in problem diagrams. Figure 1 illustrates the problem diagram for describing the functional requirement *RQ3* annotated by the security requirement *RQ9*. For more information about context elicitation and problem decomposition, see our previous work [Alebrahim et al. 2011b].

Phase 2- domain knowledge elicitation & modeling. The system-to-be comprises the software to be built and its surrounding environment such as people, devices, and existing software [Lamsweerde 2009b]. In requirements engineering, properties of the environment and constraints about it, called *domain knowledge*, need to be captured in addition to exploring the requirements [Jackson 2001; Lamsweerde 2009a]. Hence, the second phase (see 2 in Fig. 5) involves *domain knowledge elicitation & modeling*. For more information, see our previous work [Alebrahim et al. 2014].

Phase 3- solution identification & analysis. The first two phases are concerned with the activities accommodated in the requirements engineering (known as *problem peak* in the Twin Peaks model). Finding security strategies to achieve security requirements is the aim of the third phase (see 3 in Fig. 5). To this end, we explore the solution space by identifying those security patterns and mechanisms such as *encryption* and *RBAC* that support the achievement of security requirements which have already been annotated in problem diagrams (for example in Fig. 1).

Phase 4- solution selection & application. Once we have identified the most appropriate security pattern for our particular security problem, we map it to one of our proposed problem-oriented security patterns in the fourth phase (see 4 in Fig. 5). Then we apply the selected problem-oriented security pattern to refine the requirement models by integrating security mechanisms.

We assume that we have identified *symmetric encryption* as an appropriate mechanism to fulfill confidentiality in phase three. In phase four, we map the problem diagram shown in Fig. 1 to the first part of the graphical pattern described in Section 3.2, namely the *functional problem diagram*. Figure 1 represents an instance of the *functional problem diagram*, in which the machine *StoreMeterData* represents the machine *FunctionalM*, the domain *TemporaryStorage* represents the domain *Domain*, the functional requirement *RQ3* represents the functional requirement *FunctionalReq*, and the confidentiality requirement *RQ9* represents the confidentiality requirement *ConfidentialityReq*. As mentioned in Section 3.2, depending on the functional requirement, the problem diagram might contain other domains that are not relevant for the security problem at hand. Hence, they are not represented in the pattern. The lexical domain *MeterData* in Figure 1 is such a case.

We instantiate the second part of the pattern, namely the *security problem diagram*. Doing this, we obtain the upper problem diagram in Fig. 6, in which the domain *TemporaryStorage* represents the domain *Domain*, the domain *Key* represents the domain *SecretKey*, the confidentiality requirement *RQ9* represents the confidentiality requirement *ConfidentialityReq*, and the machine *SymEnc* represents the machine *SymEncM* in the pattern. In this problem diagram, the machine *SymEnc* receives the key (*keyContent*) from the domain *Key* and the data (*tempData*) from the domain *TemporaryStorage* and encrypts it by producing the *encryptedData*.

We instantiate the third part of the pattern, namely the *composition problem diagram*, by mapping the domain *Domain* to the domain *TemporaryStorage*, the machine *SymEncM* to the machine *SymEnc*, the machine *FunctionalM* to the machine *StoreMeterData*, the composition requirement *Req* to the composition requirement *RQ3+RQ9*, and the composition machine *CompositionM* to the composition machine *SecurityManager*. Doing this, we obtain the lower problem diagram in Fig. 6, in which the *functional problem diagram* represented in Fig. 1 and the *security problem diagram* in Fig. 6 are combined. The machine *SecurityManager* receives the *tempData* from the domain *TemporaryStorage*, sends it to the machine *SymEnc*. The machine *SymEnc* encrypts the data and

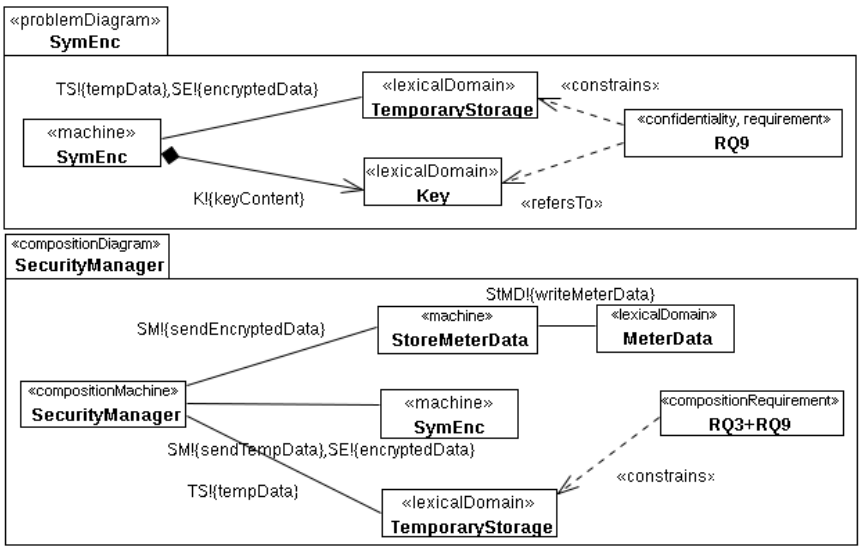

Fig. 6. Instance of the second and third parts of the *problem-oriented symmetric encryption pattern*

sends the *encryptedData* back to the machine *SecurityManager*. This machine sends the encrypted data to the machine *StoreMeterData*, which stores the data persistently in the domain *MeterData*.

In this way, we successfully instantiated and applied the problem-oriented symmetric encryption pattern in phase four of our method in order to fulfill the confidentiality requirement *RQ9* for the functional requirement *RQ3*. We only showed the instantiation of the graphical part of the pattern. The corresponding template can be easily filled out using the information obtained from the solution space and according to the responsibility for the achievement of necessary conditions. As mentioned before, the templates as part of the problem-oriented security patterns describe the effect of each pattern on the requirement models. According to this part of the templates, we have to update the requirement models including domain knowledge (first and second phases of the method) by selecting and applying a problem-oriented security pattern. Therefore, our method has to be conducted as a recursive one to obtain correct requirements and a sound architecture.

Phase 5- quality-based architecture derivation. In this phase (see 5 in Fig. 5), we derive an architecture description from problem descriptions by mapping domains from problem diagrams to components in the software architecture. For more information, see our previous work [Alebrahim et al. 2011a].

Benefits. Problem-oriented security patterns allow software engineers not only to think about security problems as early as possible in the software development life cycle, but also to think about solution approaches solving such security problems. Furthermore, the corresponding templates represent consequences of applying such solution approaches by providing new assumptions and/or requirements to be considered when deciding on a specific pattern.

By exploring the solution space, we find appropriate solution mechanisms, which can be used for refining security requirement models in the requirement engineering phase. However, problem-oriented security patterns do not really apply a solution. They only enforce the requirements analyst to think about the problem, its solution and the consequences of applying a specific solution as early as possible. This results in preparing the requirement models for applying the “classical” security patterns later on in the design phase. For example, to think about new lexical domains and machines to be introduced, new assumptions to be captured, and new requirements to be elicited.

Thus, they do not replace the application of the “classical” security patterns. Problem-oriented security patterns are located in the problem space aiming at structuring and elaborating security problems while “classical” security patterns are accommodated in the solution space aiming at solving security problems. Hence, problem-oriented security patterns in the requirements engineering phase represent the counterpart to the “classical” security patterns in the design phase.

In a model-driven software development, the elaborated security requirement models can easily be transformed into a particular security pattern at the design level. As the requirement models and the architecture models can be kept in one single model, it is possible to perform integrity checks in order to verify whether there exist security solutions on the architecture level that fulfill the security requirements annotated in the requirement models. In addition, traceability links between requirement and architecture models can be easily provided in such a model-based approach. Hence, problem-oriented security patterns support bridging the gap between security problems and security solutions.

5. RELATED WORK

There has been a number of research works that proposed patterns for different areas of software engineering and other application domains. We mainly discuss here those approaches related to the problem frames approach.

Five basic problem frames are defined in the problem frames approach proposed by Jackson [Jackson 2001], namely the *required behaviour*, *commanded behaviour*, *information display*, *simple workpieces*, and *transformation* to which all problems can be mapped. The problem frames approach provides no support for addressing quality requirements. In our previous work [Alebrahim et al. 2011b], we proposed an extension of it by introducing new elements that enable us annotating quality requirements. In this paper, we make use of the extension to address quality requirements in the *functional problem diagram* and propose a security solution for the annotated quality requirements. Hence, the problem frames provide a framework which allows for expressing problem-oriented security patterns as problem diagrams.

Beckers et al. [Beckers et al. 2013] propose a meta model for describing context patterns for various kinds of domain knowledge. The meta model is based on a number of context patterns for different areas of domain knowledge such as Peer-to-Peer, cloud computing, and the legal domain developed in the past by the authors. To improve the understanding of the context and eliciting required information, these patterns can be integrated into existing software development methods. The domain knowledge required for the software engineering can be captured by instantiating such context patterns. Similar to our patterns, the context patterns are provided for the requirements engineering phase. However, they differ from our problem-oriented security patterns as they elicit and analyze the context supporting the elicitation of requirements while we focus on finding solution approaches for already elicited security requirements.

Hatebur et al. [Hatebur et al. 2006] propose security problem frames and concretized security problem frames. Security problem frames represent special kinds of problem frames which address the security problems. They do not take into account a solution. They have to be transformed into concretized security problem frames which address a solution using generic security mechanisms. Similar to this approach, we make use of the problem frames approach as a basis for providing security patterns. The first part of our graphical pattern, namely the generic problem diagram corresponds to the security problem frames proposed in [Hatebur et al. 2006]. The second part of our graphical pattern represents a generic security mechanism, while the third part corresponds to the concretized security problem frames. In addition, we provide a template for each pattern including meta information and necessary conditions that need to be considered when applying a specific pattern.

The same authors present a pattern system in a further work [Hatebur et al. 2007] based on security problem frames and their counterparts concretized security problem frames. In this work, the relationships and dependencies between different frames are represented explicitly in the pattern system. The pattern system should support the security engineer by choosing the appropriate concretized security problem frame for an identified security problem frame.

A pattern language for security risk analysis of web applications is proposed by Li et al. [Li et al. 2013], which can be used to support the conduction of risk analysis in the early phase of the software development life cycle. The authors introduce three basic pattern types, namely security requirement patterns, web application architecture design patterns, and risk analysis model patterns that can be combined to build security risk analysis composite patterns. A security requirement pattern is defined according to the problem frames notation. The output of the security requirement pattern, namely a security requirement, represents the input for the web application architecture design pattern representing the architecture design for a specific web application. Our problem-oriented security patterns can be used as an intermediate step between the security requirement patterns and web application architecture design patterns to facilitate the transformation of security requirement patterns in the requirement analysis phase into the web application architecture design patterns in the design phase. In addition, our patterns help identifying new assumptions / requirements that need to be considered when applying a specific security pattern.

Choppy et al. [Choppy et al. 2005] propose software architectural patterns that correspond to the different problem frames to be used in the design phase. To create an architectural solution for a concrete problem frame, the relating architectural pattern must be instantiated which provides the starting point for building a software architecture. Our security patterns focus on the analysis phase of software engineering, whereas the architectural patterns as solutions to the problem frames are defined for the design phase of software engineering.

Architectural Frames (AFrames) [Rapanotti et al. 2004] represent combinations of architectural styles and Problem Frames. They are introduced to make use of the knowledge of existing software architectural patterns such as Pipe-and-Filter and Model-View-Controller (MVC) in the problem frames approach to decompose the problem frames. The corresponding AFrames are applied to the classes of transformation and control problems. Similar to our problem-oriented security patterns, AFrames use the knowledge gained in the solution space in the problem space. However, they are used for decomposing the problem frames as functional problems, whereas our proposed patterns are concerned with security problems.

6. CONCLUSION

We have provided a systematic structure for problem-oriented security patterns consisting of a three-part graphical pattern and a template describing the affect of the security pattern on requirements when applied. We have presented three examples of problem-oriented security patterns to address confidentiality, integrity, and authenticity problems on the requirement level. Other security patterns and mechanisms located in the solution space can easily be transformed into the problem space analogously using the proposed structure for problem-oriented security patterns. Necessary for such transformation is the knowledge about the patterns and/or mechanisms, their structure, and their constituents which we gain in phase 3 of our method (*solution identification & analysis*) to build problem-oriented security patterns.

The proposed patterns facilitate the use of proven security solutions from the design phase in a systematic manner by adopting them for the requirements analysis. The instantiations of the patterns can be used as a basis for a seamless transition from requirement analysis to architectural design. The proposed patterns can be integrated into software development processes based on problem frames to refine the requirement models and bridge the gap between security problems and security solutions.

Note that the problem-oriented security patterns are not intended to only update the requirement models *after* applying the “classical” security patterns in the design phase. Rather they have to be applied first in phase 4 (*solution selection & application*) to adapt the requirement models by adding new domains, new assumptions, facts, and/or requirements. We then apply the security patterns to design the architecture by making more fine-grained decisions such as selecting the algorithms, secret key (key length, . . .) in phase 5 (*quality-based architecture derivation*).

In the future, we will extend the basis of our existing problem-oriented security patterns with more patterns in order to provide a catalog of problem-oriented security patterns for requirements analysis addressing more

kinds of security issues. In addition, we strive for improving our current problem-oriented software development process by introducing a new step for the use of problem-oriented security patterns in a systematic manner. Furthermore, we plan to provide *problem-oriented performance patterns* which are structured in a similar way as the proposed problem-oriented security patterns. They will be integrated into our problem-oriented software development process to derive software architectures that not only meet the security requirements but also the performance requirements.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd Christopher Preschern for his valuable feedback to improve this paper.

REFERENCES

ALEBRAHIM, A., CHOPPY, C., FASSBENDER, S., AND HEISEL, M. 2014. Optimizing functional and quality requirements according to stakeholders' goals. In *System Quality and Software Architecture (SQSA)*. Elsevier, 75–120.

ALEBRAHIM, A., HATEBUR, D., AND HEISEL, M. 2011a. A method to derive software architectures from quality requirements. In *Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC)*, T. D. Thu and K. Leung, Eds. IEEE Computer Society, 322–330.

ALEBRAHIM, A., HATEBUR, D., AND HEISEL, M. 2011b. Towards systematic integration of quality requirements into software architecture. In *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, I. Crnkovic, V. Gruhn, and M. Book, Eds. LNCS 6903. Springer Verlag, 17–25.

ALEBRAHIM, A., HEISEL, M., AND MEIS, R. 2014. 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)*. LNCS 8583. Springer, 370–386.

ALEBRAHIM, A., TUN, T. T., YU, Y., HEISEL, M., , AND NUSEIBEH, B. 2012. An aspect-oriented approach to relating security requirements and access control. In *Proceedings of the CAiSE Forum*. CEUR Workshop Proceedings Series, vol. 855. CEUR-WS.org, 15–22.

BECKERS, K., FASSBENDER, S., AND HEISEL, M. 2013. A meta-model approach to the fundamentals for a pattern language for context elicitation. In *Proceedings of the 18th European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, –. Accepted for Publication.

CHOPPY, C., HATEBUR, D., AND HEISEL, M. 2005. Architectural patterns for problem frames. *IEE Proceedings – Software, Special issue on Relating Software Requirements and Architecture 152,* 4, 198–208.

CHUNG, L., NIXON, B. A., YU, E., AND MYLOPOULOS, J. 2000. *Non-functional requirements in software engineering*. Klewer Academic.

CÔTÉ, I., HATEBUR, D., HEISEL, M., SCHMIDT, H., AND WENTZLAFF, I. 2008. A Systematic Account of Problem Frames. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*. Universitätsverlag Konstanz, 749–767.

FOWLER, M. 1996. *Analysis Patterns: Reusable Object Models*. Addison Wesley.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

HATEBUR, D. AND HEISEL, M. 2010a. Making Pattern- and Model-Based Software Development More Rigorous. In *Proceedings of 12th International Conference on Formal Engineering Methods (ICFEM)*, J. S. Dong and H. Zhu, Eds. LNCS 6447. Springer Verlag, 253–269.

HATEBUR, D. AND HEISEL, M. 2010b. A UML profile for requirements analysis of dependable software. In *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, E. Schoitsch, Ed. LNCS 6351. Springer Verlag, 317–331.

HATEBUR, D., HEISEL, M., AND SCHMIDT, H. 2006. Security engineering using problem frames. In *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*. Springer Verlag, 238–253.

HATEBUR, D., HEISEL, M., AND SCHMIDT, H. 2007. A pattern system for security requirements engineering. In *Proceedings of the 7th International Conference on Availability, Reliability and Security (AReS)*. IEEE Computer Society, Los Alamitos, CA, USA, 356–365.

JACKSON, M. 2001. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley.

KREUTZMANN, H., VOLLMER, S., TEKAMPE, N., AND ABROMEIT, A. 2011a. Protection profile for the gateway of a smart metering system. Tech. rep., BSI.

KREUTZMANN, H., VOLLMER, S., TEKAMPE, N., AND ABROMEIT, A. 2011b. Protection profile for the security module of a smart metering system. Tech. rep., BSI.

LAMSWEERDE, A. 2009a. Reasoning about alternative requirements options. In *Conceptual Modeling: Foundations and Applications*, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Vol. LNCS 5600. Springer, 380–397.

LAMSWEERDE, A. 2009b. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

LANEY, R., BARROCA, L., JACKSON, M., AND NUSEIBEH, B. 2004. Composing requirements using problem frames. In *Proceedings of the 4th IEEE International Requirements Engineering Conference (RE)*. Press, 122–131.

LI, Y., KOBRO RUNDE, R., AND STØLEN, K. 2013. A meta-model approach to the fundamentals for a pattern language for context elicitation. In *Proceedings of the 20th Conference on Pattern Languages of Programs (PLOP)*.

NUSEIBEH, B. 2001. Weaving together requirements and architectures. *IEEE Computer 34,* 3, 115–117.

RAPANOTTI, L., HALL, J. G., JACKSON, M., AND NUSEIBEH, B. 2004. Architecture-driven problem decomposition. In *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE)*. 80–89.

SCHUMACHER, M., FERNANDEZ-BUGLIONI, E., HYBERTSON, D., BUSCHMANN, F., AND SOMMERLAD, P. 2005. *Security patterns: integrating security and systems engineering*. John Wiley & Sons.

SHAW, M. AND GARLAN, G. 1996. *Software Aechitecture: Perspectives on an emerging discipline*. Prentice Hall.

WHALEN, M., GACEK, A., COFER, D., MURUGESAN, A., HEIMDAHL, M., AND RAYADURGAM, S. 2013. Your "What" Is My "How": Iteration and Hierarchy in System Design. *IEEE Software 30,* 2, 54–60.

YSKOUT, K., HEYMAN, T., SCANDARIATO, R., AND JOOSEN, W. 2006. A system of security patterns. Report CW 469, K.U.Leuven, Department of Computer Science.