A Meta-Model for Context-Patterns

KRISTIAN BECKERS and STEPHAN FASSBENDER and MARITA HEISEL,
paluno, The Ruhr Institute for Software Technology

It is essential for building the right software system to elicit and analyze requirements. Writing requirements that can achieve the purpose of building the right system is only possible if the domain knowledge of the system-to-be and its environment is known and considered thoroughly. We consider this as the context problem of software development. In the past, we tackled this problem by describing common structures and stakeholders for several different domains. The common elements of the context were obtained from observations about the domain in terms of standards, domain specific publications and implementations. But the description of the structure of a context-pattern, especially in terms of its static structure, was not aligned. This inhibits relating context-patterns to form a pattern language. It is also difficult for inexperienced pattern creators to describe newly observed patterns without any guidance.

We propose a meta model for describing context-patterns. The meta model contains elements, which can be used to structure and describe domain knowledge in a generic form. These context-patterns can afterwards be instantiated with the domain knowledge required for software engineering. This work is based on already existing patterns, which we abstracted into a meta-model. We present our context-patterns, show how we used them to construct our meta-model, and provide an example of how to describe a context-pattern using our meta-model. We contribute this meta model as a basis for a pattern language for context elicitation.

Categories and Subject Descriptors: Software and its engineering [**Software organization and properties**]: Software system structures—*Software system models - Model-driven software engineering*; Software and its engineering [**Software notations and tools**]: Context specific languages—*Domain specific languages*; Software and its engineering [**Software creation and management**]: Designing software—*Requirements analysis*

General Terms: Human Factors

Additional Key Words and Phrases: Domain Knowledge, Requirements Engineering, Context Establishment

ACM Reference Format:

Beckers, K., Faßbender, S. and Heisel, M. 2015. A Meta-Model for Context-Patterns. EuroPLoP '13: Proceedings of the 18th European Conference on Pattern Languages of Program, Article 5 (July 2013) , 15 pages. ACM.

1. INTRODUCTION

The long known credo of requirements engineering states that it is challenging to build the right system if you do not know what right is. Requirements engineering methods have to consider domain knowledge, otherwise severe problems can occur during software development, e.g., technical solutions to requirements might be impractical or costly. It is an open research question of *how* to elicit domain knowledge correctly for effective requirements elicitation [Niknafs and Berry 2012]. Several requirements engineering methods exist. Fabian et. al [Fabian et al. 2010] concluded in their survey about these methods that it is not yet state of the art to consider domain knowledge.

We propose to build patterns for a structured domain knowledge elicitation. Depending on the kind of domain knowledge that we have to elicit for a software engineering process, we always have certain elements that require

Author's address: Kristian Beckers, Oststrasse 99, 47057 Duisburg, Germany; email kristian.beckers@uni-duisburg-essen.de; Stephan Faßbender, Oststrasse 99, 47057 Duisburg, Germany; email: stephan.fassbender@uni-due.de; Maritta Heisel, Oststrasse 99, 47057 Duisburg, Germany; email: stephan.fassbender@uni-due.de
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
EuroPLoP '13, July 10 - 14, 2013, Irsee, Germany
Copyright 2015 ACM. ISBN 978-1-4503-3465-5/15/07...$15.00
DOI: http://dx.doi.org/10.1145/2739011.2739016

consideration. We base our approach on Jackson's work [Jackson 2001] that considers requirements engineering from the point of view of a machine in its environment. The machine is the software to be built and requirements are the effect the machine is supposed to have on the environment. Our patterns do not enforce considering the machine explicitly, but demand a description of the environment.

In this paper, we develop and present a meta-model for building patterns for considering domain knowledge during requirements engineering. We consider different kinds of domain knowledge, e.g., technical or organizational domain knowledge. Therefore, we use a bottom up approach, starting with a set of previously and independently developed context-patterns.

The ultimate aim of this paper is to identify the common concepts, which are part of the already obtained context-pattern, and aggregate them to a meta-model of elements one has to talk and think about when describing a new context-pattern.This is quite similar to what Jackson [Jackson 2001] proposed for requirements. He defined a meta-model of reoccurring domains, like causal, biddable and lexical domains. These domains are used to define basic requirements patterns, so-called *Problem Frames*. In this work we show a similar meta-model for context elicitation. We show *how* we derived it from already existing context elicitation patterns, and how it can be used to describe the structural part of a new context-pattern.

This meta-model has several benefits. First, it forms a uniform basis for our context-patterns, making them comparable. Second, findings and results for one pattern can be transferred to the other pattern via a generalization of meta-model elements. Third, the meta-model contains the important conceptual elements for context elicitation patterns. Fourth, it enables us to form a pattern language for the context elicitation pattern.

Note that there are different views on the term *pattern language* and a *meta-model* is an important part of it. According to Fowler [Fowler 1996; 2002] a pattern language indeed is about the relations between patterns. Hafiz et al. [Hafiz et al. 2012] agree and elaborate that an enumeration of patterns is just a pattern catalog. Both, Hafiz et al. and Fowler, basically adopt the view of Alexander [Alexander 1978] towards patterns and pattern languages. In contrast, Jackson [Jackson 2001] calls his domain-types and interfaces in between already a language for expressing problem frames (which are kind of patterns [Jackson 2001]). The frames themselves are not related by Jackson. To be precise, Jackson never uses the term "pattern language", but a language for expressing problem frames.

Those different views can be aligned in some way. Fowler states that a defined "form" [Fowler 1996] or "pattern structure" [Fowler 2002] is essential for expressing a pattern. He refers further to the work of Alexander. For Fowler, a pattern structure should consist of a "sketch" [Fowler 2002], which is a structural and graphical description, and a textual description of behaviour and relations. A very similar understanding of how to describe a pattern can be found in Hafiz et al. [Hafiz et al. 2012] and Fernandez et al. [Fernandez and Pan 2001]. This understanding is in line with what Jackson calls a language to describe a problem frame (pattern) [Jackson 2001]. Hafiz refines the structural and graphical description further to a common vocabulary and syntax [Hafiz et al. 2012]. The textual description additionally needs a grammar and behavioural elements. Hence, our meta-model is not a full pattern language. It "only" defines the vocabulary and syntax for describing patterns. But to form a pattern language it is necessary to have a common way to describe patterns [Fowler 1996; 2002]. Hence, we did the first step towards a pattern language and will complete the missing steps in future work.

Using this meta-model we empower requirements engineers to describe their own context-patterns, which capture the most important parts for understanding the context of a system-to-be. Note, that the difference between our meta model and Tolendano's [Toledano 2002] meta-pattern is that he abstracted existing patterns into meta-patterns, while we want to create a meta model as a basis for a pattern language for context elicitation.

In addition, patterns produced with our meta-model are scalable, because it is possible to attach diagrams to the patterns, e.g., UML activity diagrams that explain interactions between different stakeholders. These diagrams can support different views and levels of granularity. All these diagrams are based upon our patterns. Hence, we can apply consistency checks and traceability links with the elements in the attached diagrams and the created patterns.

The domain knowledge elicited using our patterns is not limited to software engineering. We showed in previous work, e.g., [Beckers et al. 2013], that the knowledge can also support the establishment of security standards of the ISO 27000 family of standards [ISO/IEC 2009].

The rest of the paper is organized as follows. In the next section we show several previously and independently described patterns which were the input for our bottom up method to build the meta-model. We show the meta-model and its development based on these pattern in Sect. 4. We explain how to use the meta-model in Sect. 5. Section 6 concludes the paper.

2. DEFINING CONTEXT-PATTERN

We address the context elicitation and consideration as discussed previously in Section 1 by describing common structures and stakeholders for several different domains in so-called *context-patterns* for structured domain knowledge elicitation. Depending on the kind of domain knowledge that we have to elicit for a software engineering process, we always have certain elements that require consideration. We base our approach on Jackson's work [Jackson 2001] that considers requirements engineering from the point of view of a machine in its environment.

A context-pattern consists of the following parts:

Method. A method contains a sequence of steps. Each step is described using well defined activities, inputs, and outputs. Inputs are descriptions of the required artifacts to perform the activities of the method step. Activities are descriptions of the processing of all inputs into outputs. Outputs are the desired results of the activities of this step. A context-pattern has to contain a method that describes how to use the context-pattern.

Graphical pattern. Our context-patterns do not enforce considering the machine, meaning the system we are going to build explicitly, but demand a description of its environment in graphical form. This environment contains domain knowledge. In particular, any given environment considers certain elements, e.g., stakeholders or technical elements. Moreover, we believe that every environment of a software engineering problem can be divided into parts that have direct physical contact with the machine and parts in the environment that have an effect on the machine without physical contact, e.g., laws. These relations between the environment and the machine have to be part of the graphic, as well. A context-pattern has to contain at least one graphical pattern. We use a UML-based notation for our graphical patterns that uses, e.g., stick figures for actors, but we also use notation elements that are not part of the UML, such as rectangles that symbolize an environment and all elements in the rectangle belong to this environment.

Templates. Templates contain additional information about elements in the graphical pattern. For example, a graphical pattern contains a graphical figure of a stakeholder and a corresponding template for the stakeholder can contain, e.g., the motivation of the stakeholder for using the machine and the relations to other stakeholders. Templates provide the means to attach further information to the graphical pattern. The reason for adding this refinement in a template and not in the graphical pattern is not to overload the graphical pattern with too many elements. Templates are optional elements of context-patterns, because not all graphical patterns require a refinement.

3. A CATALOG OF CONTEXT PATTERNS

In the following we will list our catalog of context-patterns. We will only elaborate the graphical part of the patterns as only this information is of importance for the rest of the paper.

3.1 Peer-to-Peer System Analysis Pattern

Aligning software systems to meet requirements is hard, which is even more difficult when a Peer-to-Peer (P2P)-based software system shall be developed. For example, the effects of churn, the random leaving or joining of peers in the system, can cause data loss. If a requirement exists stating that no data loss shall occur in the system, then churn presents a challenge that has to be considered for this requirement. A software engineer can design a

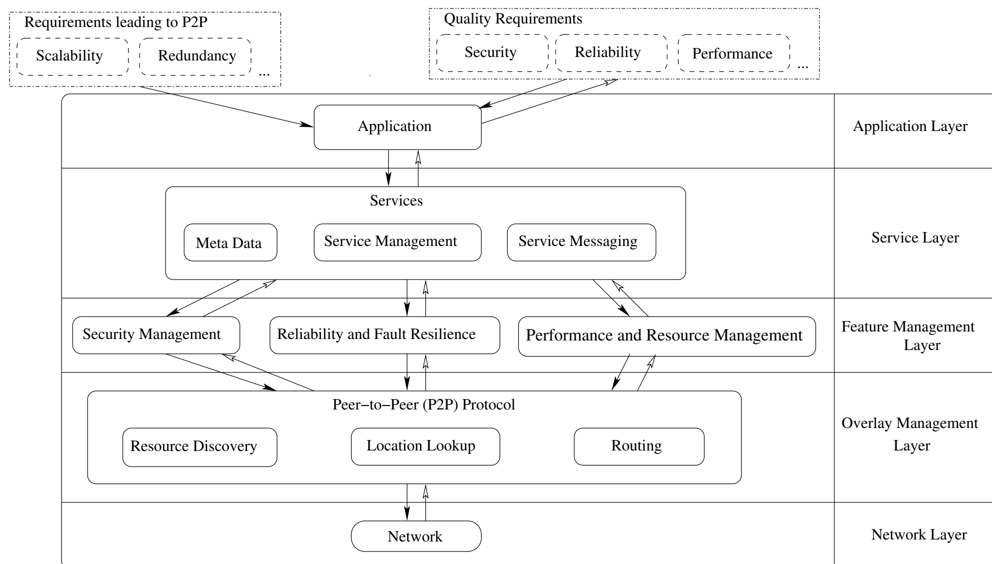

Fig. 1: P2P pattern

countermeasure if she is aware of the challenge. This, however, is difficult, because numerous challenges are caused by attributes of the P2P protocol or even the network layer.

We present a pattern-based method to identify existing challenges in a P2P-based software system. An initial pattern considers all layers of a P2P architecture and offers more detailed patterns for, e.g., P2P protocols. The instantiation of these patterns enables an analysis of the system's challenges and reveals the information in which layer each challenge originates. An extensive description of the context-pattern can be found in [Beckers and Faßbender 2012]. Our *P2P* pattern (see Figure 1 top) is based upon the P2P architecture from Lua et al. [Lua et al. 2005], which is derived from a survey of existing *P2P* systems. This survey describes *P2P* systems as layered architectures that contain at least the following layers.

The *Application Layer* concerns applications that are implemented using the underlying *P2P* overlay. For example, a Voice-over-IP (VoIP) application. The *Service Layer* adds application-specific functionality to the *P2P* infrastructure. For example, for parallel and computing-intensive tasks, or for content and file management. Meta-data describe what the service offers, for instance, content storage using *P2P* technology. Service messaging describes the way services communicate. The *Feature Management Layer* contains elements that deal with security, reliability and fault resiliency, as well as performance and resource management of a *P2P* system. All these aspects are important for maintaining the robustness of a *P2P* system. The *Overlay Management Layer* is concerned with peer and resource discovery and routing algorithms. The *Network Layer* describes the ability of the peers to connect in an ad hoc manner over the internet or small wireless or sensor-based networks.

3.2 Service-oriented Architecture Pattern

Our Service-oriented Architectures (SOA) pattern concerns eliciting domain knowledge for SOA. A detailed description of the context-pattern can be found in [Beckers et al. 2012]. The context-pattern was derived from a survey and research roadmap for SOA [Papazoglou et al. 2008] and validated using industry (e.g. [Arsanjani et al. 2007; Arsanjani et al. 2008]) and research (e.g. [Perepletchikov et al. 2008]) reports.

A SOA spans different layers [Beckers et al. 2012], which form a pattern on a SOA with technological focus, as depicted in Figure 2 on the top. The first and top layer is the *Business Domain* layer, which represents the

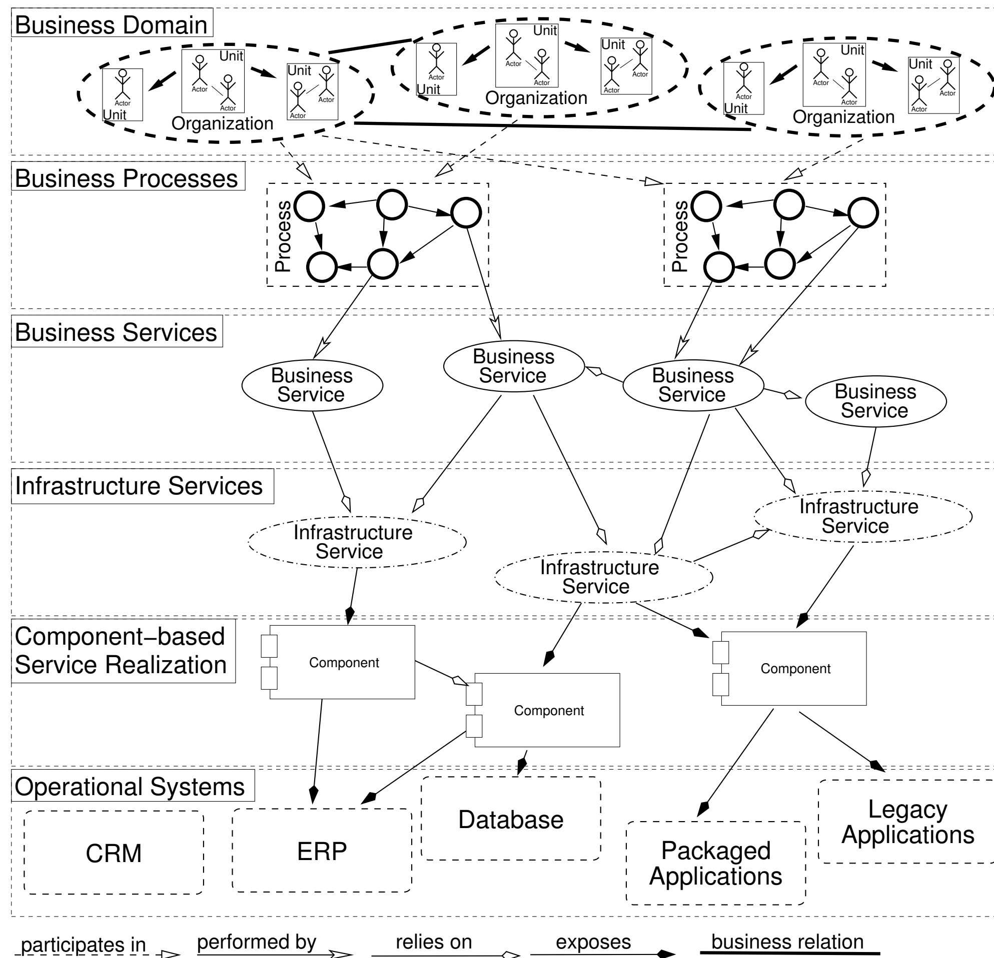

Fig. 2: SOA Layer Pattern

real world. It consists of *Organizations*, their structure and actors, and their *business relations* to each other. The second layer is the *Business Process* layer. To run the business, certain *Processes* are executed. Organizations *participate in* these processes. These processes are supported by *Business Services*, which form the *Business Service* layer. A business service encapsulates a business function, which *performs* a process activity within a business process. All business services rely upon *Infrastructure Services*, which form the fourth layer. The infrastructure services offer the technical functions needed for the business services. These technical functions are either implemented especially for the SOA, or they expose interfaces from the *Operational Systems* used in an organization. These operational systems, such as databases or legacy systems, are part of the last SOA layer at the bottom of the SOA stack.

In Figure 2 at the bottom, we adapted problem-based methods, such as problem frames by Jackson [Jackson 2001], to enrich the SOA layer pattern with its environmental context. The white area in Figure 2 (bottom) spans the SOA layers that form the machine. The business processes describe the behavior of the machine. The business services, infrastructure services, components, and operational systems describe the structure of the machine. Note

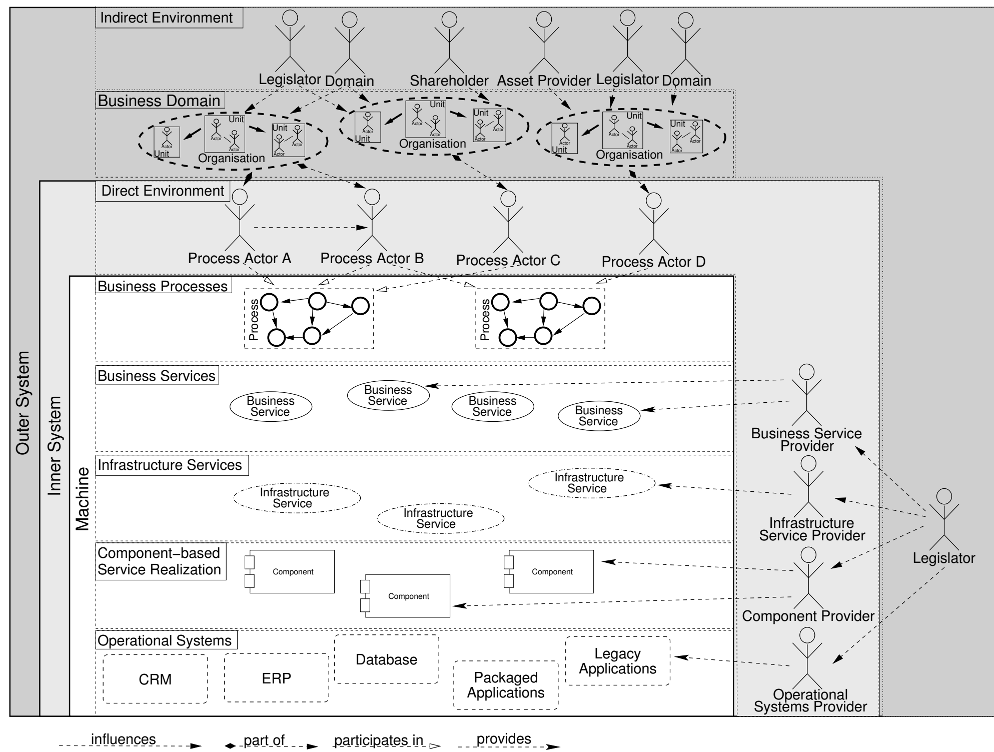

Fig. 3: SOA Layer Stakeholder Pattern

that the business processes are not part of the machine altogether, as the processes also include actors which are not part of the machine. Thus, the processes are the bridge between the SOA machine and its environment. The environment is depicted by the gray parts of Figure 2 (bottom). The light gray part spans the *Direct Environment* and includes all entities which participate in the business processes or provide a part, like a component, of the machine. An *entity* is, for example, something that exits in the environment independently of the machine or other entities. The dark gray part in Figure 2 (bottom) spans the indirect environment. It comprises all entities not related to the machine but to the direct environment. The Business Domain layer is one bridge between the direct and indirect environment. Some entities of the Direct Environment are part of organizations. Some entities of the Indirect Environment influence one or more organizations. The machine and the Direct Environment form the *inner system*, while the *outer system* also includes the Indirect Environment.

The entities we focused on for the stakeholder SOA pattern are stakeholders, because all requirements to be elicited stem from them. There are two general kinds of stakeholders. The *direct stakeholders* are part of the direct environment, while the *indirect stakeholders* are part of the indirect environment. We derived more specific stakeholders from the direct and indirect stakeholders, because these two classes are very generic. Process actors and different kinds of providers are part of the direct environment. Legislators, domains, shareholders and asset providers are part of the indirect environment. In Figure 2 (bottom), the resulting stakeholder classes are depicted as stick figures. For a detailed description of these stakeholders we refer to our previous work [Beckers et al. 2012].

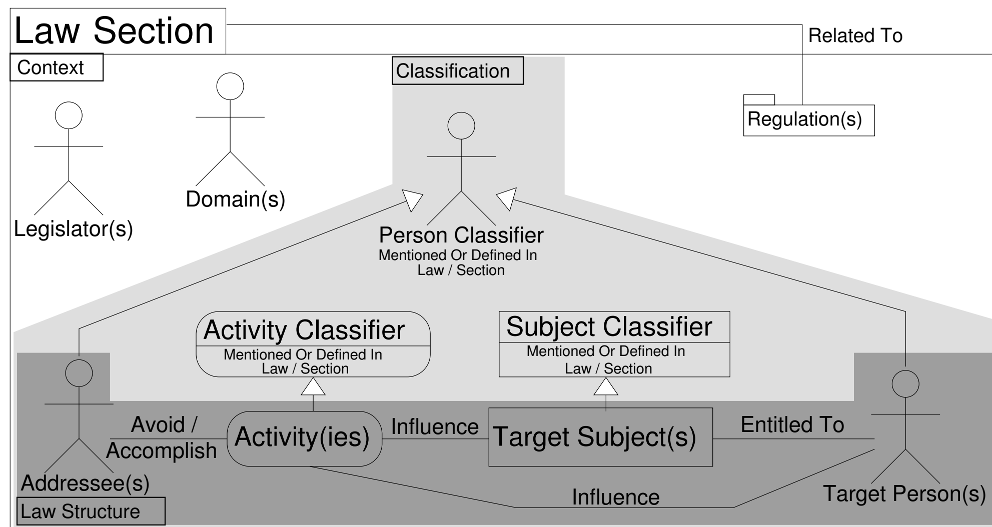

Fig. 4: Law Pattern

3.3 Law Pattern

Commonly, laws are not adequately considered during requirements engineering. Therefore, they are not covered in the subsequent system development phases. One fundamental reason for this is that the involved engineers are typically not cross-disciplinary experts in law and software and systems engineering. Hence, we present in this section a context-pattern for identifying laws and regulations including a method to systematically consider laws in the requirements engineering process. For our method we chose the German law as the binding law. The patterns are based on legal literature (e.g. [Schwacke 2003; Larenz 1983]) and validated using laws (e.g. the federal data protection act [Bundestag der Bundesrepublik Deutschland (parliamentary council of federal republic of Germany) 2009])

Based on the structure of laws (details can be found in [Beckers et al. 2012]), we define a *law pattern* shown in Fig. 4. The pattern consists of three parts: the dark grey part represents the *Law Structure*, the light gray part depicts the *Classification* to consider the specialization of the elements contained in the *Law Structure* in related laws or sections, and the white part considers the *Context*. The *Context* part of the law pattern contains the *Legislator(s)* defining the jurisdiction, and the *Domain(s)* clarifying for which domain the law was established.

As it is necessary to know in which context and relation a law is used, we introduce *Regulation(s)*, which are *Related To* the section at hand. *Regulation(s)*, *Legislator(s)*, and *Domain(s)* can be ordered in hierarchies, similar to classifiers. For instance, Germany is part of the EU and consists of several states.

Figure 5 shows our law identification pattern. The structure is similar to the law pattern in Fig. 4 to allow a matching of instances of both patterns. In contrast to the legal vocabulary used in the *Law Structure* of our law pattern, the wording for the elements in the dark gray colored *Core Structure* of our law identification pattern is based on terms known from requirements engineering. For example, the element *Asset(s)* in our law identification pattern represents the element *Target Subject(s)* in our law pattern.

Our law identification pattern takes into account that requirements are often interdependent (*Requirement(s)* in the *Context* part). Given a law relevant to a requirement, the same law might be relevant to the dependent requirements, too. Furthermore, the pattern helps to document similar dependencies for a given *Activity* using the *Related Process(es)* in the *Context* part.

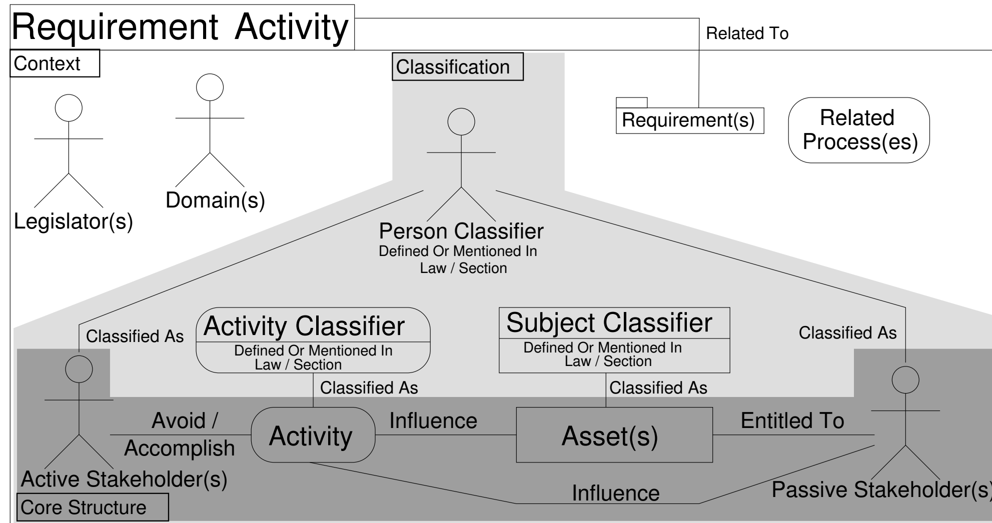

Fig. 5: Law Identification Pattern

3.4 Cloud System Analysis Pattern

We present our cloud system analysis pattern that helps to systematically describe cloud computing scenarios and identify assets in these scenarios. The pattern is based on several cloud standards and publications (e.g. [Armbrust et al. 2009; Mell and Grance 2009; Vaquero et al. 2008])

The *cloud system analysis pattern* shown in Fig. 6 provides a conceptual view on cloud computing systems and serves to systematically analyse stakeholders and technical cloud elements. The notation used to specify the pattern is based on UML[1] notation, i.e. the stick figures represent roles, the boxes represent concepts or entities of the real world, the named lines represent relations (associations) equipped with cardinalities, the unfilled diamond represents a *part-of* relation, and the unfilled triangles represent generalization.

A *Cloud* is embedded into an environment consisting of two parts, namely the *Direct System Environment* and the *Indirect System Environment*. The *Direct System Environment* contains stakeholders and other systems that directly interact with the *Cloud*, i.e. they are connected by associations. Moreover, associations between stakeholders in the *Direct* and *Indirect System Environment* exist, but not between stakeholders in the *Indirect System Environment* and the cloud. Typically, the *Indirect System Environment* is a significant source for compliance and privacy requirements.

The *Cloud Provider* owns a *Pool* consisting of *Resources*, which are divided into *Hardware* and *Software* resources. The provider offers its resources as *Services*, i.e. *IaaS*, *PaaS*, or *SaaS*. The boxes *Pool* and *Service* in Fig. 6 are hatched, because it is not necessary to instantiate them. Instead, the specialized cloud services such as *IaaS*, *PaaS*, and *SaaS* and specialized *Resources* are instantiated. The *Cloud Developer* represents a software developer assigned by the *Cloud Customer*. The developer prepares and maintains an *IaaS* or *PaaS* offer. The *IaaS* offer is a virtualized hardware, in some cases equipped with a basic operating system. The *Cloud Developer* deploys a set of software named *Cloud Software Stack* (e.g., web servers, applications, databases) into the *IaaS* in order to offer the functionality required to build a *PaaS*. In our pattern *PaaS* consists of an *IaaS*, a *Cloud Software Stack* and a *cloud programming interface (CPI)*, which we subsume as *Software Product*. The *Cloud Customer* hires a *Cloud Developer* to prepare and create *SaaS* offers based on the CPI, finally used by the

[1]Unified Modeling Language: `http://www.omg.org/spec/UML/2.3/`

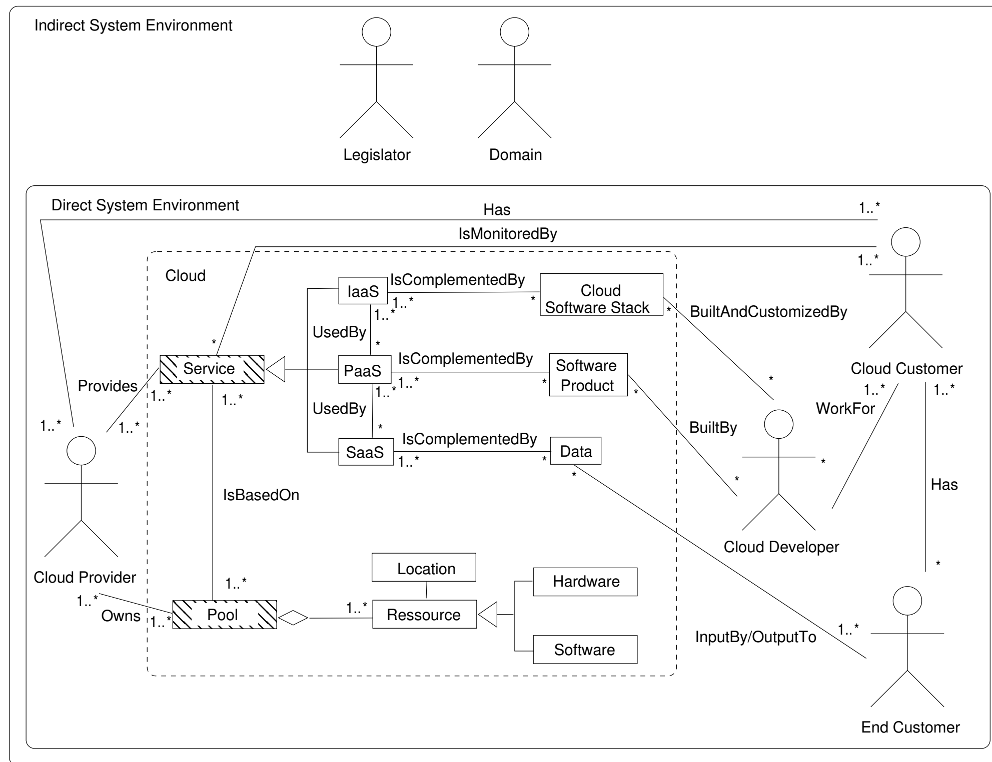

Fig. 6: Cloud System Analysis Pattern

End Customers. *SaaS* processes and stores *Data* in- and output from the *End Customers*. The *Cloud Provider*, *Cloud Customer*, *Cloud Developer*, and *End Customer* are part of the *Direct System Environment*. Hence, we categorize them as *direct stakeholders*. The *Legislator* and the *Domain* (and possibly other stakeholders) are part of the *Indirect System Environment*. Therefore, we categorize them as *indirect stakeholders*.

4. DERIVING A META-MODEL FOR PATTERN-BASED CONTEXT ELICITATION

The meta-model was derived in a bottom up way from the different patterns we described independently for different domains. For the process of deriving the general elements, which then form the meta-model, we started to analyze each context elicitation pattern in isolation. For each element in a context elicitation pattern we discussed what the general concept behind this element is or if it is a general concept in itself. Therefore, we set up a table containing the elements of the current pattern to be analyzed as rows and the conceptual elements as columns. For each concept found we checked if this concept was already covered in the table or not. In the case it was already covered we only added a cross to the table. In the case the concept was not covered by a conceptual element, we added a new column. After iterating over the elements of the pattern, we did a second step by adding the found conceptual elements as rows and analyzing for each of them if they could be further generalized in a reasonable way or not. This way we found also new conceptual elements. Hence, we had to do the second step several times. If nothing new was found, we finished the analysis. This way, we obtained the conceptual elements, which were candidates for the meta-model.

Table I shows the result of this phase for the SOA pattern. In this case, we analyzed two patterns in conjunction, because the stakeholder SOA pattern reuses many elements of the SOA layer pattern.

		General Concept												
		Layer	Stake-holder	Process	Active Re-source	Relation	Environ-ment	Indirect Environ-ment	Indirect Stake-holder	Direct Environ-ment	Direct Stake-holder	Ma-chine	Area	Re-source
SOA Layer Pattern Element	Business Organizations	x												
	Organization		x											
	Business Processes	x												
	Process			x										
	Business Services	x												
	Business Service				x									
	Infrastructure Services	x												
	Infrastructure Service				x									
	Component-based Service Realization	x												
	Component				x									
	Operational Systems	x												
	CRM				x									
	ERP				x									
	Database				x									
	Packaged Applications				x									
	Legacy Applications				x									
	Participates In					x								
	Performed By					x								
	Relies On					x								
	Exposes					x								
	Business Relation					x								
Stakeholder SOA Pattern Element	Outer System						x							
	Indirect Environment							x						
	Legislator								x					
	Domain								x					
	Shareholder								x					
	Asset Provider								x					
	Inner System						x							
	Direct Environment									x				
	Process Actor										x			
	Business Service Provider										x			
	Infrastructure Service Provider										x			
	Component Provider										x			
	Operational Systems Provider										x			
	Machine											x		
	Influences					x								
	Part Of					x								
	Provides					x								
Conceptual Element	Layer												x	
	Stakeholder													
	Process													
	Active Resource													x
	Relation													
	Environment												x	
	Indirect Environment						x							
	Indirect Stakeholder		x											
	Direct Environment						x							
	Direct Stakeholder		x											
	Machine												x	
	Area													
	Resource													

Table I. : Analysis of the SOA Layer Pattern and the Stakeholder SOA Pattern Elements

In a next phase we harmonized the conceptual elements by comparing the found elements, merging them if needed and setting up their relations. This way we got a coherent set of conceptual elements over all patterns.

In the last phase we had to choose which conceptual elements should be part of the meta-model. Table II shows the conceptual elements and in which of the patterns a corresponding element exists. Additionally, we selected for each pattern those elements which were not explicitly part of the pattern and checked if the missing element is an implicit part of the pattern. The patterns were also tagged with the information if there is a technical or organizational view provided or a combination of both. This is important to consider, because there might be elements which only occur in one of the views. Those elements might be excluded by just looking at the pure occurrence number, because they can only occur in a subset of the pattern. But those elements might be nevertheless important to capture aspects which are special for a view.

The general rule to include an element into the meta-model or not, was to add every element with an occurrence greater than three, which means the element occurs in more than the half of the patterns. In case of a view specific element, an occurrence of greater than two was sufficient, because the number of patterns associated with a view was four. Every element with an occurrence of two was subject to be discussed. The occurrence of an element was calculated only considering the explicit occurrence in a pattern.

	Type	technical	technical	Technical, organizational	technical, organizational	organizational	organizational
	Pattern	P2P Pattern	SOA Layer Pattern	Stakeholder SOA Pattern	Cloud Pattern	Law Identification Pattern	Law Pattern
Meta-model Element	Pattern	x	x	x	x	x	x
	Areas	x	x	x	x	x	x
	Machine	o	o	x	x		
	Environment		o	x	x	x	x
	Direct Environment		o	x	x	x	x
	Indirect Environment		o	x	x	x	x
	Layer	x	x	x			
	Process		x	x		x	x
	Activity		o	o		x	x
	Stakeholder		x	x	x	x	x
	Direct Stakeholder		o	x	x	x	x
	Indirect Stakeholder			x	x	x	x
	Resource	x	x	x	x	x	x
	Active Resource	x	x	x	x		
	Passive Resource				x	x	x
	Relation	x	x	x	x	x	x
uncovered Elements	Requirements	x				x	
	Requirements leading to P2P	x					
	Requirements Influenced by P2P	x					

x = contains element o = contains element implicit

Table II. : Overview of Elements of the Context-Patterns and their relation to the Meta-model

We had to discuss the conceptual elements requirement and machine. For the machine element it seemed that it is only part of patterns which mix-up the technical and the organizational view. So the first reason to include them is that for eliciting the context of a software problem the most usual pattern is one which mixes the technical and organizational view. This reason was supported by the experiences of the authors and context elicitation patterns, which are currently developed and studied, but which are not published yet. A second reason was the fact that the patterns with a more technical view contain the machine implicitly. For example, for the SOA Layer pattern the machine is not an explicit model element, but the extension to the Stakeholder SOA pattern shows that elements of the SOA layer pattern directly relate to the machine. We could not find similar evidence for the requirement element. Moreover, we think that the requirement is part of the phases which follow the context elicitation. For the P2P pattern we only added them for visualization means. Law Identification patterns are used in an iterative way. Thus, they are applied after eliciting the ideal context without legal restriction. Hence, this is a very specific case, which one cannot generalize. As a result, the requirement element is excluded and the machine element added to the context elicitation meta-model. Finally, we formed the meta-model as depicted in Fig. 7 out of the selected conceptual elements. The meta-model was modeled using the UML notation.

The root element is the `Pattern` itself. Each pattern consists of at least one `Area`. In general, an area contains elements of the same kind, view or level. An area can contain other areas to split it up and make it more fine-grained. An area can be the `Machine`, or an `Environment`, which contains in turn elements that have some kind of relation to the machine, or a `Layer`, which encapsulates elements of the same hierarchy level.

The environment can be further refined. There are elements which directly interact with the machine, captured in the `Direct Environment`. And there are elements which have an influence on the system via elements of the direct environment, captured by the `Indirect Environment`.

An element, which is part of an `Area`, can be a `Process`, a `Stakeholder`, or a `Resource`. A process describes some kind of workflow or sequence of activities. Therefore, it can contain `Activities`. A stakeholder describes a person, a group of persons, or organizational units, which have some kind of influence on the machine. A stakeholder can be refined to a `Direct Stakeholder`, who interacts directly with the machine, and an `Indirect`

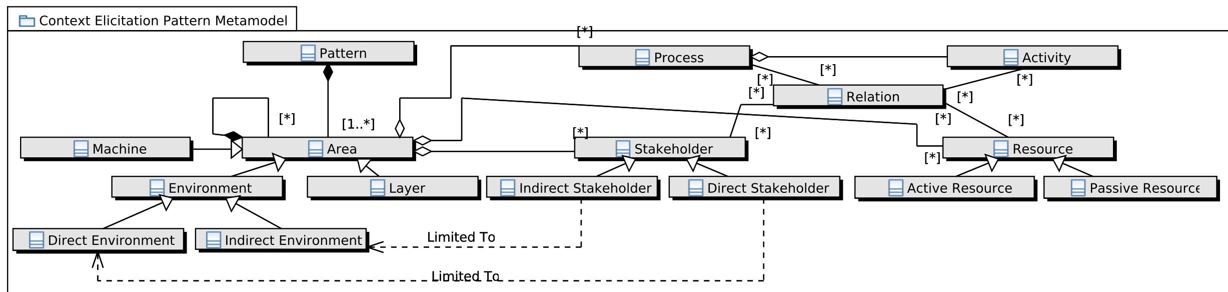

Fig. 7: Meta-Model for Context-Patterns

`Stakeholder`, who only interacts with direct stakeholders, but has some interest in or influence on the machine. A `Resource` describes some material or immaterial element, which is needed to run the machine or which is processed by the machine and which is not a stakeholder. A resource can be an `Active Resource` with some behavior or a `Passive Resource` without any behavior.

This meta-model has several benefits. First, it forms a uniform basis for our context-patterns, making them comparable. If a method already makes use of one of the patterns, it is now easy to generalize the usage to the elements of the meta-model. This enables one to replace a given used pattern by another one easily. Second, findings and results for one pattern can be transferred to the other pattern via a generalization to the meta-model elements. Third, the meta-model contains the important conceptual elements for context elicitation patterns. Thus, it is helpful to know this elements and search for them in a specific domain when setting up a new context-pattern for a domain. Fourth, it enables to form a pattern language for the context elicitation pattern. The common meta-model eases relating the patterns to each other.

5. APPLICATION OF THE META-MODEL

After the definition of the meta-model, we instantiated it for each of our context-patterns. Thus, we aligned all of the patterns to the same foundation making them comparable. Additionally, when integrating context elicitation patterns into requirements engineering methods, this can be done in general only referring to the context elicitation meta-model.

To check whether the meta-model is applicable to any context elicitation pattern, we instantiated the meta-model for all of the source context elicitation pattern and additionally a smart grid context-pattern. This pattern was not part of the set of patterns used for deriving the meta-model. Thus, it did not influence the meta-model. But in case the smart grid pattern can be fully mapped to the meta-model, we show some evidence that the meta-model is reasonable and useful in general.

For a structured elicitation of information about the context of a smart grid software, we derived a context-pattern for smart grids. We conducted a in depth analysis of several documents like the CC protection profiles for smart meters [Kreutzmann et al. 2011a; 2011b], the documentation of the OpenNode project [OPEN node project 2010; 2011], the documentation of the OpenMeter project [OPEN meter project 2009], Siemens case studies from the NESSoS project, and the Canadian smart grid implementation program [of Gas and Markets 2011b; 2011a]. The resulting elements and their mapping to the meta-model is shown in Fig. 8.

The root *Pattern* element is the *Smart Grid Pattern* itself. The smart grid pattern contains a *Direct* and an *Indirect Environment*, which map to the meta-model elements with the same name. Further, the smart grid pattern contains three *Areas*, namely the *Grid*, the *Micro Grid* and the *Micro Grid Element*. These areas contain different kinds of *Grid Elements*. A grid element is an *Active Resource*. Grid elements are connected by *Grid Element Relations* which are *Relations*. The direct environment contains different kinds of *Direct Stakeholders* which are

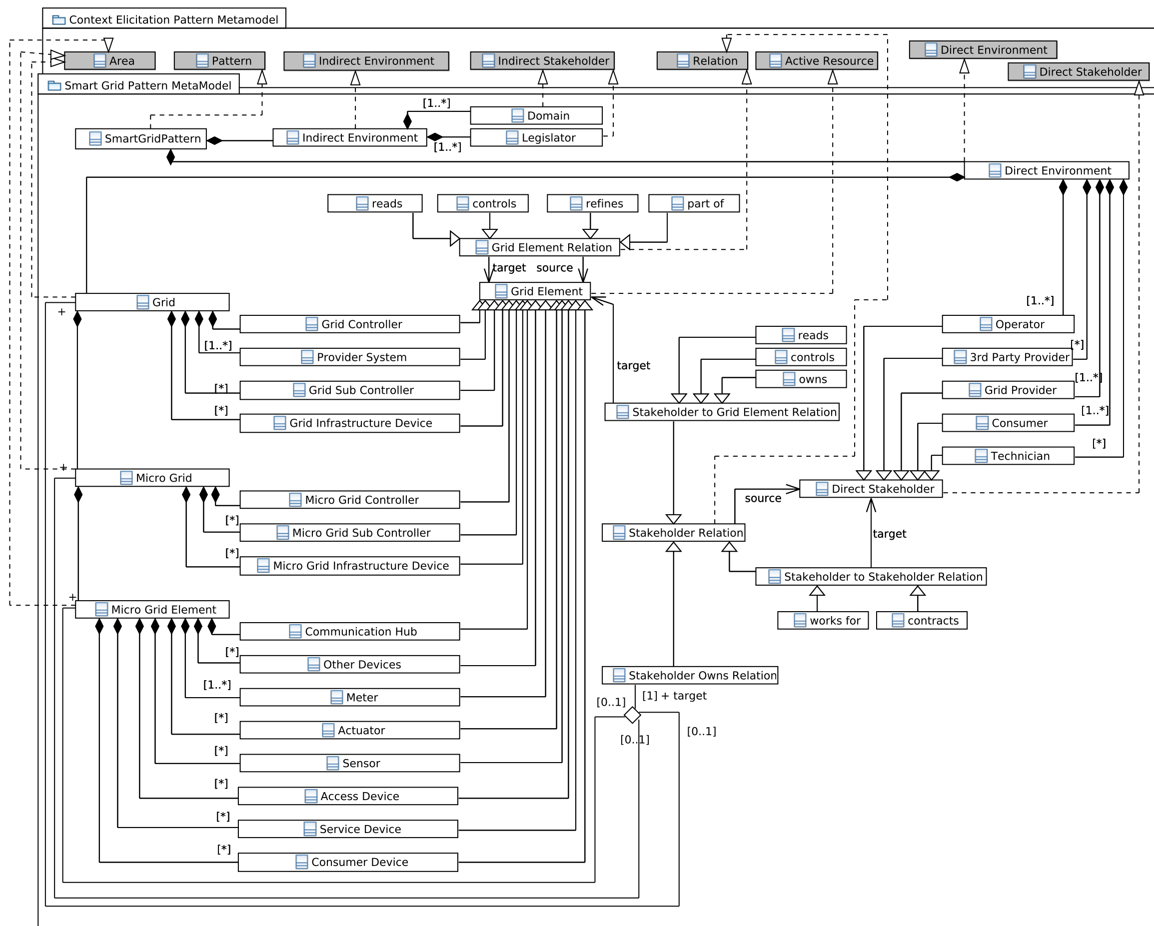

Fig. 8: Smart Grid Pattern Metamodel

Direct Stakeholders of the meta-model. Direct stakeholders are related to each other, to grid elements and to areas by different kinds of *Stakeholder Relations* which are *Relations*, too. The indirect environment contains *Domains* and *Legislators*, which are *Indirect Stakeholders*.

This application of the meta-model shows that the meta-model is sufficient for instantiating context-pattern. The smart grid pattern could be described using the meta-model without any problems. This way we prove that the generalization we did for forming the meta-model was reasonable.

6. CONCLUSIONS

We have presented a first step for creating a pattern language for context-patterns, which provide a structured means for eliciting domain knowledge. This step is creating a meta model for context-patterns. We illustrated our approach by showing context-patterns, e.g., patterns that consider specific technologies like Peer-to-Peer networks, specific types of architectures like cloud computing, and specific domains, e.g., the legal domain. All of these patterns relate to our meta model.

We can use instantiated patterns as a basis for writing requirements, deriving architectures or structured discussions about a specific domain. In addition, our patterns can be used outside the domain of software engineering for example for scope descriptions, asset identification, and threat analysis, when building an ISO 27001 [ISO/IEC 2005] compliant Information Security Management System [Beckers et al. 2013].

Our approach offers the following main benefits:

- A meta-model for describing context-patterns for various kinds of domain knowledge. This enables
  - comparing different context-patterns
  - transferring information between context-patterns instances
  - a guided description of new patterns using common elements and terms
  - forming a pattern language for context elicitation
- The patterns can be accompanied by further diagrams to support different views and detail levels to support scalability
- The patterns can be integrated into existing software development processes in order to improve context elicitation activities
- The patterns are useful beyond software engineering. For example, they can be applied to create the documentation for implementing security standards, e.g., ISO 27001 [ISO/IEC 2005].

The next step of our work is to formulated and describe the pattern language for context-patterns. The meta-model is one important step in this direction but not sufficient to form a full pattern language. We also aim at describing a structured method for deriving and describing context-patterns for not already covered domains.

In the future, we will extend our method to provide support for creating textual requirements patterns and derive software architectures. We will also provide the means for consistency checks of instantiated patterns and requirements. For example, it will be possible to check if a stakeholder in the domain knowledge pattern instance is considered in software requirements. In the future we will propose a number of consistency checks between our context-pattern and models that are derived using our patterns.

7. ACKNOWLEDGEMENTS

We thank our shepherd Hugo Ferreira for fruitful discussions and constructive feedback, as well as the members of our workshop Veli-Pekka Eloranta, Frank Frey, Carsten Hentrich, James Noble, Daniel Sagenschneider, Dietmar Schütz, Michael Weiss, Uwe Zdun and Christian Köppe for helpful discussions.

This research was partially supported by the EU project Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS, ICT-2009.1.4 Trustworthy ICT, Grant No. 256980) and the Ministry of Innovation, Science, Research and Technology of the German State of North Rhine-Westphalia and EFRE (Grant No. 300266902 and Grant No. 300267002).

REFERENCES

ALEXANDER, C. 1978. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R. H., KONWINSKI, A., LEE, G., PATTERSON, D. A., RABKIN, A., STOICA, I., AND ZAHARIA, M. 2009. Above the clouds: A berkeley view of cloud computing. Tech. rep., EECS Department, University of California, Berkeley.

ARSANJANI, A., GHOSH, S., ALLAM, A., ABDOLLAH, T., GARIAPATHY, S., AND HOLLEY, K. 2008. SOMA: a method for developing service-oriented solutions. *IBM Systems Journal 47,* 3, 377–396.

ARSANJANI, A., ZHANG, L.-J., ELLIS, M., ALLAM, A., AND CHANNABASAVAIAH, K. 2007. Design an SOA solution using a reference architecture. Tech. rep., IBM. `http://www.ibm.com/developerworks/library/ar-archtemp/`.

BECKERS, K., CÃŤTÃĽ, I., FAÃ§BENDER, S., HEISEL, M., AND HOFBAUER, S. 2013. A pattern-based method for establishing a cloud-specific information security management system. *Requirements Engineering*, 1–53.

BECKERS, K. AND FASSBENDER, S. 2012. Peer-to-peer driven software engineering considering security, reliability, and performance. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES) - 2nd International Workshop on Resilience and IT-Risk in Social Infrastructures (RISI 2012)*. IEEE Computer Society, 485–494.

BECKERS, K., FASSBENDER, S., HEISEL, M., AND MEIS, R. 2012. Pattern-based context establishment for service-oriented architectures. In *Software Service and Application Engineering*. LNCS 7365. Springer, 81–101.

BECKERS, K., FASSBENDER, S., KÜSTER, J.-C., AND SCHMIDT, H. 2012. A pattern-based method for identifying and analyzing laws. In *Proceedings of the International Working Conference onRequirements Engineering: Foundation for Software Quality (REFSQ)*. LNCS 7195. Springer, 256–262.

BUNDESTAG DER BUNDESREPUBLIK DEUTSCHLAND (PARLIAMENTARY COUNCIL OF FEDERAL REPUBLIC OF GERMANY). 2009. Bundesdatenschutzgesetz (Federal Data Protection Act). available at: http://www.gesetze-im-internet.de/englisch_bdsg/federal_data_protection_act.pdf.

FABIAN, B., GÜRSES, S., HEISEL, M., SANTEN, T., AND SCHMIDT, H. 2010. A comparison of security requirements engineering methods. *Requirements Engineering – Special Issue on Security Requirements Engineering 15,* 1, 7–40.

FERNANDEZ, E. B. AND PAN, R. 2001. A Pattern Language for Security Models. In *8th Conference of Pattern Languages of Programs (PloP)*.

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

FOWLER, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

HAFIZ, M., ADAMCZYK, P., AND JOHNSON, R. E. 2012. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. Onward! '12. ACM, New York, NY, USA, 139–158.

ISO/IEC. 2005. Information technology - Security techniques - Information security management systems - Requirements. ISO/IEC 27001, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC).

ISO/IEC. 2009. Information technology - Security techniques - Information security management systems - Overview and Vocabulary. ISO/IEC 27000, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC).

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.

LARENZ, K. 1983. *Methodenlehre der Rechtswissenschaft* 5. Ed. Springer.

LUA, E. K., CROWCROFT, J., PIAS, M., SHARMA, R., AND LIM, S. 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials 7*, 72–93.

MELL, P. AND GRANCE, T. 2009. The NIST definition of cloud computing. Working Paper of the National Institute of Standards and Technology (NIST).

NIKNAFS, A. AND BERRY, D. M. 2012. The impact of domain knowledge on the effectiveness of requirements idea generation during requirements elicitation. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*. 181 –190.

OF GAS, O. AND MARKETS, E. 2011a. Smart Metering Implementation Programme, Response to Prospectus Consultation, Design Requirements. Tech. rep., Office of Gas and Electricity Markets.

OF GAS, O. AND MARKETS, E. 2011b. Smart Metering Implementation Programme, Response to Prospectus Consultation, Overview Document. Tech. rep., Office of Gas and Electricity Markets.

OPEN METER PROJECT. 2009. Requirements of AMI. Tech. rep., OPEN meter project.

OPEN NODE PROJECT. 2010. Evaluation of general requirements according state of the art . Tech. rep., OPEN node project.

OPEN NODE PROJECT. 2011. Functional Use cases. Tech. rep., OPEN node project.

PAPAZOGLOU, M. P., TRAVERSO, P., DUSTDAR, S., AND LEYMANN, F. 2008. Service-oriented computing: a research roadmap. *Int. J. Cooperative Inf. Syst. 17,* 2, 223–255.

PEREPLETCHIKOV, M., RYAN, C., FRAMPTON, K., AND SCHMIDT, H. W. 2008. Formalising service-oriented design. *Journal of Software 3,* 2, 1–14.

SCHWACKE, P. 2003. *Juristische Methodik mit Technik der Fallbearbeitung* 4. Ed. Kohlhammer Deutscher Gemeindeverlag.

TOLEDANO, M. D. D. 2002. Meta-patterns: Design patterns explained. Tech. rep.

VAQUERO, L. M., RODERO-MERINO, L., CACERES, J., AND LINDNER, M. 2008. A break in the clouds: Towards a cloud definition. *Special Interest Group on Data Communication (SIGCOMM) Computer Communication Review 39,* 1, 50–55.