Towards Systematic Integration of Performance and Security Requirements into Software Architecture *

Azadeh Alebrahim
azadeh.alebrahim@uni-duisburg-essen.de

Denis Hatebur
denis.hatebur@uni-duisburg-essen.de

Maritta Heisel
maritta.heisel@uni-duisburg-essen.de

Software Engineering
Department of Computer Science and Applied Cognitive Science
Faculty of Engineering, University Duisburg-Essen
Duisburg, Germany

ABSTRACT

We present a model- and pattern-based method that allows software engineers to take quality requirements into account right from the beginning of the software development process. The method comprises requirements analysis as well as the derivation of a software architecture from requirements documents. In that architecture, quality requirements are reflected explicitly.

For requirements analysis, we use an enhancement of the problem frame approach [14], where software development problems are represented by *problem diagrams*. In our enhanced version of the problem frame approach, we use UML notation, and we have added the possibility to complement functional requirements with quality requirements, such as security or performance requirements.

The derivation of a software architecture starts from a set of problem diagrams, annotated with functional as well as quality requirements. First, we set up an initial software architecture, taking into account the decomposition of the overall software development problem into subproblems. Next, we incorporate quality requirements into that architecture by using security or performance patterns or mechanisms. To obtain the final architecture, (functional) design patterns are applied. The method is tool-supported, which allows developers to check semantic integrity conditions in the different models.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Security, Performance

*Part of this work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft - DFG) under grant number HE3322/4-1.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
QoSA 2011, Boulder, Colorado, USA
Copyright 2011 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Keywords

Quality-driven design, quality requirements, software architecture, software architecture design

1. INTRODUCTION

Taking quality (or non-functional) requirements into account when developing a software architecture is a demanding task, for which satisfactory solutions are still sought for. There are several reasons for this situation. First, quality requirements must be elicitated, analyzed, and documented as thoroughly as functional ones, which is often not the case. Second, requirements engineering and architectural design are often regarded as separate phases in software development, between which no seamless connection and no feedback exists. Hence, the knowledge gained in the requirements engineering phase is not used in a systematic way when developing a software architecture. Third, the current techniques for incorporating quality requirements into software architectures are even less developed than the ones that concentrate on functional requirements only. As a result, quality requirements seem to be fulfilled more "by chance" than by construction.

In this paper, we want to contribute to overcome this unsatisfactory situation. We present a method that

1. provides a seamless transition from requirements analysis to architectural design,
2. takes quality requirements (in particular, security and performance requirements) into account explicitly,
3. is model- and pattern-based, and for which
4. tool support exists.

We consider security and performance requirements, because they are quite different in nature. Security requirements can often be transformed into functional ones. For example, the confidential transmission of data can be achieved through encryption, which is an added functionality. Performance requirements, on the other hand, can hardly be transformed into functional ones. Therefore, these two kinds of requirements are appropriate representatives of quality requirements. If these two can be treated in a similar way, we may hope that our results are generalizable also for other kinds of quality requirements.

As a basis for requirements analysis, we use Jackson's problem frame approach [14]. We have carried over problem frames to UML [21] by defining a specific UML profile, and we have implemented a tool supporting requirements

analysis and architectural design based on problem frames [12]. The tool, called *UML4PF*, provides the possibility to automatically check semantic integrity conditions for individual requirements or architectural models, as well as coherence conditions between different models. As a basis for architectural design, we use a method that we developed for deriving architectures based on functional requirements [7].

In the present paper, we extend our previous requirements analysis and architectural design methods by explicitly taking into account quality requirements. The analysis documents are extended by quality requirements that complement functional ones. For this purpose, we have extended the UML profile described in [12]. The so enhanced problem descriptions form the starting point for architectural design. In a first step, we proceed similarly as described in [7]. That is, we define an initial software architecture that is oriented on the decomposition of the overall software development problem into subproblems. In a second step, we transform that architecture according to the quality requirements to be considered. For this purpose, we apply appropriate security or performance patterns, or we introduce components providing proven security or performance mechanisms, such as encryption or load balancing. Furthermore, we apply functional design patterns [10], such as *Facade*, to obtain a clean and modular software architecture. Finally, we have defined quality stereotypes that serve as hints for implementers.

The rest of the paper is organized as follows. In Sect. 2, we introduce the case study that serves as a running example throughout the paper. We then present the basics on which our method builds in Sect. 3, namely problem frames and security and performance patterns and mechanisms. In Sect. 4, we present the UML profile we defined to carry over the problem frame approach to UML, as well as the tool UML4PF. Section 5 is devoted to describing our method in detail. Related work is discussed in Sect. 6, and conclusions and future work are given in Sect. 7.

2. CASE STUDY

As a case study, we use a chat application. Users should be able to log into the system after they are successfully registered. They should be able to join given chat rooms, where they can communicate with other chat participants in the same chat room. The system mission can be described as follows:

> "A text-message-based communication platform should be developed, which allows multi-user communication via private I/O devices."

After that, functional and quality requirements should be elicitated. In this paper, we focus on the most important functional requirement and its corresponding quality requirements. The functional requirement in focus is:

> *Communicate*: Users can send text messages to a chat room, which should be shown to the users in that chat room in the current chat session in the correct temporal order on their displays.

We consider some corresponding quality requirements. As a performance requirement we define the following:

> *Response Time*: The sent text message should be shown on the receiver's display in less than 1500 ms in 95% of all cases.

Note that we need to describe a usage model in order to fulfill a performance requirement appropriately. We capture the usage model in a *Load Profile* that specifies the size of the data to be transmitted, the number of concurrent users, and their behavior. We will describe the load profile in Sect. 5.2 in detail.

As a security requirement we consider the following:

> *Confidentiality*: Text messages should be transmitted in a confidential way.

Besides the confidentiality of text messages, their *integrity* is also important. For reasons of space, however, we do not address integrity in this paper.

In following sections, we describe our method to derive a quality-based software architecture from problem descriptions, and illustrate its application to the chat case study with the aforementioned requirements.

3. BASIC CONCEPTS

In this section, we introduce the basic concepts our method relies on. First, we describe a requirements engineering process based on problem frames. Second, we describe a set of established patterns and mechanisms that we integrate into software architecture in order to meet performance and security requirements.

3.1 Requirements Description using Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Michael Jackson [14], 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* (e.g., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements.

Software development with problem frames proceeds as follows: first the environment in which the machine will operate is represented by a *context diagram*. A context diagram consists of machines, domains and interfaces. Then, the problem is decomposed into subproblems, which are represented by *problem diagrams*. A problem diagram consists of a submachine of the machine given in the context diagram, the relevant domains, the interfaces between these domains, and a requirement. Figures 1 and 3–5 show problem diagrams in UML notation.

3.2 Mechanisms and Patterns for Performance

To satisfy performance requirements, different mechanisms, also called patterns [9], are available. We describe two such mechanisms.

3.2.1 Load Balancing

Load Balancing is a mechanism that is used to distribute computational load evenly over two or more hardware components. The load balancing pattern consists of a component, which is called *Load Balancer*, and multiple hardware components that implement the same functionality. The load balancer can be realized as a hardware or a software

component. Note that the load balancer itself may become a bottleneck and therefore should be analyzed appropriately.

Applying this pattern may also increase the availability of an application or a component through redundancy. This needs additional mechanisms for the load balancer to switch from a failed component to a working component in the case of a failure.

3.2.2 Master-Worker

Master-Worker makes it possible to serve requests in parallel, similarly to load balancing. In contrast to load balancing that distributes the load through hardware components, the master-worker pattern provides a software solution. When a request is complex and takes a long time to complete, it should be divided into parallel smaller tasks. This pattern consists of a software component, which is called *Master* and two or more other software components, called *Worker*. The task of the master is to divide the request into parallel tasks and to forward them to the workers. The worker components manage the smaller tasks. If the required hardware does not exist to increase the performance or buying extra hardware is not affordable, applying the master-worker pattern is the only option for speed-up. Since parallelization algorithms could cause overhead cost, it should be determined, under which conditions applying this pattern is profitable.

3.3 Mechanisms and Patterns for Security

Like design patterns for functional requirements, security patterns for security requirements are available. Although these patterns do not cover all security issues, they provide a structured basis to integrate security into software architecture. We describe the security pattern *Single Access Point* and and the *Encryption* mechanism that can be applied when developing secure applications. We only give an overview and do not describe them in detail. A collection of security patterns can be found in [25, 19, 26].

3.3.1 Single Access Point

It may be difficult and expensive to provide security for an application, which has multiple entry points. Protecting an application is easier when there is only one way to access an application. The typical solution is applying the *Single Access Point* pattern, which provides only one entry point into the system. An example for this pattern is to create a login screen that collects user information such as user name and password.

3.3.2 Encryption

Encryption is an important means to achieve confidentiality. A plaintext is encrypted using a secret *key* and decrypted either using the same key (symmetric encryption) or a different key (asymmetric encryption). One advantage of symmetric encryption is that it is faster than asymmetric encryption. The disadvantage is that both communication parties must know the same key, which has to be distributed securely or negotiated. Asymmetric encryption uses key pairs. A sender uses the public key of the receiver to encrypt a message, and only the receiver can decrypt the message using its private key. In asymmetric encryption, there is no key distribution problem, but a trusted third party is needed that issues the key pairs. It is also slower than symmetric encryption, which may cause performance problems.

4. TOOL-SUPPORTED REQUIREMENTS ENGINEERING

It is important that the results of the requirements analysis with problem frames can be easily re-used in later phases of the development process. Since UML is a widely used notation to express analysis and design artifacts in a software development process, we defined a new UML profile [12, 7] that extends the UML meta-model to support problem-frame-based requirements analysis with UML. This profile can be used to create the diagrams for the problem frame approach. To address quality requirements in the requirement engineering process we enhance our UML profile with annotations for quality requirements as stereotypes. In addition, the tool UML4PF supports the requirements engineering process as well as architectural design using the UML profile.

4.1 UML Profile for Problem Frames

As mentioned in Sect. 3.1, during the problem analysis phase, a number of diagrams are set up that provide a detailed description of the software development problem to be solved. Using specialized stereotypes, our UML profile allows us to express the different diagrams occurring in the problem frame approach using UML diagrams.

A class with the stereotype ≪`machine`≫ represents the software to be developed (possibly complemented by some hardware). Jackson distinguishes the domain types biddable domains that are usually people, causal domains that comply with some physical laws, and lexical domains that are data representations. The domain types are modeled by the stereotypes ≪`BiddableDomain`≫ and ≪`CausalDomain`≫ being subclasses of the stereotype ≪`Domain`≫. A lexical domain (≪`LexicalDomain`≫) is modeled as a special case of a causal domain (see Fig. 1). To describe the problem context, a *connection domain* between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. They are modeled as classes with the stereotype ≪`ConnectionDomain`≫. Connection domains are, e.g., video cameras, sensors, or networks. This kind of modeling allows one to add further domain types, such as ≪`DisplayDomain`≫ (introduced in [8]), being a special case of a causal domain.

In problem diagrams, *interfaces* connect domains, and they contain shared phenomena. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation *U!sendTM* (between *CA_communicate* and *User*) means that the phenomenon *sendTM* is controlled by the domain *User*. The interfaces are marked with specializations of the stereotype ≪`connection`≫, e.g., a user interface (≪`ui`≫) between *User* and *CA_communicate* machine in Fig. 1.

The stereotype ≪`requirement`≫ represents a functional requirement. When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`constrains`≫. Such a constrained domain is the core of any problem description, because it has to be controlled according to the requirements.

A requirement may refer to several domains in the en-

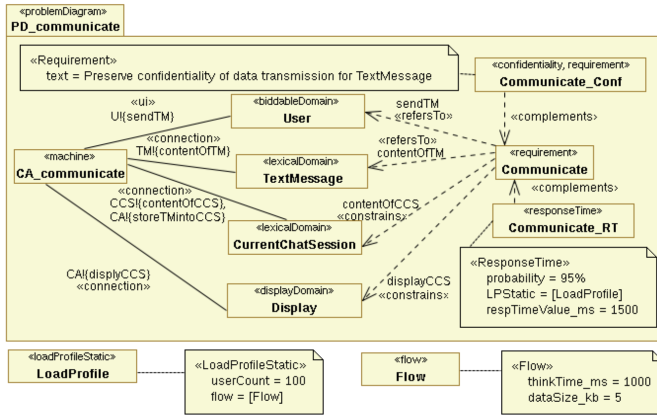

Figure 1: Problem Diagram for the requirement *Communicate*

vironment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`refersTo`≫.

The problem diagram in Fig. 1 describes the requirement *Communicate* in more detail, e.g., it describes that the *CA_communicate* machine can show to the *User* the *CurrentChatSession* on its *Display* (*CA!*{*displayCCS*}). The requirement constrains the *CurrentChatSession* of the *User* and its *Display*. The requirement refers to the users and the text messages.

The problem frame approach substantially supports developers in analyzing problems to be solved. It points out what domains have to be considered, and what knowledge must be described and reasoned about when analyzing a problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge* and are specified in *domain knowledge diagrams*. Domain knowledge consists of *assumptions* annotated with the stereotype ≪`assumption`≫ and *facts* annotated with the stereotype ≪`fact`≫. Assumptions are conditions that are needed, so that the *requirements* are accomplishable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment regardless of how the machine is built.

4.2 Annotating Problem Descriptions with Quality Requirements

The problem frame approach proposed by Jackson provides a method that addresses functional requirements only. Quality requirements are not considered. We extend our UML profile for problem frames to complement functional requirements with quality requirements. Classes with stereotypes such as ≪`confidentiality`≫, ≪`integrity`≫, ≪`responseTime`≫ or ≪`throughput`≫ represent quality requirements. The dependency from a quality requirement to a requirement is expressed with the stereotype ≪`complements`≫ (see Fig. 1). The profile can be extended with other quality requirements, e.g., usability. From the problem descriptions, we derive a software architecture that is suitable to solve the software development problem specified by the problem descriptions. The elements of problem diagrams, namely machine and lexical domains, can be mapped to components of an architecture in a fairly straightforward way.

4.3 Tool Support

We provide tool support for the software development process based on the
problem frame approach. Our tool, called UML4PF [1], can be used to create diagrams, which are mapped to a part of a global model and a graphical representation of this part. Basis is the Eclipse platform [1] together with its plug-ins *Eclipse Modeling Framework* (EMF) [2] and OCL. Our UML profile for problem frames is conceived as an Eclipse plug-in, extending the EMF meta-model.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus [3] as it is available as an Eclipse plug-in, open-source, and EMF-based.

To ensure the integrity and coherence of the model, we set up OCL constraints. Based on the model information, UML4PF can automatically detect semantic errors in the model by evaluating the constraints. Elements of the created model can be re-used in later development steps. We can also validate that the artifacts of later development steps are consistent with the requirements engineering diagrams For more details, see [12].

5. DERIVING QUALITY-BASED ARCHITECTURES FROM PROBLEM DESCRIPTIONS

We now present our method to derive software architectures, taking quality requirements into account. The method comprises requirements analysis, as well as the derivation of a software architecture from requirements documents.

5.1 Problem Diagrams

As described in Sect. 3.1, the first step in the software development process based on problem frames is to create a context diagram, which represents the environment in which the machine will operate. For reasons of space we do not show this diagram for the chat application and continue with the problem decomposition step. We decompose the whole problem into subproblems represented by problem diagrams. Each problem diagram describes one subproblem with the corresponding requirement. We focus on the requirement *Communicate* described in Sect. 2. The corresponding problem diagram using our UML profile for problem frames is depicted in Fig. 1. It consists of the domains *User, TextMessage, CurrentChatSession* and *Display*. The requirement *Communicate* refers to the domains *User* and *TextMessage*, expressed by the stereotype ≪`refersTo`≫ and constrains the domains *CurrentChatSession* and *Display*, expressed by the stereotype ≪`constrains`≫.

5.2 Annotate Problem Diagrams with Quality Requirements

In this step, we address quality requirements by annotating problem diagrams with suitable stereotypes. In Sect. 2, we defined one security and one performance requirement related to the functional requirement *Communicate*. The requirement *Communicate* is complemented by the confidentiality requirement *Communicate_conf* that requires confidentiality of data transmission for *TextMessage* and the response time requirement *Communicate_RT*. The response time requirement states that the response time for sending

[1] Available under
http://swe.uni-duisburg-essen.de/en/research/tool/

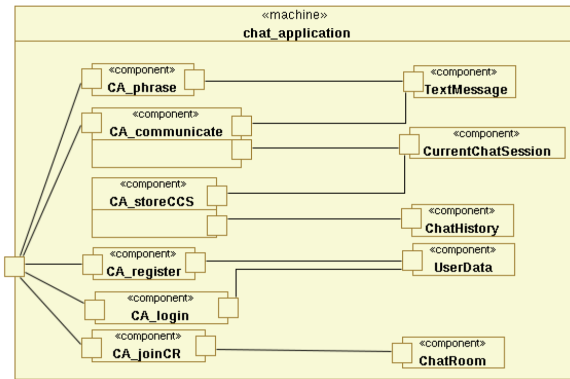

Figure 2: Initial Architecture

text messages should be less than 1500 ms in 95% of all cases by using a defined load profile (attributes for ≪`responseTime`≫ in Fig. 1). The load profile specifies the load by defining the attributes *userCount* and *flow*. The attribute *userCount* describes the number of concurrent users, in this case 100, and *flow* describes a scenario by defining the attributes *thinkTime_ms*, which is the time that the user waits between two actions and *dataSize_kb*. The think time in our case is 1000 ms, and the maximum data size is 5 kb.

5.3 Initial Architecture

During the requirements analysis phase, we create a set of problem diagrams, annotated with quality requirements. In the initial architecture, each machine domain in a problem diagram is mapped to a component. The initial architecture for the chat consists of one component for the overall machine with stereotype ≪`machine`≫, in our case, *chat_application*, see Fig. 2. The submachines associated with the subproblems of the chat application, including *Communicate* are components of the *chat_application*. There is one component for each submachine identified in the problem diagrams. If lexical domains should be part of the machine, they are contained in the initial architecture, e.g., *TextMessage* in Fig. 2.

5.4 Choose Design Alternative

In order to refine the initial architecture, we need to take a design decision. We decide on the kind of the application, e.g., client-server, peer-to-peer, or standalone, which is either given by the stakeholder or decided by the software architect. For standalone applications, we can skip two steps and continue with the step described in Sect. 5.4.3.

For distributed applications, we split the components of the initial architecture so that each component is allocated to only one computer. In the following, we describe the method for a client-server architecture in more detail. Peer-to-peer systems can be developed according to a similar procedure.

5.4.1 Split Problem Diagrams

Due to the fact that requirements analysis and architectural design are often intertwined, a separation of these two steps is difficult. Therefore, they should be considered concurrently during the software development process. Design decisions can constrain meeting the requirements, and changing requirements affects the architecture. Nuseibeh calls this process the Twin Peaks model [18]. We proceed in

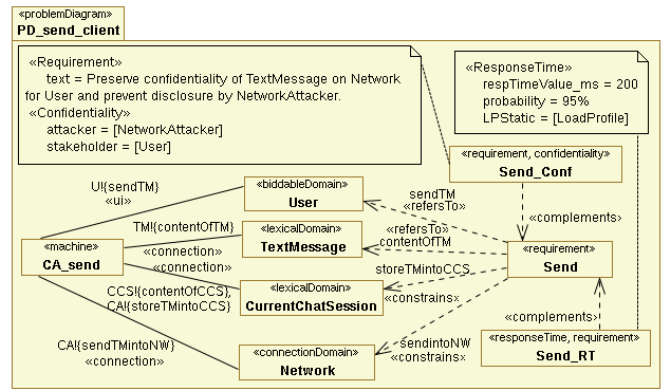

Figure 3: Problem Diagram for the requirement *Send* annotated with quality requirements

a similar way. After having chosen a client-server architecture for the chat application, we go back to the requirements description and decompose the problem diagrams in such a way that each subproblem is allocated to only one of the distributed components. This may lead us to introduce connection domains, e.g., networks (see Sect. 4.1). In the example of the chat application, the problem diagram depicted in Fig. 1 is split into three problem diagrams, which address the problems of sending text messages to the server that belongs to the client (see Fig. 3), forwarding text messages from the server to the receivers that belongs to the server (see Fig. 4), and receiving text messages that belongs to the client (see Fig. 5). For each of these three subproblems, we introduced the connection domain *Network* to achieve the distribution.

5.4.2 Split Quality Requirements

Analogously to splitting the problem diagrams and so splitting the functional requirements, we also have to split the corresponding quality requirements. In case of a response time requirement, the response time should be divided so that all subproblems together satisfy the desired response time. The *Communicate* requirement states a response time of 1500 ms maximum. This must be achieved through the three subproblems *Send, Forward* and *Receive*. We must also consider the time that the data needs to be transported over the network. In our case study, each of the machines *CA_send* and *CA_forward* is required to send a text message to the server or to forward the text message to the receivers, respectively, within 200 ms. The machine *CA_receive* may take 300 ms to process the received text message and display it, in 95% of all cases with the load profile we described in Sect. 5.2. This leaves 800 ms to transmit data from the client to the server and back. As mentioned in Sect. 4.1, we specify assumptions and facts about the environment in a domain knowledge diagram. We cannot meet the performance and specifically response time requirements, if we have no knowledge about the real circumstances in the environment. Therefore we specify knowledge about the network and the computational power of clients and server in the domain knowledge diagram for performance depicted in Fig. 6. It contains specific knowledge about client and server, e.g., the number of processor cores, processor speed and memory. Additionally, we assume that the response time to transmit data over a network with 64 kb/s minimum is 400 ms.

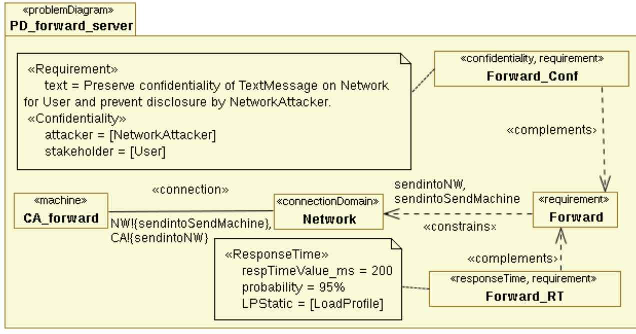

Figure 4: Problem Diagram for the requirement *Forward* annotated with quality requirements

To fulfill the confidentiality requirement for the problem *PD_communicate* (Fig. 1), we require confidentiality for each subproblem. Therefore, we annotate each subproblem with a corresponding refined confidentiality requirement. This requirement contains a *stakeholder* that is interested in preserving the confidentiality of data, and an *attacker* that the chat application should be protected against, as attributes. The stakeholder in our case is the *User*, and the attacker is a *NetworkAttacker* who is able to attack the data transported over the network. Possible attributes of an attacker are specified more precisely in [11]. The refined confidentiality requirements for each subproblem are shown in Figs. 3–5.

5.4.3 Concretized Quality Problem Diagrams

The goal of this step is to find solution approaches in terms of mechanisms and patterns to prepare for solving the given security and performance problems. We have given examples of such solutions in Sect. 3.2. We elaborate the problem diagrams annotated with quality requirements from the previous step by introducing domains reflecting specific solution approaches. We call the elaborated problem diagrams containing solution approaches *Concretized Quality Problem Diagrams*.

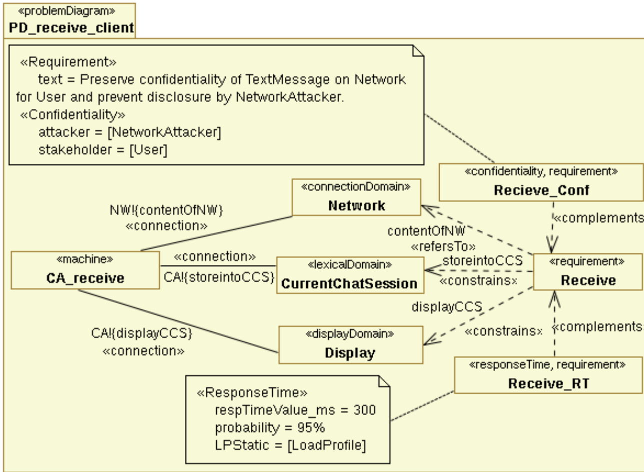

Figure 5: Problem Diagram for the requirement *Receive* annotated with quality requirements

For example, the problem diagram for the *Send* problem describes the problem of sending text messages with two additional quality requirements for security and performance, respectively (see Fig. 3). The requirement for per-

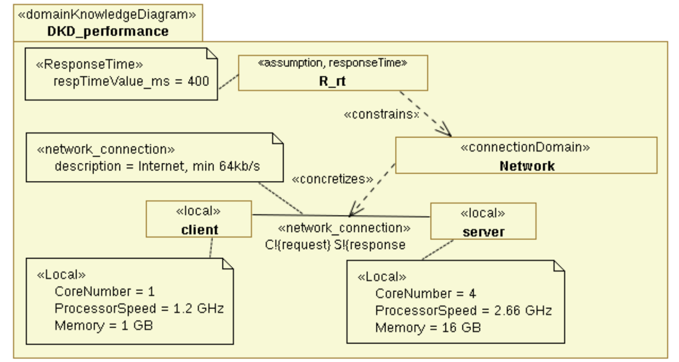

Figure 6: Domain Knowledge Diagram for performance

formance states that sending a text message should be performed within 200 ms in 95% of all cases. However, there is no architectural mechanism or pattern that can achieve the performance requirement for the *Send* problem. If the fulfillment of a quality requirement relies on the implementation, we annotate the corresponding machine with a stereotype that serves as a hint to develop a particularly efficient implementation or an implementation that does not leak information. In this case, we annotate the *CA_send* machine with the stereotype ≪`responseTime`≫ (see Fig. 7). But even if an additional functionality (or a component) is added to fulfill the requirement, it may be useful to add stereotypes that serve as hints for implementation. Therefore, we add the stereotype ≪`confidentiality`≫ to the *Encryption* machine in Fig. 7.

The security requirement describes that a text message should be transmitted confidentially over an insecure network. To take this quality requirement into account, we specify the concretized quality problem diagram given in Fig. 7 that provides an encryption mechanism to solve the problem. We first apply a symmetric and then an asymmetric encryption mechanism. In order to solve the problem using these mechanisms, we need to introduce a new machine *Encryption* that provides the encryption functionality as a part of the *CA_send* machine. The *Encryption* machine encrypts the text message with a generated random number that serves as a symmetric key. For each receiver, we encrypt the symmetric key with its public key, using an asymmetric encryption mechanism. For this reason, we introduce the new domain *ReceiverUserPublicKey*. The sender machine sends the message and the encrypted symmetric key to the server, and the server forwards them to all receivers. Since encryption takes time, we annotate the *Encryption* machine with the stereotype ≪`responseTime`≫ to point implementers towards efficient algorithms and implementations.

On the receiver side, we introduce a new machine, *Decryption*, as a part of the *CA_receive* machine. Thus, the receiver is able to decrypt the symmetric key with its own private key (*ReceiverUserPrivateKey*). With the decrypted symmetric key, the receiver can decrypt the text message (see Fig. 8).

The alternative solution is to encrypt the text message through the sender with a public key that belongs to the server and send the encrypted text message to the server. The server is able to decrypt the text message with its own private key and encrypt it again with the public keys of each

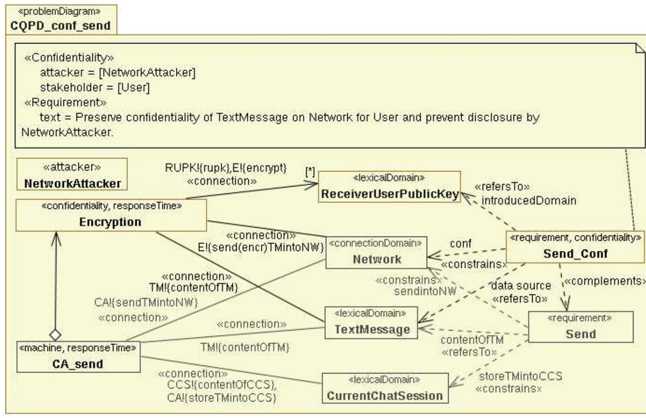

Figure 7: Concretized Quality Problem Diagram for the quality requirement ***Send_Conf***

of the receivers. In this scenario, the server needs more time for processing a message. Therefore, we decide to use the first solution.

The *Decryption* machine is also annotated with the stereotype ≪**responseTime**≫ to be implemented efficiently. As is the case with performance, we describe the necessary assumptions and facts in a domain knowledge diagram for security. We assume that each user keeps the text message to be sent, the current chat session and the own private key confidential. Generation of key pairs and secure distribution of the public key is covered by other problem diagrams. For reasons of space, we do not show the domain knowledge diagram for security.

By now we have considered the parts of the problem *Communicate* that belong to the client. Now we specify the quality problem diagram for the part that belongs to the server, namely *Forward*. This problem requires to satisfy both confidentiality and response time requirements as shown in Fig. 4. The sent text message arrives at the server in encrypted form. It will directly be forwarded to the receivers. So the confidentiality of the text messages on the server is preserved, and we do not need to take any further measures concerning the confidentiality requirement.

When many users are connected to the server concurrently and communicate, the server can be overloaded. This might lead to longer response times than permitted. We previously introduced load balancing and master-worker as mechanisms or patterns that solve performance problems. We will consider both of them for the *Forward* problem.

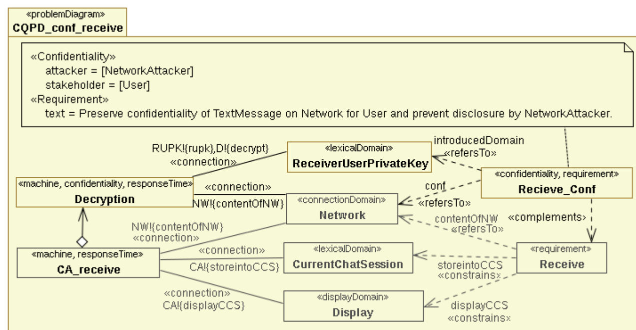

Figure 8: Concretized Quality Problem Diagram for the quality requirement ***Receive_Conf***

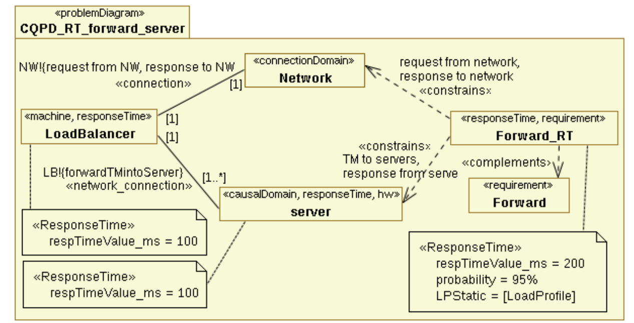

Figure 9: Concretized Quality Problem Diagram for the quality requirement ***Forward_RT***

In Fig. 9, we specify the concretized quality problem diagram that uses the load balancing mechanism to solve the response time problem. We introduce a new machine *LoadBalancer* that distributes the load from the network across several server components, each of which contains one machine for solving the *Forward* problem. To satisfy the response time requirement for the problem *Communicate*, a text message should be forwarded within 200 ms. The load balancer should conduct its task within 100 ms. This means another 100 ms for all servers to forward a text message. A suitable proportion between the number of servers and the time each server needs should be found.

Figure 10 shows a different concretized quality problem diagram. It is the result of applying the master-worker pattern to the *Forward* problem. We introduce a new machine domain *Master* that distributes the task received from the network to several *CA_forward* machines. In contrast to the previous solution, this solution consists of a single server, which contains a master and several machines that provide the forward functionality. The master machine is required to satisfy its task in less than 100 ms. The same applies to the several *CA_forward* machines.

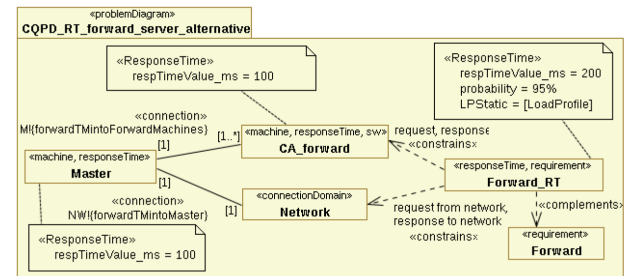

Figure 10: Alternative concretized Quality Problem Diagram for the quality requirement ***Forward_RT***

5.5 Implementable Architecture

The purpose of this step is to derive an architecture, which is implementable and achieves the required level of performance and security. We make use of problem diagrams annotated with quality requirements and concretized quality problem diagrams.

5.5.1 Allocate Components

The implementable architecture consists of one component for the overall machine (in our case, *chat application*), with stereotypes ≪**machine**≫ and ≪**implementable_architecture**≫, see Fig. 11. In the case of a standalone application, we skip this step and carry on with the next step. In

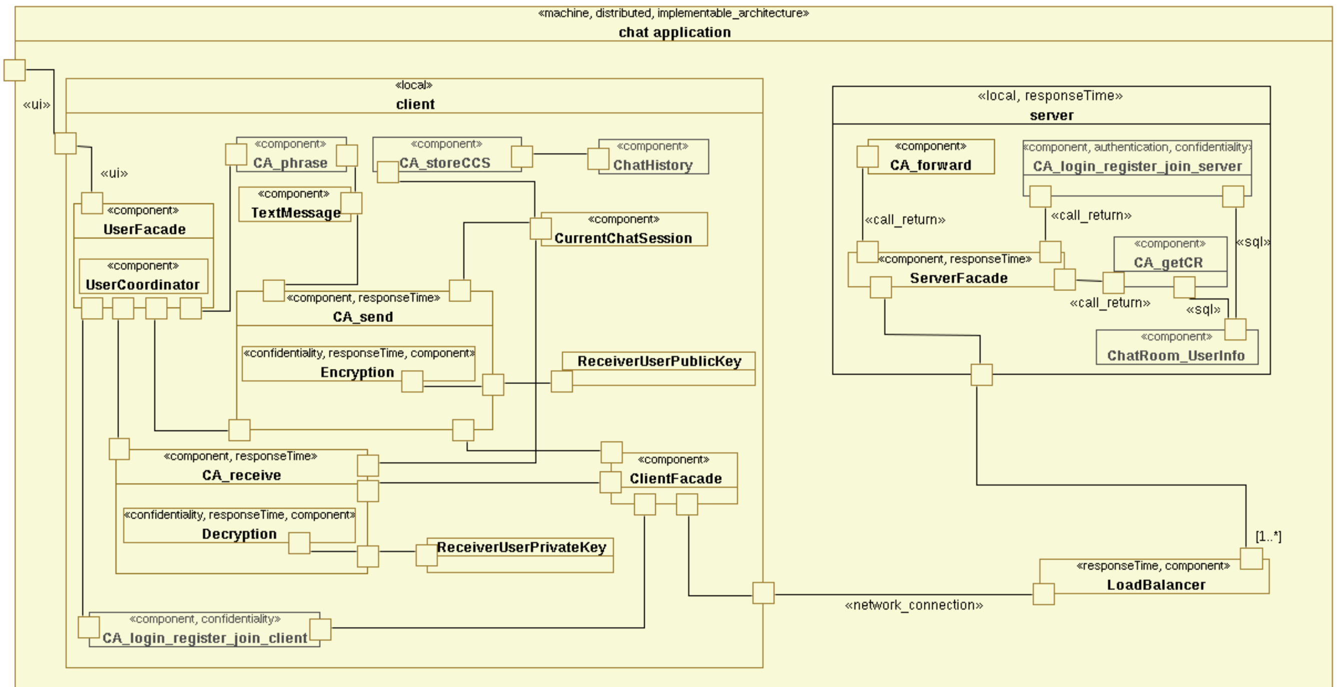

Figure 11: Implementable Architecture

the case of a distributed architecture, we add the stereotype ≪`distributed`≫ to the architecture component. Inside the component for the whole machine, there are two components representing client and server, respectively, annotated with stereotype ≪`local`≫. Now we make use of the split problem diagrams we described in Sect. 5.4.1. Each submachine in the split problem diagrams belongs to a component in the client or in the server according to functionality of that submachine. The stereotype ≪`component`≫ is added to all components. By taking the example of the chat application, the submachines *CA_send* and *CA_receive* belong to the client component, because they are related to the user. The submachine *CA_forward* belongs to the server, because it has the functionality of forwarding text messages to all receivers.

5.5.2 *Merge Components*

Related components that realize a similar functionality and contain at least one similar domain in their problem diagrams can be merged to one component. In the chat application, we could merge the *login* and *register* components of the initial architecture shown in Fig. 2. In both problems, the user enters user data, which should be transmitted to the server, the server connects to a database to process the user data, and in the end sends some feedback to the user. In general, the decision about the merging of components should be taken by an experienced architect.

5.5.3 *Apply Design Patterns*

We introduce a *Facade* component [10], if several internal components are connected to one external interface in the initial architecture. As a *Facade* component, we introduce the *UserFacade* component that realizes the *Single Access Point* pattern described in Sect. 3.3 as a single entry and exit point into the system. By providing only one access point, it is much easier to protect the system. It can be implemented as an input screen that receives the user inputs and forwards it to the server to be verified. Additionally, we provide the *ClientFacade* component on the client side in order to prevent that each single component communicates with the server directly. On the server side, we introduce the *ServerFacade* component. Adding a *Facade* component causes only one method invocation more and hence does not impair the performance of the software. If interaction restrictions have to be taken into account, i.e., actions have to happen in a certain order, we have to add one or more *Coordinator* components. In our example of the chat application, the user must first authenticate before taking any action. Therefore, we introduce a *UserCoordinator*. To obtain a clear structure of the software architecture, we integrate the *UserCoordinator* in the *UserFacade*. The implementable architecture after applying these patterns is shown in Fig. 11.

5.5.4 *Apply Quality Mechanisms and Patterns*

Now we make use of the mechanisms and patterns we introduced in Sect. 3.2 and 3.3 and the concretized quality problem diagrams we specified in Sect. 5.4.3.

Considering solution domains (e.g., *Encryption*) in concretized quality problem diagrams provides a seamless integration of quality mechanisms into software architecture. We extend the existing architecture with new domains we obtain from the concretized quality problem diagrams. All new domains are annotated with stereotype ≪`component`≫. Additionally to this stereotype, the new domains retain their ≪`responseTime`≫ and ≪`confidentiality`≫ stereotypes from the concretized quality problem diagrams that serve as hints for the technical realization. For example, the *Encryption* machine is integrated in the *CA_send* component and annotated with stereotypes ≪`responseTime`≫ and ≪`confidentiality`≫. The domain *ReceiverUserPublicKey* is connected to the *Encryption* machine. The *Decryption* machine

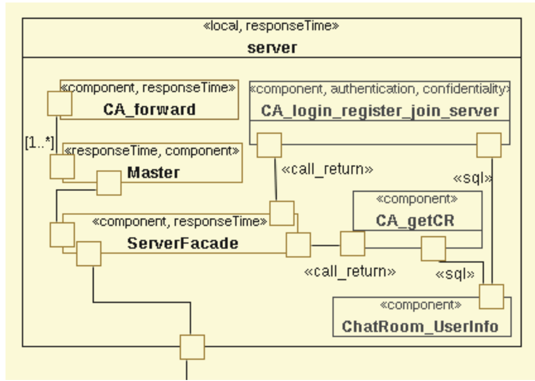

Figure 12: Alternative Implementable Architecture

is integrated in the *CA_receive* component and annotated with stereotypes ≪`responseTime`≫ and ≪`confidentiality`≫. The domain *ReceiverUserPrivateKey* is connected to the *Decryption* machine. The *LoadBalancer* is placed before the servers and is annotated with stereotype ≪`responeTime`≫. Its port multiplicity [1..*] (see Fig. 11) means that the load balancer component can be connected with several server components.

An alternative architecture would contain a *Master* component with several *CA_forward* components. The port multiplicity at the master component means that this component is connected with several *CA_forward* components. The *CA_forward* components are annotated with stereotype ≪`responseTime`≫ inside a single server, instead of a *LoadBalancer* component with several servers, see Fig. 12.

6. RELATED WORK

Consideration of software quality during the software development process, specifically in the requirement analysis phase, is still a challenging research problem. There are approaches that deal with only one type of quality requirement, e.g., security.

An approach to transform security requirements to design is provided by Mouratidis and Jürjens [17]. It starts with the goal-oriented security requirements engineering approach Secure Tropos [16], and connects it with a model-based security engineering approach, namely UMLsec [15].

Yskout et al. [27] present a semi-automated approach to support the transition from security requirements to architecture. They focus on delegation, authorization and auditing as security requirements. They presuppose an architecture that fulfills the functional requirements, and they apply security solutions to the functional architecture by transforming security requirements.

Schmidt and Wentzlaff [20] develop architectures from requirements based on the problem frame approach, taking into account usability and security. By way of an example, they demonstrate how to balance security and usability requirements.

Heyman et al. [13] present the security twin peaks model, an elaboration of the twin peaks model proposed by Nuseibeh [18]. This model addresses the co-development of secure software architectures and security requirements. In our method, we proceed in a similar way, interleaving requirements analysis and architectural design.

Attribute Driven Design (ADD) [24, 4] is a method to design a conceptual architecture. It focuses on the high-level design of an architecture, and hence does not support detailed design. ADD deals with achieving quality attributes including security and performance through architecture mechanisms. Identifying the mechanisms partially relies on the architect's expertise.

Q-ImPrESS [5] is a project that focuses on the generation and evaluation of architectures according to quality properties, in particular performance. The phases design and implementation of the software development process are particularly in focus. In contrast to our contribution, it does not use requirements descriptions as a starting point.

The notation and evaluation of performance attributes of an architecture is the focus of the component model Palladio [6], which is also included in the project Q-ImPrESS. In Palladio, a set of notations, concepts and a tool are provided, which allow its users to model and simulate architectures for performance evaluation. The tool could be used for simulating and thus evaluating software architecture performance. The concepts and the included tool, however, cannot be used to evaluate an architecture's security.

There exist some UML profiles that deal with quality requirements. They focus on modeling and analysing quality requirements. A well-known UML profile is MARTE [22] that allows the annotation of embedded and real-time systems with performance attributes. The UML profile for schedulability, performance and time [23] is another profile that provides a means for analysis of performance-related aspects for real-time systems. These profiles are strongly designed to introduce performance attributes into UML models. But they do not support other types of attributes. In contrast, using our UML profile is not limited to specific kinds of systems, such as real-time systems.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a detailed, UML-based and tool-supported method to derive software architectures from requirements documents, thereby taking quality requirements into account. Our method addresses all the problems we identified in the introduction:

We provide means to specify quality requirements thoroughly. Problem diagrams are a means to describe software development problems precisely by referring to and constraining different domains that are interconnected with one another and with the machine to be developed. Such diagrams can be annotated with precise quality requirements. We have defined appropriate stereotypes with corresponding attributes.

Seamless transition from requirements analysis to architectural design. The two phases are not separated, but intertwined. An architectural decision drives the revision of problem descriptions, and concretized problem descriptions lead directly to architectural components and connections.

Explicit consideration of quality requirements. Our method builds on established approaches to achieve quality properties, such as encryption or load balancing. The application of these mechanisms or patterns is directly visible in the software architecture.

In the future, we will elaborate our method further. In the past, we had no difficulties to express quality requirements as complements to functional requirements. However, it is often claimed that quality requirements are crosscutting and cannot be attached to a concrete functionality. We intend to investigate this issue.

In the present work, we have not investigated possible conflicts between different quality requirements. We have just noted that e.g., encryption takes time and that we therefore should pay attention to performance requirements when introducing encryption mechanisms. We strive for a more systematic treatment of conflicting quality requirements.

In this paper, we have concentrated on structural descriptions of software architectures. In the future, we will extend our method to also support deriving behavioral descriptions for the developed architectures and automatically checking their coherence with the structural descriptions.

Acknowledgments.
We would like to thank Holger Schmidt for his detailed comments to this paper.

8. REFERENCES

[1] Eclipse - An Open Development Platform, Feb 2011. http://www.eclipse.org/.

[2] Eclipse Modeling Framework Project (EMF), Feb 2011. http://www.eclipse.org/modeling/emf/.

[3] Papyrus UML Modelling Tool, Feb 2011. http://www.papyrusuml.org/.

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

[5] S. Becker, S. Dešić, J. Doppelhamer, D. Huljenić, H. Koziolek, E. Kruse, M. Masetti, W. Safonov, I. Skuliber, J. Stammel, M. Trifu, J. Tysiak, and R. Weiss. Q-ImPrESS Project Deliverable D1.1 – Requirements document. final version, Q-ImPrESS Consortium, 2009.

[6] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3 – 22, 2009. http://dx.doi.org/10.1016/j.jss.2008.03.066.

[7] C. Choppy, D. Hatebur, and M. Heisel. Systematic architectural design based on problem patterns. In P. Avgeriou, J. Grundy, J. Hall, P. Lago, and I. Mistrik, editors, *Relating Software Requirements and Architectures*, chapter 7. Springer, 2011. To appear.

[8] I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff. A Systematic Account of Problem Frames. In *Proc. of the European Conf. on Pattern Languages of Programs (EuroPLoP)*, pages 749–767. Universitätsverlag Konstanz, 2008.

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

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Wiley & Sons, Boston, USA, 1995.

[11] D. Hatebur and M. Heisel. A foundation for requirements analysis of dependable software. In B. Buth, G. Rabe, and T. Seyfarth, editors, *Proc. of the Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 5775, pages 311–325. Springer, 2009.

[12] D. Hatebur and M. Heisel. Making Pattern- and Model-Based Software Development More Rigorous. In J. S. Dong and H. Zhu, editors, *Proc. of 12th Int. Conf. on Formal Engineering Methods (ICFEM)*, LNCS 6447, pages 253–269. Springer, 2010.

[13] T. Heyman, K. Yskout, R. Scandariato, H. Schmidt, and Y. Yu. The security twin peaks. In *Proc. of the Int. Symposium on Engineering Secure Software and Systems (ESSoS)*, LNCS 6542, pages 167–180. Springer, 2011.

[14] M. Jackson. *Problem Frames. Analyzing and structuring software development problems.* Addison-Wesley, 2001.

[15] J. Jürjens. *Secure Systems Development with UML.* Springer, 2004.

[16] H. Mouratidis. *A Security Oriented Approach in the Development of Multiagent Systems: Applied to the Management of the Health and Social Care Needs of Older People in England*. PhD thesis, University of Sheffield, U.K., 2004.

[17] H. Mouratidis and J. Jürjens. From goal-driven security requirements engineering to secure design. *Int. J. Intell. Syst.*, 25:813–840, 2010.

[18] B. Nuseibeh. Weaving Together Requirements and Architectures. *Computer*, 34:115–117, 2001.

[19] D. G. Rosado, E. Fernandez-Medina, M. Piattini, and C. Gutierrez. A study of security architectural patterns. In *Proc. of the 1st Int. Conf. on Availability, Reliability and Security*, pages 358–365, Washington, DC, USA, 2006. IEEE Computer Society.

[20] H. Schmidt and I. Wentzlaff. Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In *Proc. of the European Workshop on Software Architectures (EWSA)*, volume 4344, pages 189–203. Springer, 2006.

[21] "UML Revision Task Force". *OMG Unified Modeling Language (UML), Superstructure.* http://www.omg.org/spec/UML/2.3/Superstructure/PDF.

[22] "UML Revision Task Force". *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems.* http://www.omg.org/spec/MARTE/1.0/PDF.

[23] "UML Revision Task Force". *UML Profile for Schedulability, Performance, and Time Specification.* http://www.omg.org/spec/SPTP/1.0/PDF.

[24] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-Driven Design (ADD). Version 2.0, Software Engineering Institute, 2006.

[25] J. Yoder and J. Barcalow. Architectural Patterns for Enabling Application Security. In *Proc. of the 4th Conf. on Pattern Languages of Programming (Plop)*, 1998.

[26] K. Yskout, T. Heyman, R. Scandariato, and W. Joosen. A system of security patterns. CW Reports CW469, K.U.Leuven, 2006.

[27] K. Yskout, R. Scandariato, B. D. Win, and W. Joosen. Transforming security requirements into architecture. In *Proc. of the 3rd Int. Conf. on Availability, Reliability and Security*, pages 1421–1428, Washington, DC, USA, 2008. IEEE Computer Society.