A Method to Derive Software Architectures from Quality Requirements

Azadeh Alebrahim, Denis Hatebur, Maritta Heisel
Software Engineering
Department of Computer Science and Applied Cognitive Science
University Duisburg-Essen, Germany
azadeh.alebrahim, denis.hatebur, maritta.heisel@uni-duisburg-essen.de

***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 which quality requirements are reflected explicitly. For requirements analysis, we use an enhancement of the problem frame approach, where software development problems are represented by *problem diagrams*. 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. Then, we incorporate quality requirements into that architecture by using security or performance patterns or mechanisms. The method is tool-supported, which allows developers to check semantic integrity conditions in the different models.**

***Keywords*-Quality-driven design; quality requirements; software architecture; performance, security** [1]

I. Introduction

The treatment of quality (or non-functional) requirements in software development is not yet as well mastered as the treatment of functional requirements. 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 must be integrated in such a way that the knowledge gained in the requirements engineering phase is 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.

In this paper, we want to contribute to improve this 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, whereas for performance requirements this is hardly the case. Therefore, these two kinds of requirements are appropriate representatives 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 [22] 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* (available under http://www.uml4pf.org), 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 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. The so enhanced problem descriptions form the starting point for architectural design. In a first step, 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, applying appropriate security or performance patterns or mechanims 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. As compared to the short paper [4], this paper presents a further elaborated method.

The rest of the paper is organized as follows. In Sect. II, we introduce the case study that serves as a running example. We then present the basics on which our method builds in Sect. III. In Sect. IV, we present the UML profile we defined to carry over the problem frame approach to UML, as well as the tool UML4PF. Section V is devoted to describing our method in detail. Related work is discussed in Sect. VI, and conclusions and future work are given in Sect. VII.

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

II. Case Study

We illustrate our approach by a chat application, which allows a text-message-based communication via private I/O devices. Users should be able to communicate with other chat participants in a same chat room.

We focus on the functional *Communicate* requirement with the description "*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*" and its corresponding quality requirements *Response Time* with the description "*The sent text message should be shown on the receiver's display in 1500 ms maximum*" and *Confidentiality* with the description "*Text messages should be transmitted in a confidential way*".

Note that in order to specify performance requirements properly, more details have to be given. We use the MARTE profile [23] for this purpose, which we describe in more detail in Sect. V-B. Besides the confidentiality of text messages, their *integrity* is also important. For reasons of space, however, we do not address integrity in this paper.

III. Basic Concepts

Our method relies on a requirements engineering process based on problem frames (Sect. III-A) and uses established patterns and mechanisms to meet performance and security requirements (Sect. III-B).

A. 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* (i.e., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements. Requirements analysis 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, 3, and 4 show problem diagrams in UML notation.

B. Mechanisms and Patterns for Performance and Security

To satisfy performance and security requirements, different mechanisms, also called patterns, are available [9], [20], [26]. We will use the following patterns and mechanisms:

1) 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 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.

2) Master-Worker: makes it possible to serve requests in parallel, similarly to load balancing. In contrast to load balancing that uses hardware components, the master-worker pattern provides a software solution. It consists of a software component 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, which manage the smaller tasks.

3) 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. An example for this pattern is to create a login screen that collects user information such as user name and password.

4) 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. In asymmetric encryption, there is no key distribution problem, but a trusted third party is needed that issues the key pairs.

IV. 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, 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.

A. UML Profile for Problem Frames

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. Jackson distinguishes the domain types biddable domains (represented by

the stereotype ≪BiddableDomain≫) that are usually people, causal domains (≪CausalDomain≫) that comply with some physical laws, and lexical domains (≪LexicalDomain≫) that are data representations. To describe the problem context, a *connection domain* (≪ConnectionDomain≫) between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Examples are 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 e.g. be events, operation calls or messages. They are observable by at least two domains, but controlled by only one domain, as indicated by "!". In Fig. 1 the notation $U!\{sendTM\}$ (between *CA_communicate* and *User*) means that the phenomenon *sendTM* is controlled by the domain *User*. 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 or quality 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≫. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype ≪refersTo≫.

The problem diagram in Fig. 1 describes the requirement *Communicate* in more detail. It describes that the *CA_communicate* machine can show to the user the *CurrentChatSession* on its *Display* (*CAC!*$\{$*displayCCS*$\}$). The requirement constrains the *CurrentChatSession* and the *Display*. It refers to the *User* and the *TextMessage*.

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*, which is represented by *domain knowledge diagrams*. Domain knowledge consists of *assumptions* and *facts*. Assumptions usually describe required user behavior, whereas facts describe fixed properties of the problem environment, regardless of how the machine is built.

B. Annotating Problem Descriptions with Quality Requirements

To consider quality requirements, we extended our UML profile for problem frames to complement functional requirements with dependability requirements [13]. Classes with stereotypes such as ≪confidentiality≫, ≪integrity≫ and corresponding attributes such as *attacker*, *stakeholder* address security requirements. The dependency from a quality requirement to a requirement is expressed with the stereotype ≪complements≫ (Fig. 1).

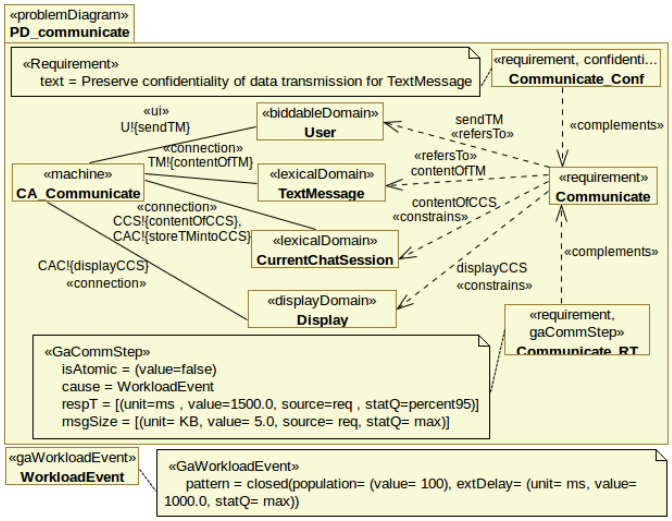

Figure 1. Problem diagram for the requirement *Communicate*

To provide support for annotating problem descriptions with performance requirements, we use the UML profile MARTE (Modeling and Analysis of Real-time and Embedded Systems) [23]. We focus on the GQAM package (Generic Quantitative Analysis Modeling). To define workload and behavior concerns we make use of the GQAM_Workload package by instantiating the appropriate attributes of this package. Each *BehaviorScenario* is composed of *Steps*, each of which can be refined as another *BehaviorScenario*. A behavior scenario is triggered by the *WorkloadEvent*, which may be generated by an *ArrivalPattern* such as the *ClosedPattern* that allows us to model a number of concurrent users and a think time (the time the user waits between two requests) by instantiating the attributes *population* and *extDelay*. We define a *BehaviorScenario* composed of one *Step* for the requirement *Communicate_RT* (see Sect. V-B), which is refined in three *BehaviorScenario* instances, each of which is composed of a single *Step*. The *Step* instances represent the requirements *Send_RT*, *Forward_RT* and *Receive_RT* that stand in the precedence relationship *Sequence* (see Fig. 15.3, p. 289 of the MARTE specification [23]).

C. Tool Support

We provide tool support for software development based on problem frames. Our tool, called UML4PF, 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 [21]. 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 have set up a number of OCL constraints. Based on the model information, UML4PF can automatically detect semantic errors in the model by evaluating the constraints. We can also validate that the artifacts of later development steps are consistent with the requirements engineering diagrams. For more details, see [12], [7].

V. Deriving Quality-based Architectures

We first give an overview of our method illustrated in Fig. 2, before applying it to the chat described in Sect. II.

We first decompose the overall problem into subproblems (*Problem Diagrams*), each of which is related to one or more functional requirements. Then we annotate each subproblem by complementing functional requirements with related quality requirements (*Quality Problem Diagrams*). In the next step we take a design decision concerning the kind of distribution of the software architecture (*Choose Design Alternative*). Then we go back to the requirements descriptions and regarding the design decision split the problem diagrams accordingly (*Split Problem Diagrams*). Analogously to splitting the problem diagrams and so splitting the functional requirements, we also have to split the corresponding quality requirements (*Split Quality Requirements*). Then we set up an initial architecture by mapping each machine domain in a problem diagram to a component (*Initial Architecture*). After that we elaborate the problem diagrams annotated with quality requirements by introducing domains reflecting specific solution approaches (*Concretized Quality Problem Diagrams*). In the next step we derive an architecture, which is implementable and achieves the required level of performance and security (*Implementable Architecture*). We make use of problem diagrams annotated with quality requirements and concretized quality problem diagrams. To obtain the implementable architecture, we first merge related components (*Merge Components*). Next, we apply appropriate design patterns (*Apply Design Patterns*). Finally we make use of mechanisms and patterns and the concretized quality problem diagrams (*Apply Quality Mechanisms/Patterns*).

We now present these steps in detail.

A. Problem Diagrams

As described in Sect. III-A, 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.[2] For reasons of space we do not show the context diagram for the chat application and continue with the problem decomposition step. We decompose the overall 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. II. 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* and constrains the domains *CurrentChatSession* and *Display*.

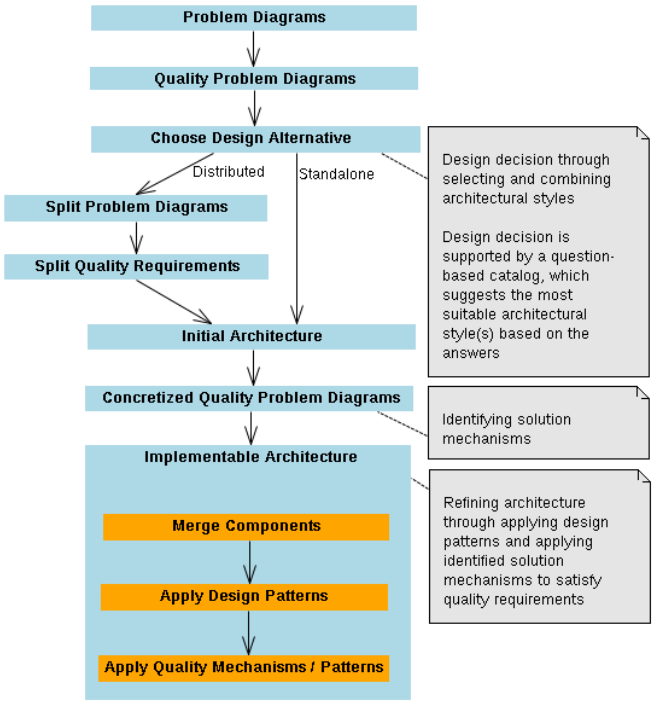

Figure 2. Method for quality-based derivation of software architectures

B. Quality Problem Diagrams

In this step, we address quality requirements by annotating problem diagrams with suitable stereotypes. In Sect. II, we defined one security and one performance requirement related to the functional requirement *Communicate*. That requirement is complemented by the confidentiality requirement *Communicate_conf* that requires confidentiality of data transmission for *TextMessage* and the response time requirement *Communicate_RT* representing one *BehaviorScenario* composed of one *Step* described with the stereotype ≪`gaCommStep`≫ in Fig. 1. The response time requirement is modeled by instantiating the relevant attributes of the *Step* class in the MARTE GQAM_Workload package described

[2]Woods and Rozanski also suggest a new view point, called *context view* [25], in addition to their 6 view points to guide the architectural desgin to define the environment of a system [18].

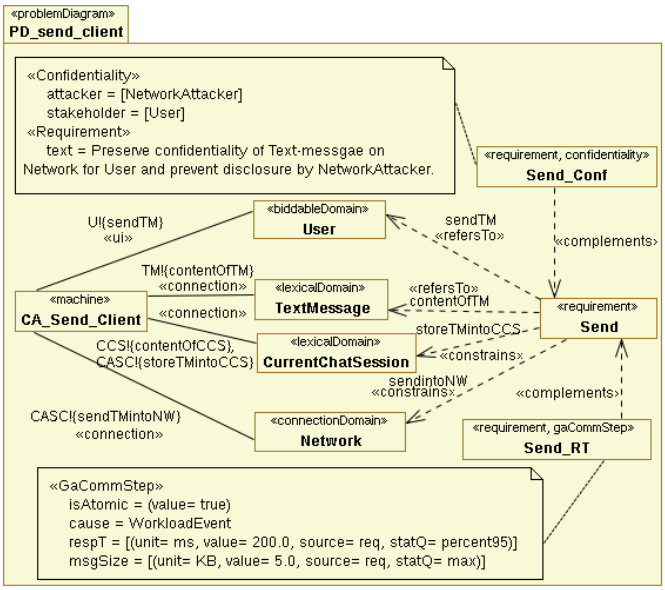

Figure 3. Problem diagram for *Send*, annotated with quality requirements

in Sect. IV-B. The *respT* attribute states that the required response time for sending text messages should be 1500 ms maximum. The *cause* attribute represents the triggering event, which is in our case a *ClosedPattern* with 100 concurrent users (*population*), each of which needs a think time of 1000 ms (*extDelay*). The *msgSize* attribute states that the sending text messages should be 5 KB maximum.

C. Choose Design Alternative

We now need to take a design decision concerning the kind of distribution of the software to be developed, e.g., client-server, peer-to-peer, or standalone. This decision is either taken by the stakeholder or by the software architect. In this paper, we do not discuss how this decision is taken.
For standalone applications, we can skip two steps and continue with the step "Initial Architecture". In the following, we describe how to proceed for a client-server architecture in more detail.

D. Split Problem Diagrams

After having chosen a client-server architecture for the chat application, we go back to the requirements descriptions 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. IV-A). For 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 (not shown). For each of these three subproblems, we introduced the connection domain *Network* to achieve the distribution.

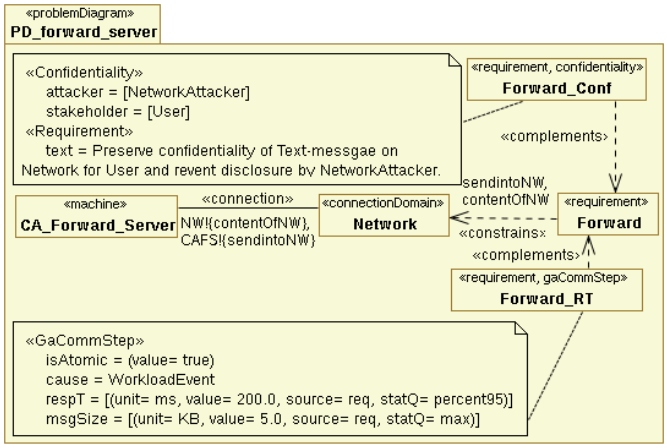

Figure 4. Problem diagram for *Forward*, annotated with quality requirements

E. 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. This leaves 800 ms to transmit data from the client to the server and back. We cannot meet performance and specifically response time requirements, if we have no knowledge about the real circumstances in the environment. Therefore we specify knowledge (see Sect. IV-A) about the network and the computational power of clients and server in the domain knowledge diagram for performance depicted in Fig. 5. It contains specific knowledge about client and server, e.g., the number of processor cores, processor speed and memory.
To fulfill the confidentiality requirement for the problem *PD_communicate* (Fig. 1), we require confidentiality for each subproblem, thus annotating each subproblem with a corresponding refined confidentiality requirement. It contains a *stakeholder* who 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. Attributes of an attacker are specified more precisely in [13]. The refined confidentiality requirements for the *Send* and *Forward* subproblems are shown in Figs. 3 and 4.

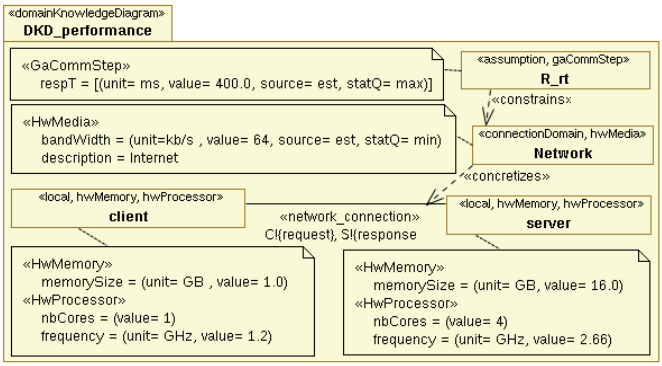

Figure 5. Domain knowledge diagram for performance

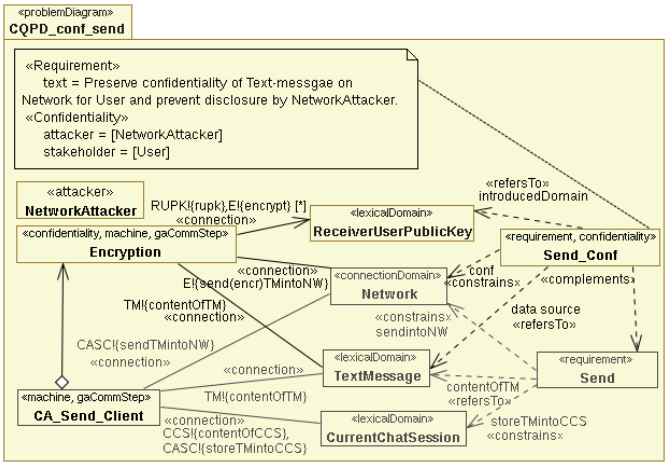

Figure 6. Concretized quality problem diagram for quality requirement *Send_Conf*

F. Initial Architecture

The initial architecture consists of one component for the overall machine (e.g., *chat application*), with stereotypes ≪machine≫ and ≪initial_architecture≫. For a distributed architecture, we add the stereotype ≪distributed≫ to the architecture component. For client-server architectures, there are two components representing client and server, respectively, inside the overall machine. Then we make use of the split problem diagrams we described in Sect. V-D. Each submachine in the split problem diagrams becomes a component either in the client or in the server. In the chat application, the submachines *CA_send* and *CA_receive* belong to the client component, whereas *CA_forward* belongs to the server.

G. Concretized Quality Problem Diagrams

The goal of this step is to find solution approaches, e.g., the ones given in Sect. III-B, to prepare for solving the given security and performance problems. We elaborate the problem diagrams annotated with quality requirements by introducing domains reflecting specific solution approaches. We call the elaborated problem diagrams containing solution approaches *concretized quality problem diagrams*.

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 performance requirement states that sending a text message should be performed within 200 ms. However, this requirement cannot be achieved by architectural means. Instead, it must be taken care of in the implementation. In such a case, 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 ≪gaCommStep≫ (see Fig. 6).

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. 6 that uses an encryption mechanism to solve the problem. We decide to first apply a symmetric and then an asymmetric encryption mechanism[3]. To solve the problem using these mechanisms, we 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. Therefore, 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 ≪gaCommStep≫ to point implementers towards efficient algorithms and implementations. We also need to introduce new components on the receiver side, namely a new machine *Decryption* and a component *ReceiverUserPrivateKey*.

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.

To address the performance requirements for *Forward*, we consider two alternatives. Figure 7 shows the concretized quality problem diagram that uses the load balancing mechanism to solve the response time problem. A new machine *LoadBalancer* distributes the load from the network across several server components, each of which contains one

[3]Other solutions are possible, but are not discussed here.

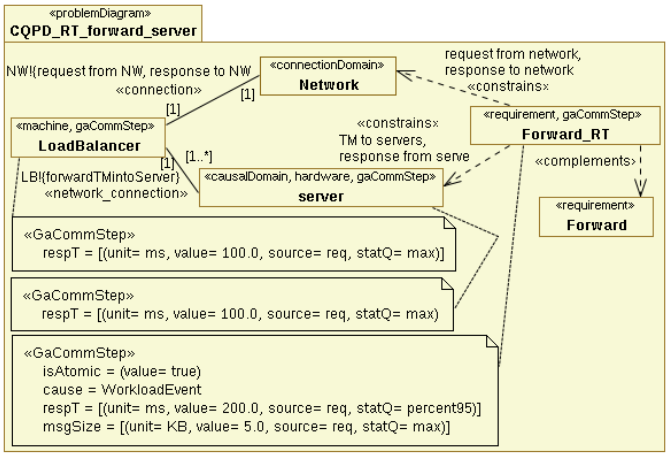

Figure 7. Concretized quality problem diagram for quality requirement

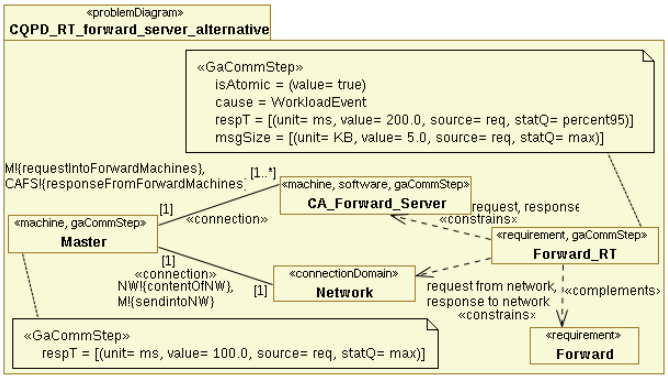

Figure 8. Alternative concretized quality problem diagram for quality requirement *Forward_RT*

machine for solving the *Forward* problem.

Figure 8 shows the concretized quality problem diagram that results when applying the master-worker pattern to the *Forward* problem. A new machine domain *Master* 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.

H. Implementable Architecture

The purpose of this step is to derive an architecture, which is implementable and fulfills the performance and security requirements. We proceed in three steps.

1) 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 components for logging in and registering users (not discussed previously). In general, the decision about the merging of components should be taken by an experienced architect.

2) 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. III-B (see Fig. 9). Additionally, we provide the *ClientFacade* component on the client side in order to prevent each single component from communicating with the server directly. On the server side, we introduce the *ServerFacade* component. Adding a *Facade* component causes only one additional method invocation 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, 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*.

3) Apply Quality Mechanisms and Patterns: Now we make use of the mechanisms and patterns described in Sect. III-B and the concretized quality problem diagrams specified in Sect. V-G. 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 ≪gaCommStep≫ and ≪confidentiality≫ stereotypes.

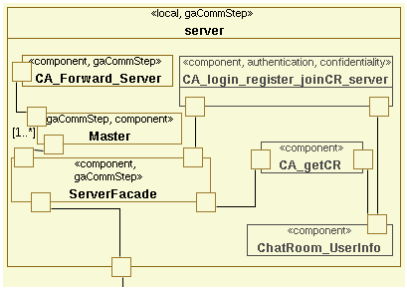

Figure 10. Alternative implementable architecture

For example, the *Encryption* machine is integrated in the *CA_send* component and annotated with stereotypes ≪gaCommStep≫ and ≪confidentiality≫. The domain *ReceiverUserPublicKey* is connected to the *Encryption* machine. The *Decryption* machine is integrated in the *CA_receive* component, and the domain *ReceiverUserPrivateKey* is connected to the *Decryption* machine. The *LoadBalancer* is placed before the servers. Its port multiplicity [1..*] (see Fig. 9) means that it can be connected with several server components.

An alternative architecture would contain a *Master* component with several *CA_forward* components. The *CA_forward* components are inside a single server (see Fig. 10), instead of a *LoadBalancer* component with several servers.

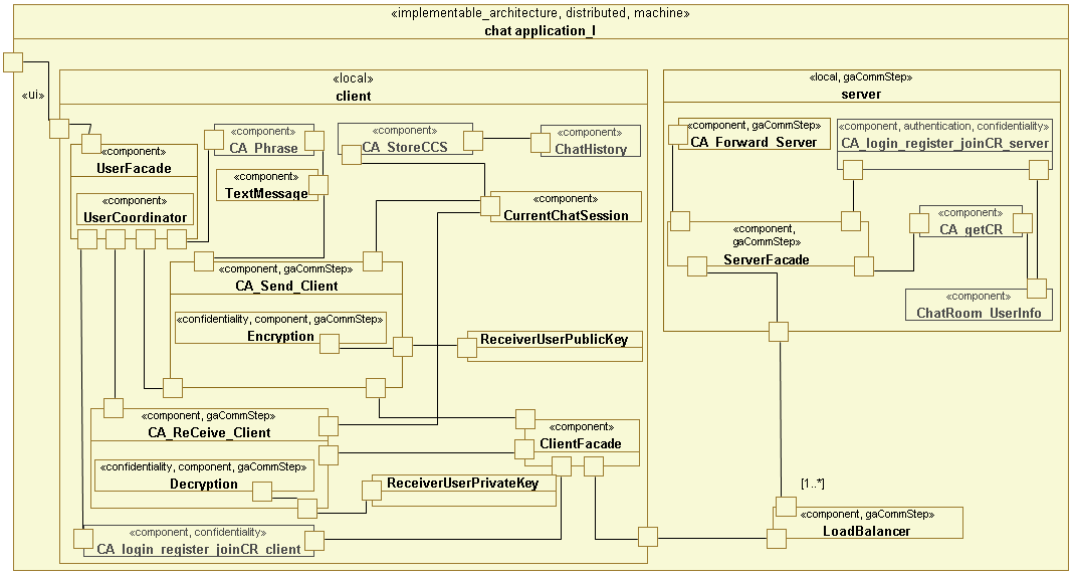

Figure 9. Implementable architecture

VI. Related Work

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]. UMLsec is a UML profile for representing security properties in UML diagrams. It does not provide support for the analysis phase of the software development process.

Schmidt and Wentzlaff [19] develop architectures from requirements based on the problem frame approach, taking into account usability and security. They show how to balance security and usability requirements.

Hall et al. [11] present an extension of the problem frames approach allowing design concerns. They relate software architectures to requirements in the problem domain. However they only consider functional requirements and not quality requirements.

Attribute Driven Design (ADD) [24] 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. Identifying mechanisms to achieve quality attributes 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.

VII. Conclusion

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:

Thorough specification of quality requirements. Problem diagrams are a means to describe software development problems precisely. Such diagrams can be annotated with elaborated quality requirements, based on UML profiles.

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.

Our method builds on established techniques such as problem frames, security and performance patterns. Its novelty lies in the fact that the different approaches are integrated and intertwined explicitly by an underlying methodology and a common notation. The notation as well as the methodology are open and can be developed further to enhance the power and breadth of the approach. Furthermore, tool support

helps software engineers to explore different architectural alternatives and provides valuable feedback by automatic checking of semantic integrity conditions.

A possible limitation of our approach might be the fact that quality requirements are attached to functional requirements. This might make it difficult to treat cross-cutting concerns. We intend to investigate this issue.

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

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] A. Alebrahim, D. Hatebur, and M. Heisel. Towards systematic integration of quality requirements into software architecture. In I. Crnkovic and V. Gruhn, editors, *Proc. ECSA 2011*, LNCS 6903, pages 17–25. Springer Verlag, 2011.

[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 9, pages 133–160. Springer, 2011.

[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] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *Proceedings of IEEE International Requirements Engineering Conference (RE'02*, pages 137–144. IEEE Computer Society Press, 2002.

[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*, LNCS 6447, pages 253–269. Springer, 2010.

[13] D. Hatebur and M. Heisel. A UML profile for requirements analysis of dependable software. In E. Schoitsch, editor, *Proc. of the Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 6351, pages 317–331. Springer, 2010.

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

[15] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[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] N. Rozanski and E. Woods. *Software Systems Architecture*. Addison-Wesely, Upper Saddle River, 2005.

[19] H. Schmidt and I. Wentzlaff. Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In *Proc. EWSA*, LNCS 4344, pages 189–203. Springer, 2006.

[20] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley & Sons, 2005.

[21] "UML Revision Task Force". *Object Constraint Language Specification*. http://www.omg.org/spec/OCL/2.0/PDF.

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

[23] "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.

[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] E. Woods and N. Rozanski. The system context architectural viewpoint. In *WICSA/ECSA*. IEEE, 2009.

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