Deriving Quality-based Architecture Alternatives with Patterns*

Marco Konersmann, Azadeh Alebrahim, Maritta Heisel, Michael Goedicke, Benjamin Kersten
paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Germany
{marco.konersmann, azadeh.alebrahim, maritta.heisel, michael.goedicke, benjamin.kersten}@paluno.uni-due.de

Abstract: We propose in this paper an iterative method composed of three steps to derive architecture alternatives from quality requirements using a catalogue of patterns and styles. The solution candidates are chosen by answering a set of questions which reflects the requirements. We instantiate then the solution candidates using a UML-based enhancement of the problem frame approach. To ensure that the instantiated architectures fulfill the quality requirements, we evaluate them in the next step. A desired refinement of the software architectures is then achieved by iterating over the described steps.

1 Introduction

The development of software architecture is a challenging task, even when the requirements of a software system are clear. For building upon common knowledge and best practices, the use of catalogues containing architectural patterns and styles (e.g. [TMD09]) has shown to be valuable. These catalogues are used in architectural design methods, that aim to give guidance for deriving architectures from requirements (e.g. [BCK03, HNS99]). In these methods, catalogues are used as a reference to find solutions for an architectural problem by choosing appropriate patterns and styles (called *solution candidates* in this document) from the catalogue. Trying to find appropriate solution candidates in a catalogue is, however, not trivial. Usually more than one candidate is appropriate, but they differ in their quality attributes. The existing approaches are imprecise or do not provide any aid for finding good candidates from catalogues [BR10]. Sequentially evaluating the candidates of a catalogue is not efficient.

In this paper we propose an iterative method to derive alternative architectures that differ in their quality attributes. The method integrates our approaches to this problem in [AHH11] and [MKG11]. We first obtain alternative solution candidates proposed from catalogues by relating questions to these candidates. Answers to these questions rate the solution candidates with respect to the requirements of the system to develop. In the second step, we derive architecture alternatives from the best-rated candidates. For the derivation we

*The work reported herein was funded by the German Research Foundation under the grants no. GO774/5-1 and HE3322/4-1.

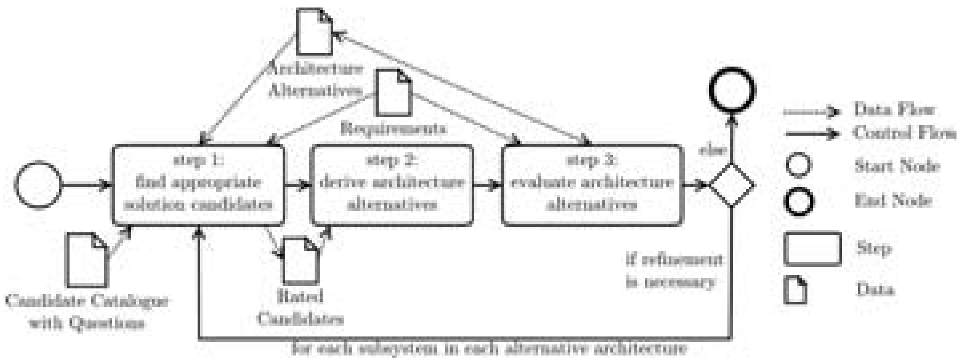
Architecture Alternatives
Requirements
step 1: find appropriate solution candidates
step 2: derive architecture alternatives
step 3: evaluate architecture alternatives
else
if refinement is necessary
Candidate Catalogue with Questions
Rated Candidates
for each subsystem in each alternative architecture
Data Flow
Control Flow
Start Node
End Node
Step
Data

Figure 1: An overview of the proposed method for architecture derivation

make use of an enhancement of the problem frame approach with the focus on quality requirements. In the third step we evaluate the derived architectures against the quality requirements. To refine the architectures we iterate over these three steps. In each further iteration, subsystems of the architecture alternatives are refined.

We illustrate our approach with 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. The text messages will be shown to the users in that chat room in the current chat session in the correct temporal order on the users' displays*" and its corresponding quality requirement *Response Time* with the description "*The sent text message should be shown on the user's display in 1500 ms maximum*". Note that in order to specify performance requirements properly, more details have to be given. We use the MARTE profile [UML] for this purpose.

The remainder of this paper is structured as follows: Section 2 describes the approach using the chat application as running example. The approach is then discussed in section 3. In section 4 we differentiate the approach from related work, before we conclude and present future work in section 5.

2 Architecture Alternative Derivation Method

The method for deriving architecture alternatives is separated into three steps. In step 1 (section 2.1), solution candidates are rated with respect to the system's requirements. In step 2 (section 2.2) alternative architectures are derived from the best-rated solution candidates. The alternatives are functionally equivalent to an external observer. Their internal structure and behaviour differs. The alternatives are evaluated against the quality requirements in step 3 (section 2.3). These three steps have to be executed manually. We currently develop tool support for the process. To refine the alternatives we iterate over these three steps (section 2.4). In each iteration, subsystems of the derived architecture alternatives are refined. By refining each subsystem of each positively evaluated alternative, a tree of alternative architectures is spanned. Figure 1 gives an overview of the method.

2.1 Step 1: Find Appropriate Solution Candidates

Figure 2 describes the step for finding promising candidates. This step is described in detail in [MKG11]. The inputs are the system's requirements, and a catalogue of solution candidates. The catalogue includes candidates, that reference questions. The questions are targeted at software quality, e.g. scalability or security. Answers to those questions impact the rating for a candidate. These ratings vary between *-1 (probably not appropriate)* and *1 (probably appropriate)* or *excludes*. The latter means that the candidate cannot be applied.

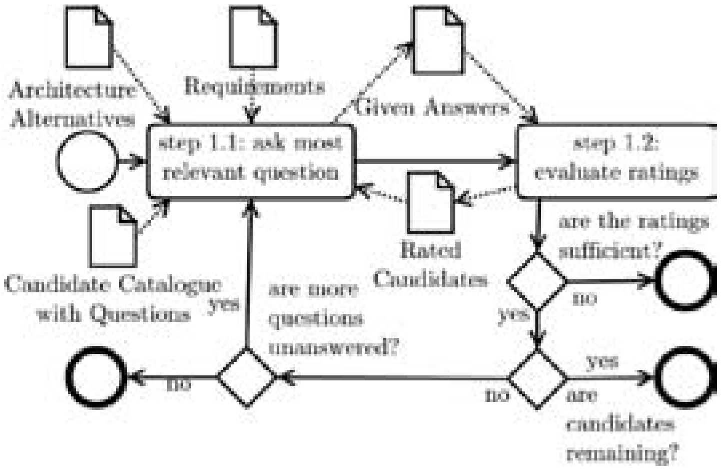
Architecture Alternatives
Requirements
Given Answers
step 1.1: ask most relevant question
step 1.2: evaluate ratings
Rated Candidates
Candidate Catalogue with Questions
are the ratings sufficient?
no
yes
are more questions unanswered?
yes
no
no
yes
are candidates remaining?

Figure 2: An overview of the method's step 1

Step 1.1: Ask Most-Relevant Question

In step 1.1, the questions are sorted in decreasing order by the number of references from candidates. Then, the most referenced question is asked. That question has an impact on the most candidates. Due to the order, the impact of the answers decrease with every asked question. When a question is only referenced by excluded candidates, it will not be asked.

Step 1.2: Evaluate Ratings

In step 1.2 the candidates are sorted in decreasing order by their ratings. A candidate's rating is calculated by each of its answer's rating. If any answer's rating is *excludes*, then the candidate is excluded. Else the result is the arithmetic mean of each answer's

Candidate	**Q1**	**Q2**	**Q3**	**...**	**Arithmetic Mean**
Client/Server	1	0.5	0.9		0.19
Simple Peer-to-Peer	1	0	-0.1		0.22
Standalone	excludes	x	x		0.03
Pipes & Filters	0.2	0	x		0.0
Publish-Subscribe	0.2	0.4	-0.1		0.14
Layered	x	x	0.3	...	0.11
...	...	...	...	...	

Q1: Is the system necessarily distributed? $\rightarrow$ Yes
Q2: Are high request-peaks expected? $\rightarrow$ No
Q3: Does the system conduct sensible / confidential data? $\rightarrow$ Yes

Table 1: Rated answers for the first iteration of the chat application. The columns 2 to 4 show the ratings for answers given to the questions Q1 to Q3. The questions and answers are shown below the table. The last column shows the candidate rating.

ratings. The ratings give guidance to find out which candidates are probably appropriate. If more candidates are remaining, the ratings are not perceived detailed enough, and more questions remain unanswered, step 1.1 will be repeated with the next-most referenced question.

In our example, a pattern catalogue with 10 patterns referencing 25 questions was used. The ratings were defined by experience. An excerpt of the questions and given answers in the first method iteration is shown in table 1. The table shows only the ratings for the given answers. As a result of this step, the candidates *Client/Server* and *Simple Peer-to-Peer* were identified to be the most-promising candidates. The other candidates were excluded or had a lower rating.

2.2 Step 2: Derive Architecture Alternatives

In this step we derive the structural view of the architecture by instantiating architecture alternatives with all solution candidates that we obtained from the previous step. In order to derive a component-based architecture we need to decompose the overall problem in subproblems. For this purpose we use a requirements engineering process based on the problem frames approach [Jac01], which we describe briefly in this section. Then we instantiate the solution candidates by using the results of applying the problem frames approach. Figure 3 shows an overview of the second step.

Requirements Engineering using Problem Frames

We describe briefly an extension of Jackson's problem frames [Jac01], which we apply to instantiate the solution candidates. This approach uses a UML profile [CHH11] that extends the UML meta-model to support problem-frame-based requirements analysis. We

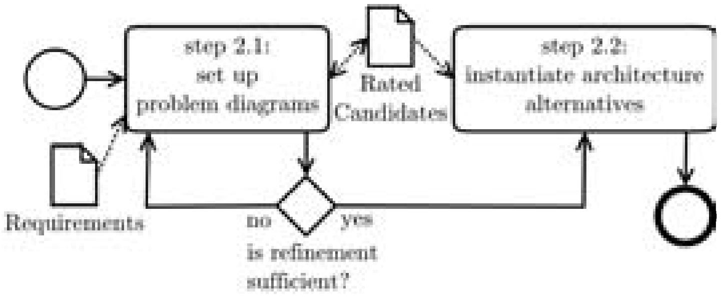
step 2.1: set up problem diagrams
Rated Candidates
step 2.2: instantiate architecture alternatives
Requirements
no
yes
is refinement sufficient?

Figure 3: An overview of the method's step 2

use this profile to create the diagrams for the problem frames approach.

Problem frames are a means to describe software development problems. 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 in a *context diagram*. A context diagram consists of a machine, domains and interfaces. In the UML profile the context diagram is represented by the stereotype ≪`contextDiagram`≫ [1]. 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.

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. In problem diagrams a connection domain establishes a connection between two other domains by means of technical devices. Examples are video cameras, sensors, or networks. A display domain represent a special case of a causal domain (introduced in [CHH$^+$08]). 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. 4 the notation $U!\{sendTM\}$ (between *CA_communicate* and *User*) means that the phenomenon *sendTM* is controlled by the domain *User*.

Each functional requirement constrains at least one domain to show the desire of the change of something in the world. A requirement may refer to several domains in the environment of the machine. Functional requirements are complemented by quality requirements. In the UML profile these relations are shown by corresponding dependencies. To provide support for annotating problem descriptions with performance requirements, we use the UML profile MARTE (Modeling and Analysis of Real-time and Embedded

[1]In the UML profile each model element is represented by the corresponding stereotype

Systems) [UML].

The problem diagram in Figure 4 considers the chat application that was introduced in section 1. It describes the functional requirement *Communicate*. E.g., it states that the *CA_communicate* machine can show to the User the *CurrentChatSession* on its Display ($CAC!\{displayCCS\}$). The requirement constrains the *CurrentChatSession* of the *User* and its *Display*. The requirement refers to the users and the text messages. The requirement *Communicate_RT* describes the response time requirement, which complements the functional requirement *Communicate*. The requirement *Communicate_Conf* describes the confidentiality requirement complementing the functional requirement *Communicate*. In this paper we focus on the response time requirement.

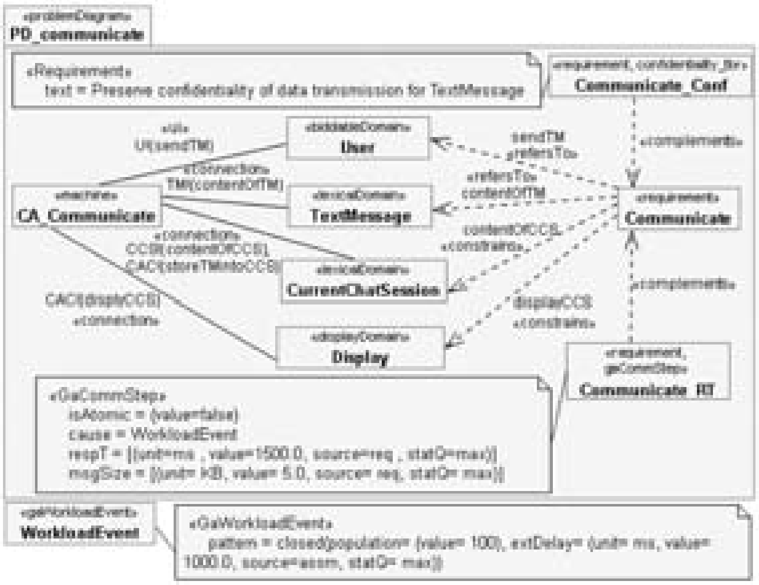

Figure 4: Problem diagram for the requirement *Communicate*

Step 2.1: Set Up Problem Diagrams

In step 2.1 we first set up the problem diagrams to prepare for the instantiation step. We decompose the overall problem into subproblems. Each problem diagram describes one subproblem with the corresponding requirement. Then we annotate each subproblem by complementing functional requirements with related quality requirements. Decomposing the overall problem into subproblems using problem frames is described in detail in [AHH11].

Then we take into account the architectural styles and patterns we obtained by answering the questions. After having chosen the appropriate candidates, we go back to the requirements descriptions and decompose the problem diagrams with respect to the chosen candidates for the architecture. This may lead us to introduce connection domains, e.g., networks. Analogously to decomposing the problem diagrams and so decomposing the functional requirements, we also have to decompose the corresponding quality requirements.

In order to apply this step on the chat example, we first need to decompose the overall chat problem into its subproblems. We focus on the functional requirement *Communicate* (see figure 4) as one subproblem. Then we address quality requirements by annotating the functional requirements with complementing quality requirements. E.g., the requirement *Communicate* is complemented by the response time requirement *Communicate_RT*. The quality requirement in this example is modeled using the MARTE profile for performance annotations. The *respT* attribute states that the required response time for sending text messages should be 1500 ms as 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.

In order to decompose problem diagrams properly we take into account the solution candidates we obtained from step 1. We proceed the example with the *Client/Server* alternative and describe each step of its instantiation in detail. After having chosen the *Client/Server* architecture style we decompose the problem diagrams so that each subproblem is allocated to only one of the distributed components.

In our example, we decompose the problem diagram depicted in figure 4 into three subproblem diagrams *Send* (Client), *Forward* (Server), and *Receive* (Client). *Send* addresses the problem of sending text messages to the server (not shown). *Forward* addresses the forwarding of text messages from the server to the receivers (see figure 5). *Receive* addresses the receiving of text messages (not shown). For each of these three subproblems, we introduced the connection domain *Network* to achieve the distribution.

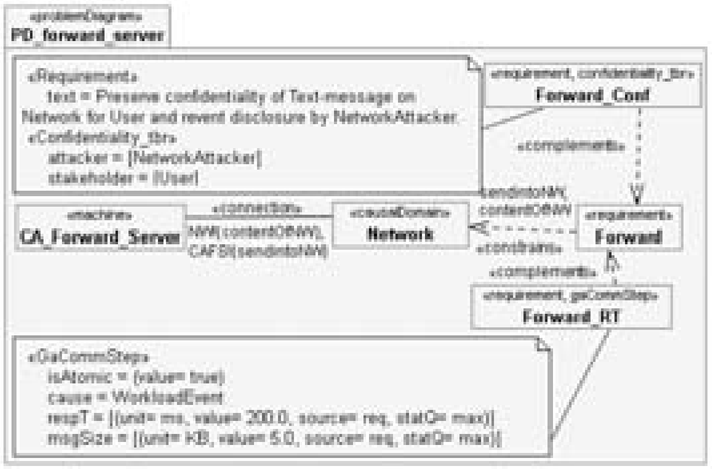

Figure 5: Problem diagram for the subproblem *Forward*, annotated with quality requirements

To fulfill the 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. We must also consider the time that the data needs to be transported over the network. In our example, 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. Note that knowledge about the real circumstances in the environment e.g. about the network and the computational power of clients and server is needed to meet performance and specifically response time requirements.

Step 2.2: Instantiate Architecture Alternatives

In step 2.2 we derive the architecture by instantiating the solution candidates through subproblems. The initial architecture that we have to refine consists of one component for the overall machine. Each submachine, which belongs to one decomposed problem diagram becomes a component in the architecture. In further iterations we refine each component by introducing domains reflecting specific solution approaches we obtained from the decomposed problem diagrams.

For the chat example we derive the architecture in the first iteration by establishing one component for the overall machine *chat application*. To indicate the distribution we add the stereotype ≪`distributed`≫ from the UML profile for problem frames to the architecture component. For the *Client/Server* architecture style, there are two components representing the clients and the server, respectively, inside the overall machine. Then we make use of the subproblem diagrams. Each submachine in the subproblem diagrams becomes a component either in the client or in the server. The submachines *CA_send* and *CA_receive* belong to the client component, whereas *CA_forward* belongs to the server.

2.3 Step 3: Evaluate Architecture Alternatives

In the evaluation step of our method, the derived architecture alternatives have to be evaluated against the system's quality requirements. Typically, a trade-off analysis will be part of the evaluation, because system qualities are often contradictory. This method does not constrain the choice of quality evaluation methods. As a result of the evaluation, some of the derived architecture alternatives can be excluded when they show to perform worse than other alternatives regarding the quality requirements. Other alternatives will be considered for refinement in the method's next iterations.

As the evaluation method is not in focus of this paper, we will not go into details here. In our example both candidates passed the evaluation. Thus in the next iterations, each candidate is considered further when the architecture is refined.

2.4 Further Iterations: Architecture Refinement

Software architectures can be specified at different levels of details. Our method reflects this by allowing for arbitrary method iterations. In the first iteration, alternatives for the overall system are derived. In each further iteration, subsystems of these alternatives are refined by executing the method with the focus on a subsystem, instead of the overall system. Consequently, only the set of requirements are considered, that are relevant to the specific subsystem in focus. The candidates ratings are reset, so that the questions need to be answered again, though against the focused subsystem's requirements.

In our example, we first focus on the *Client/Server* alternative. Within this alternative we refine the subsystem *Client*. By answering the questions in step 1, *Load Balancer* is rated as an appropriate solution candidate. In step 2, we elaborate the problem diagrams by introducing domains reflecting load balancing as a solution candidates. To specify the quality problem diagram given in figure 6 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.

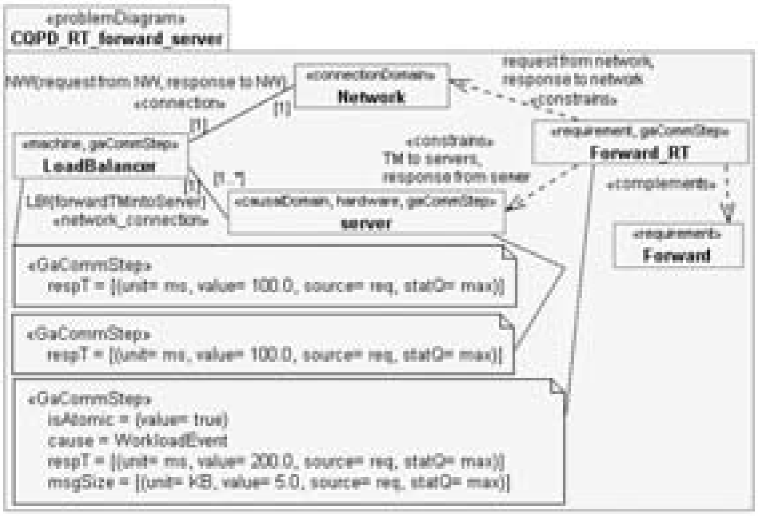

Figure 6: Problem diagram for the quality requirement *Forward_RT*

After elaborating all problem diagrams with solution candidates and instantiating the software architecture with them, we obtain the software architecture shown in figure 7 as one alternative with the *Client/Server* architectural style. The *LoadBalancer* is placed before the servers. Its port multiplicity [1..*] means that it can be connected with several server components. The Peer-to-Peer alternative and the refinement of the Client subsystem are handled analogously. This is not shown in this paper.

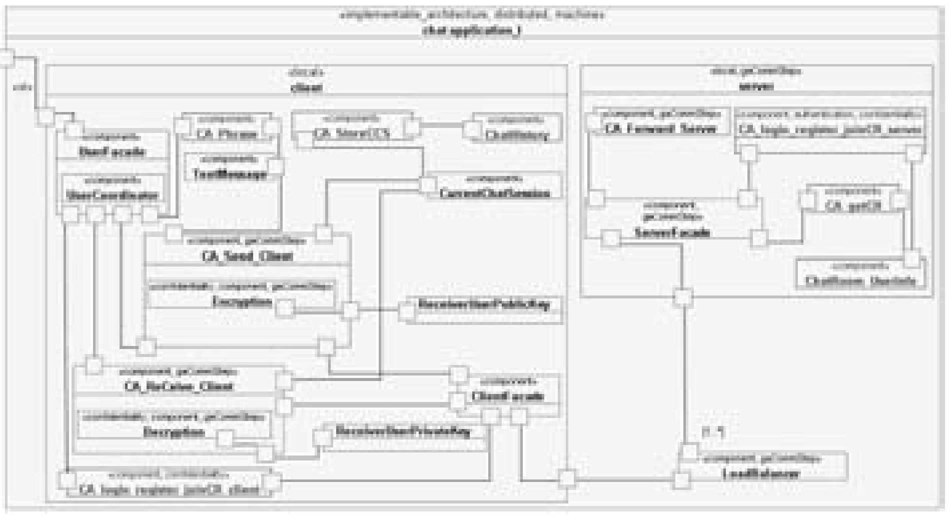

Figure 7: Implementable architecture *Communicate*

3 Discussion

The task of architectural design cannot be completely automated. Thus the approach presented here can also only give guidance. The architect has to interpret the results of the candidate ratings and choose which candidate to consider. While the ratings suggest that a candidate with a rating (arithmetic mean of all single ratings) of 0.7 is more appropriate than a candidate with the rating 0.6, this is not necessarily true. The step of instantiation also leaves choices to the architect. Thus the method is non-deterministic.

The success of the method is highly dependent on the amount and quality of data it is based on, i.e. questions, ratings, and candidates. The current set of data is small and provides only a few candidates and questions. More data should be provided by a community of experienced architects. We plan on extending the data base.

The proposed method aims at systematically deriving architecture alternatives that differ in their quality attributes, taking the quality requirements into account. Our experiments (on a distributed, highly configurable load generator and on smaller information systems, all developed in one of our working groups) indicate that the method is usable and helps to systematically explore the design space of a system, while reducing the development effort by highlighting appropriate alternatives. A systematic evaluation is still to be performed.

4 Related Work

The systematic derivation of architectures from requirements has been subject to research in related work for many years [SB82, Nus01]. The Twin Peaks Model [Nus01] integrates requirements elicitation with architectural design. In this method, requirements and architecture are refined concurrently, while providing feedback to each other. However, the details and mechanisms of the feedback is not specified. Also, in contrast to our approach, no alternatives are developed, that can be considered in an evaluation.

In [vL03] van Lamsweerde proposes a goal-oriented approach to architectural design. Van Lamsweerde first considers functional requirements, before adapting the architecture to meet quality requirements. In our approach, we focus on quality requirements first by choosing the solution candidates based on the quality requirements. We then add the required functionality in a systematic manner.

Attribute Driven Design (ADD) is a method that emphasizes the use of patterns and styles while deriving an architecture from requirements. However, ADD does not describe how to elicit a matching pattern from a large pattern catalogue, except for sequentially evaluating each element in the catalogue (cf. [WBB$^+$07]). There are more architecture methods that are imprecise at this point such as Siemens 4 Views [HNS99]. Therefore, our approach bridges the gap of pattern elicitation found in related work.

Zdun uses questions to select architecture patterns in [Zdu07]. In this approach, questions are directed to key characteristics of a group of patterns (e.g. "*How to realize asynchronous result handling?*"). The answers are patterns, which are related to criteria supporting the decision process. Other approaches (e.g. [HA07]) relate patterns to coarse-grained quality attributes. We believe that our approach is more suitable for less experienced teams, because they are more fine-grained and related to the system's requirements and context. This is, however, subject to validation.

Bode and Riebisch [BR10] relate solutions to quality goals. Their work focuses on rating the impact of patterns on quality goals. Bode and Riebisch develop context-independent ratings, aiming at fulfilling abstract quality goals. In contrast, our approach also takes the system's context and environmental constraints into account by explicitly considering the requirements, e.g. insecure networks that have to be used. We are confident that this allows to rate solution candidates more precisely, because such details can have a strong impact on the choice of patterns and styles.

5 Conclusion and Future Work

In this paper, we presented a method for deriving architectural alternatives for software systems. The method allows for systematically exploring the design space and aims at highlighting appropriate alternatives. It is iterative, so that it can be performed until the desired level of detail is reached. Our experiments have shown that the method is usable and helpful.

As future work, we plan to find mechanisms or a community to provide data, as the quantity and quality of ratings and patterns is very important to render the method helpful. In addition, a case study is work in progress.

References

[AHH11] Azadeh Alebrahim, Denis Hatebur, and Maritta Heisel. Towards Systematic Integration of Quality Requirements into Software Architecture. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, LNCS 6903, pages 17–25. Springer, 2011.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (SEI Series in Software Engineering)*. Addison-Wesley, 2. a. 2003. edition, 4 2003.

[BR10] Stephan Bode and Matthias Riebisch. Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability. In Muhammad Babar and Ian Gorton, editors, *Software Architecture*, LNCS 6285, pages 182–197. Springer, 2010.

[CHH$^+$08] Isabelle Côté, Denis Hatebur, Maritta Heisel, Holger Schmidt, and Ina 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.

[CHH11] 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*, pages 133–160. Springer, 2011.

[HA07] Neil Harrison and Paris Avgeriou. Leveraging Architecture Patterns to Satisfy Quality Attributes. In Flavio Oquendo, editor, *Software Architecture*, volume 4758 of *Lecture Notes in Computer Science*, pages 263–270. Springer Berlin / Heidelberg, 2007.

[HNS99] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture: A Practical Guide for Software Designers*. Addison-Wesley, 11 1999.

[Jac01] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[MKG11] Marco Müller, Benjamin Kersten, and Michael Goedicke. A Question-Based Method for Deriving Software Architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, LNCS 6903, pages 35–42. Springer, 2011.

[Nus01] Bashar Nuseibeh. Weaving Together Requirements and Architectures. *Computer*, 34:115–117, March 2001.

[SB82] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Commun. ACM*, 25:438–440, July 1982.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 1. auflage edition, 2 2009.

[UML] 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.

[vL03] Axel van Lamsweerde. From System Goals to Software Architecture. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures*, LNCS 2804, pages 25–43. Springer, 2003.

[WBB$^+$07] Rob Wojcik, Felix Bachmann, Len Bass, Paul Clements, Paulo Merson, Robert Nord, and Bill Wood. Attribute-Driven Design (ADD), Version 2.0. Technical report, Software Engineering Institute, 2007.

[Zdu07] U. Zdun. Systematic Pattern Selection using Pattern Language Grammars and Design Space Analysis. *Software: Practice and Experience*, 37(9):983–1016, 2007.