Supporting Quality-Driven Design Decisions by Modeling Variability *

Azadeh Alebrahim
azadeh.alebrahim@paluno.uni-due.de

Maritta Heisel
maritta.heisel@paluno.uni-due.de

paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Germany

ABSTRACT

Design decisions should take quality characteristics into account. To support such decisions, we capture various solution artifacts with different levels of satisfying quality requirements as variabilities in the solution space and provide them with rationales for selecting suitable variants. We present a UML-based approach to modeling variability in the problem and the solution space by adopting the notion of feature modeling. It provides a mapping of requirements variability to design solution variability to be used as a part of a general process for generating design alternatives. Our approach supports the software engineer in the process of decision-making for selecting suitable solution variants, reflecting quality concerns, and reasoning about it.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Design

Keywords

Quality requirements, design alternatives, variability modeling, feature modeling, decision-making

1. INTRODUCTION

Functional requirements capture the functionality of the system-to-be, while quality (non-functional) requirements represent quality issues of software systems. They have no clear-cut satisfaction criterion and can only be satisfied to a certain degree [4]. This leads to variability in the solution space. Quality requirements can therefore be considered as drivers in the problem space that contribute to variability in the solution space.

The general research objective in our ongoing project[1] is to derive design alternatives from quality requirements. We investigate how different user preferences and needs regarding security and performance can influence the design of software. In previous work, we developed a process to derive software architectures from quality requirements in a systematic way [1]. We also discussed the derivation of design alternatives based on quality requirements. However, this issue is not treated in depth.

This paper is concerned with supporting decisions to be taken to select suitable design solutions. In particular, we provide an appropriate view to capture the variability in the solution space systematically and to represent it explicitly. This enables the software engineer to select appropriate solution variants and to derive design alternatives based on quality requirements.

Our proposed process to derive design alternatives from (quality) requirements relies on problem frames [12]. 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[2] is a widely used notation to express analysis and design artifacts, we have carried over problem frames to UML by defining a specific UML profile [10] for problem frames.

Software Product Line Engineering (SPLE) deals with "the commonalities and differences in the applications in terms of requirements, architecture, components, and test artifacts" [16]. In SPLE, the common and variable parts are modeled as feature diagrams [6, 13, 14]. Using the notion of feature modeling, we develop a *problem-solution feature model* that accommodates possible solution variants.

Each design alternative represents a valid instance of the software architecture composed of a set of solutions satisfying functional requirements (commonalities) and a valid set of solution variants satisfying quality requirements to a certain degree (variabilities). The set of functional solutions represents the architecture skeleton. The problem-solution feature model as one design view represents functional solutions as commonalities and quality solutions as variabilities provided with rationales. It therefore supports the process of decision-making for selecting suitable design variants from the space of quality solution variants by providing a mapping of requirement artifacts to design solution artifacts.

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

QoSA'12, June 25–28, 2012, Bertinoro, Italy.

[1] GenEDA (Generation and Evaluation of Design Alternatives for Software Architectures), www.geneda.org

[2] http://www.omg.org/spec/UML/2.3/Superstructure/PDF

The problem frame approach is only suited for the requirements analysis of single systems and fails to support variability at the requirement level. Also UML is not per se appropriate to model variability. However, it allows us to extend it with variability modeling by using the standardized extension mechanisms. We address this lack by incorporating an intermediate step into the proposed process to derive design alternatives. This step deals with variability by adopting the notion of feature modeling. We extend the UML meta-model to model variability at the requirements and the design level and to provide a mapping between variability at different levels. This makes the approach usable with conventional UML tools.

The remainder of the paper is organized as follows. In Sect. 2, we describe the background of this work by giving an overview of problem frames as well as feature modeling and variability. An example is introduced in Sect. 3. We describe our approach in Sect. 4. Related work is discussed in Sect. 5. We conclude and give hints to future work in Sect. 6.

2. BACKGROUND

In this section, we give an overview on two techniques our work relies on.

2.1 Problem-Oriented Requirements Engineering

Problem frames are a means to describe and classify software development problems. A problem frame represents a class of software problems. It is described by a *frame diagram*, which consists of domains, interfaces between them, and a requirement. Domains describe entities in the environment. 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. *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 (indicated by an exclamation mark).

The objective is to construct a *machine* (i.e., software) that controls the behavior of the environment (in which it is integrated) in accordance with the requirements. Requirements analysis with problem frames proceeds with a decomposition of the overall problem into subproblems, which are represented by *problem diagrams*. The environment in which the machine will operate is represented by a *context diagram*.

To use the advantages of UML for modeling software systems and provide tool support for problem frames, we have carried over problem frames to UML by defining a specific UML profile [10]. To address quality requirements, we extended our UML profile for problem frames to complement functional requirements with quality requirements [1].

We have developed a tool called UML4PF[3]. It is conceived as an Eclipse plug-in and includes the UML profile for problem frames. In conjunction with the Eclipse development environment[4] and Papyrus[5], it can be used to create and validate the context and problem diagrams.

[3]available under http://www.uml4pf.org
[4]http://www.eclipse.org/
[5]http://www.papyrusuml.org/

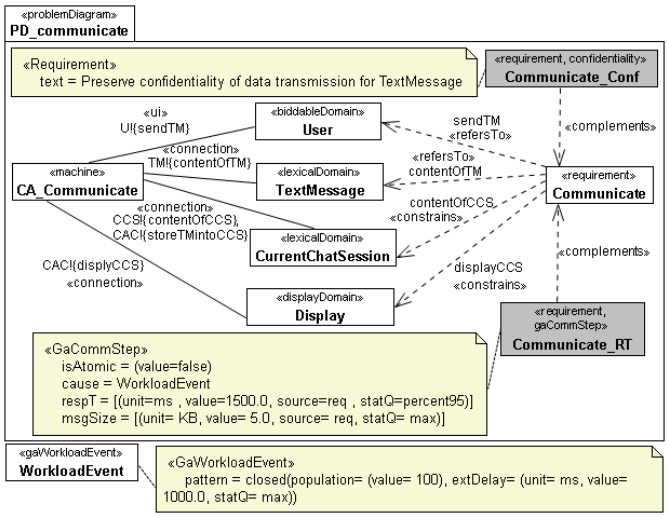

Figure 1: Problem diagram for the requirement *Communicate* using UML profile for problem frames

2.2 Feature Modeling and Variability

Feature modeling, first proposed in feature-oriented domain analysis (FODA) [13], is a domain analysis technique and notation to describe the requirements space. A feature model captures and models the variability of features in a system family.

Four types of features are distinguished [6]: mandatory, optional, alternative, and OR features. A mandatory feature should exist in every product of a product line if its parent feature exists, while an optional feature may exist if its parent exists. One feature from a set of alternative features must be selected if a parent feature is selected. At least one feature from the set of OR features must be selected if a parent feature is selected. The feature modeling notation has been extended by Czarnecki et al. [7] with concepts such as feature and group cardinalities. The semantics of feature diagrams proposed in FODA has been enriched in FORM [14] by introducing different perspectives.

The feature model encompasses a graphical hierarchy of features known as a feature diagram, composition rules, which are constraints for the use of features, and issues and decisions that provide the rationale for selecting among alternatives.

3. EXAMPLE

We illustrate our approach by a chat application, which supports text-message-based communication via private I/O devices. Users should be able to communicate with other chat participants in the same chat room. The functional requirement *Communicate* is as follows: "*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 complement the functional requirement *Communicate* with corresponding quality requirements such as *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 confiden-*

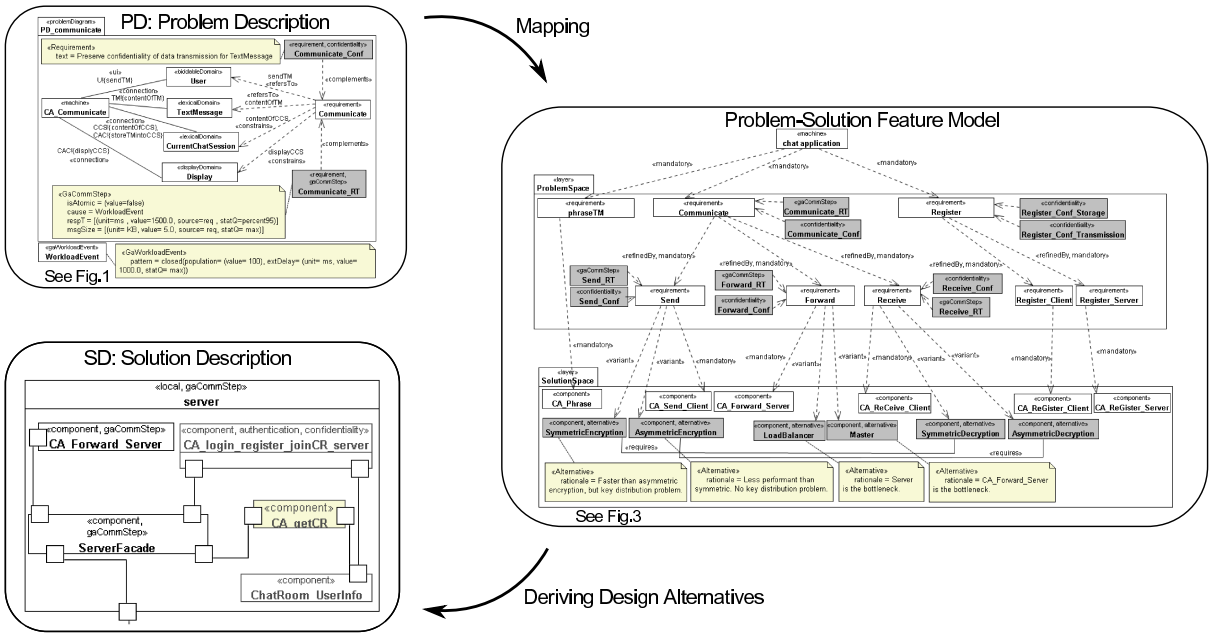

Figure 2: Overview of the process to generate design alternatives including decision support

tial way" as performance[6] and security requirements. The user should also be able to register, log in, log out, phrase text messages and store the current chat session. Figure 1 describes the requirement *Communicate*. It states that the machine *CA_communicate* can show to the *User* the *CurrentChatSession* on its *Display* ($CAC!\{displayCCS\}$). The stereotype ≪`requirement`≫ represents a 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 requirement *Communicate* constrains the *CurrentChatSession* of the *User* and its *Display* and refers to the users and the text messages. The quality requirements *Communicate_RT* and *Communicate_Conf* complement their corresponding functional requirement, which is expressed by the stereotype ≪`complements`≫.

Throughout the paper we will refer to this example to describe the proposed process.

4. DECISION SUPPORT FOR ARCHITECTURAL DESIGN

We first describe our current process [1] to derive software architectures from quality requirements that consists of two main steps *Problem-Oriented Requirements Engineering* and *Generation of Design Alternatives*. Second, we define a UML profile to be applied in the process to support variability with UML. Third, we introduce the *problem-solution feature model* as an intermediate step in our current process that assists the software engineer in the process of decision-making for selecting proper solution variants. Figure 2 gives an overview of the complete process. Finally, we define a number of integrity conditions to check the consistency between the problem-solution feature model and other model artifacts.

4.1 Process Overview

In the first main step of our process (*Problem-Oriented Requirements Engineering*), we decompose the overall problem into subproblems, represented as *problem diagrams*. Each subproblem contains one submachine to satisfy the corresponding functional requirement (e.g., submachine *CA_Communicate* should satisfy the functional requirement *Communicate* in Fig. 1). We then enrich the subproblems with related quality requirements. Quality requirements are expressed as complements of functional requirements. Various degrees of satisfaction cause variabilities in the problem space that contribute to the variability in the solution space. Each subproblem is related to at least one functional requirement and possibly one or more quality requirements.

To fulfill quality requirements, we enrich the subproblems by incorporating solution approaches in terms of mechanisms and patterns reflecting quality concerns. We propose alternative solution approaches that vary in the degree of satisfaction according to the problem context.

In the next main step (*Generation of Design Alternatives*), we derive a software architecture that achieves the necessary, but varied satisfaction levels of quality requirements. To this end, we make use of the solution approaches we integrated in the subproblems. Submachines in the subproblems are mapped to components of the software architecture (on the lower left-hand side of Fig. 2). The software architecture is represented as a composite structure diagram. Mapping solution approaches to components of the software architecture is feasible, as long as we do not consider various solution approaches for one quality requirement, which leads to design alternatives. To generate design alternatives systematically, we need an intermediate step to capture and explicitly represent the variability discovered in the problem space and to transform it into the design

[6]We use the MARTE profile [17] to specify performance requirements.

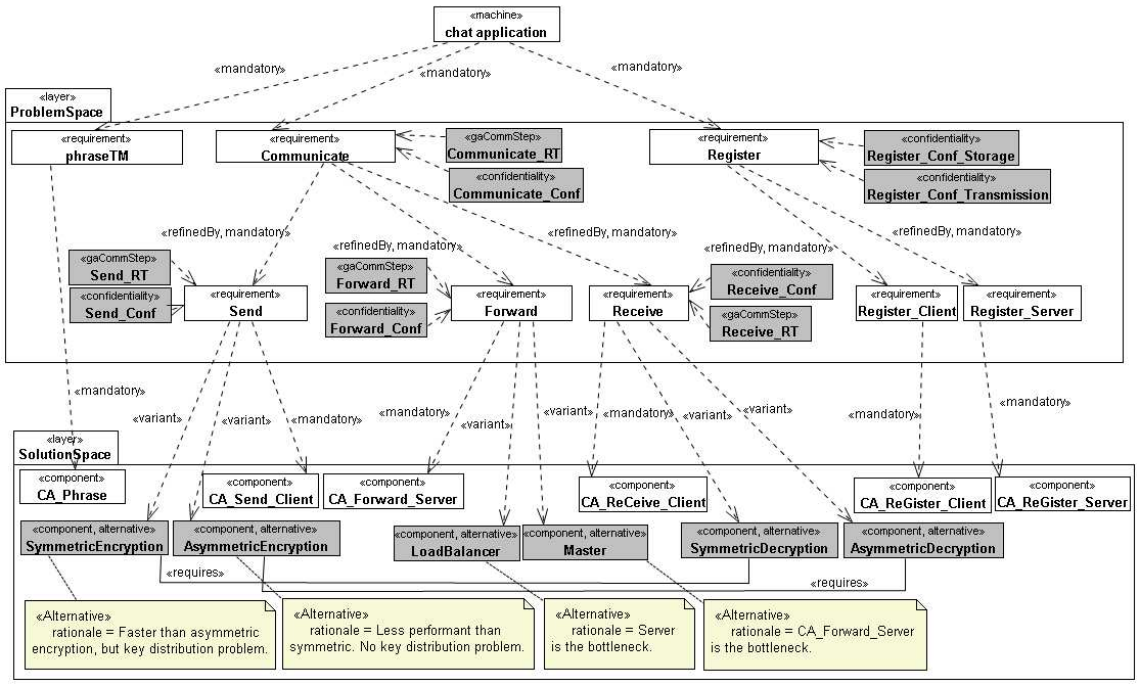

Figure 3: Problem-Solution Feature Model

space. We therefore enrich the current process with feature models representing variability by introducing the problem-solution feature model (on the right-hand side of Fig. 2). It provides a mapping of the requirements in the problem description (functional and quality requirements) to their corresponding machines (functional and quality solutions). We provide the solution variants with rationales to support the selection process among them. Modeling variability at the requirements level facilitates the identification of variations in the solution space. The intermediate step links the variability in the problem space to the variability in the solution space. Figure 2 depicts an overview of the current process including problem and solution descriptions with its extension *problem-solution feature model*.

4.2 UML Extension for Feature Modeling

In order to support variability modeling with UML in our approach, we define a UML profile that extends the UML meta-model by defining stereotypes and tagged values. This profile can be easily incorporated into our tool UML4PF.

To express variability, we define stereotypes for dependencies and classes. We introduce stereotypes ≪`variant`≫ and ≪`mandatory`≫ as dependencies between a feature class and its subfeature class. A ≪`variant`≫ dependency points to a variant feature. Feature classes representing variability are expressed with the stereotypes ≪`Alternative`≫, ≪`Optional`≫, and ≪`Or`≫ as specializations of the class with the stereotype ≪`Feature`≫. These features can be annotated with a rationale to select them [6]. Therefore, we define *rationale* as an attribute for a feature class to capture the reasoning about its selection.

The concept of layers in a feature model is introduced in FODA [13] and FORM [14]. We define the stereotype ≪`Layer`≫ for packages to facilitate modeling of different levels of abstraction. Decomposition relationships between features and subfeatures are represented in FODA with a *consists-of* relationship, in FORM with a *composed-of* relationship, and by Fey et al. [9] with a *refine* relationship. We define the stereotype ≪`refinedBy`≫ for a dependency between a feature and its subfeature to enable refining of features into more detailed subfeatures. FODA proposes the concept of *composition rules*, which are dependency relationships between features. We define two stereotypes ≪`requires`≫ and ≪`conflict`≫ to express the relationships between two features and to select between them.

4.3 Problem-Solution Feature Model

The *problem-solution feature model* containing requirements and corresponding solution mechanisms forms a foundation for representing design alternatives. This model, comprising problem and solution spaces, captures commonalities and variabilities in the requirements in the *problem space* and commonalities and variabilities in the solutions in the *solution space* explicitly. In the problem space, functional requirements represent commonalities, and quality requirements represent variabilities. The solution space encompasses functional machines as commonalities and solution mechanisms for quality requirements as variabilities. In order to generate the feature model we set up a tree with the system-to-be (machine in the problem frames terminology) as the root node, all requirements needed to build the machine (requirements in the problem diagrams) as feature nodes in the problem space, and the corresponding functional machines and solution mechanisms (submachines in the problem diagrams) as feature nodes in the solution space. Figure 3 shows a part of the problem-solution feature model for the running example.

Chat application as the root node points to the functional requirements representing commonalities via a dependency with the stereotype ≪`mandatory`≫. Corresponding quality requirements representing variabilities are highlighted in gray. Functional requirements are decomposed

into more refined functional requirements, expressed by a dependency with the stereotype ≪refinedBy≫. The functional requirement *Communicate* is refined into three functional requirements *Send*, *Forward*, and *Receive*. Functional and quality requirements are constituent parts of the problem space layer. The solution space layer encompasses the solutions for the requirements in the problem space. Requirements are linked to the solutions in the solution space. Functional requirements point to the functional machines, which serve as components (stereotype ≪component≫) in the software architecture. A dependency with the stereotype ≪mandatory≫ points to a component occurring in each design alternative. These feature nodes represent the architectural skeleton. For reasons of clarity, in Fig. 3 we link functional requirements instead of quality requirements to the solution mechanisms (highlighted in gray). This is possible, because quality requirements are represented as complement to the functional requirements. A dependency with the stereotype ≪variant≫ points to the variant solutions of the design alternatives with the stereotype ≪alternative≫. These feature nodes fulfilling corresponding quality requirements in the problem space bound the scope of variations of the architectural skeleton. An association with the stereotype ≪requires≫ between two solution variants supports the selection of allowed combinations of alternatives. In Fig. 3, the association between two solution variants *Symmetric Encryption* and *Symmetric Decryption* indicates that both variants should be selected as components of one design alternative. The alternative features can be annotated with the attribute *rationale* to capture the rationale behind a decision and support the decision-making process. Thus, traceability between requirements in the problem space and architectural components in the solution space is achieved.

Hence, an instance of a design alternative is a subtree containing all functional machines (architectural skeleton) and a valid subset of solution mechanisms.

Applying the UML profile to model features with UML facilitates the management of solution configurations. Our UML profile supports the representation of solution variants and the selection of components (solutions) to obtain a concrete architecture configuration.

4.4 Integrity Conditions

We have identified a number of integrity conditions to check the consistency between problem diagrams and the problem-solution feature model. These conditions are expressed as OCL[7] constraints. The following list shows examples of such validation conditions in natural language. It is easily possible to identify further conditions and incorporate them into UML4PF.

- All requirements in the problem space must occur in the problem diagrams and vice versa. Listing 1 depicts the corresponding OCL expression.
- All components in the solution space must be machines in the problem diagrams and vice versa.
- The machine domain of the problem-solution feature model must occur in the context diagram and vice versa.

To check the first constraint (see Listing 1), we define "pd_reqs" to be all requirement classes in problem diagrams: we select all classes with the stereotype ≪Requirement≫ from the packages with the stereotype ≪ProblemDiagram≫ (lines 1-9). Then we define "fd_reqs" to be all requirement classes in the problem space. We select all classes with the stereotype ≪Requirement≫ from the packages with the stereotype ≪Layer≫ with the name *ProblemSpace* (lines 10-19). We check if the requirements in the problem space are the same as in the problem diagrams and vice versa (line 20).

```
let                                                                    1
pd_reqs:Set(Class)=Package.allInstances() ->                           2
select(getAppliedStereotypes().name->                                  3
includes('ProblemDiagram')).                                           4
clientDependency.target ->                                             5
select(getAppliedStereotypes().name ->                                 6
includes('Requirement')).oclAsType(Class)                              7
->asSet()                                                              8
in                                                                     9
let                                                                    10
fd_reqs:Set(Class)=Package.allInstances() ->                           11
select(getAppliedStereotypes().name ->                                 12
includes('Layer'))->select(name='ProblemSpace').                       13
clientDependency.target ->                                             14
select(oclIsTypeOf(Class)) ->                                          15
select(getAppliedStereotypes().name ->                                 16
includes('Requirement')).oclAsType(Class)                              17
->asSet()                                                              18
in                                                                     19
fd_reqs=pd_reqs                                                        20
```

Listing 1: All requirements in the problem space must occur in some problem diagram

5. RELATED WORK

One related research topic aims at adopting the notion of feature modeling to generate software architectures. A feature-based method similar to ours is proposed by Bruijn and van Vliet [3] to generate software architectures with respect to functional and non-functional requirements. In contrast to our proposed process the authors treat functional and non-functional requirements separately, constructing two branches in the feature graph. Bruijn and van Vliet generate design alternatives by using *Use Case Maps (UCM)* as a scenario-based architectural description language, which is not model- and pattern-based.

To treat variability in the requirements analysis and consequently generate a customizable software design, Hui et al. [11] propose a framework for identifying requirements (user goals, user skills, user preferences) from a user perspective. However, this work focuses on earlier stages of requirements analysis by choosing goals to represent and analyse variability.

There are a number of UML extensions to model variability of product lines. A UML profile is proposed by Clauß [5]. This profile supports only the UML 1.4 and not the current UML version. Ziadi et al. [19] model the relationships between features as constraints, while we model such relationships as stereotypes, such as ≪requires≫ and ≪conflicts≫.

Another research topic is concerned with connecting problem frames with feature modeling. Zuo et al. [20] introduce an extension of the problem frames notation that provides support for product line engineering using the notion of feature analysis. While Ali et al. [2] propose a method to treat the variability of context (conditions in the operating environment influencing the behaviour of the system) in requirements, we take a step forward and connect functional and quality requirements to commonality and vari-

[7] http://www.omg.org/spec/OCL/2.0/PDF

ability in the solution space, expressed by alternative solutions. An approach for integrating SPLE and the problem frame concept considering domain concerns is proposed by Dao et al. [8]. In this work a feature model is mapped to a problem frames model. A goal model is adopted to represent various concerns and variable quality requirements. The mapping between these different models is complicated and time-consuming, hence requires a tool support.

There are approaches concerning the modeling of quality properties in the context of feature modeling. Yu et al. [18] relate stakeholder goals including quality properties captured as goal models to a feature model by introducing a mapping between goals and features. Lee and Kang [15] consider the usage context as the primary driver for feature selection. The authors present three variability models for the usage context, quality attributes, and the product and three mappings between them to derive a product configuration.

6. CONCLUSIONS AND FUTURE WORK

We have completed our previous work to derive design alternatives by introducing the problem-solution feature model as an intermediate step, which connects (quality) requirements with solutions variants. Quality requirements drive the variability in the solution space and serve as selection criteria. In addition, we annotated quality solutions in the solution space with rationales for choosing among alternatives. Thus, the problem-solution feature model represents a "decision space" and provides a good starting point to identify candidates for solution variants. From the problem-solution feature model, design alternatives can be derived as views of suitable solution candidates, represented as composite structure diagrams. Our proposed approach supports maintaining traceability between model artifacts, namely between problem descriptions, feature models, and architectural descriptions.

Setting up problem descriptions and problem-solution feature models is supported by our UML4PF tool. It provides the possibility to automatically check semantic integrity conditions for individual model artifacts, as well as coherence conditions between different models. Having UML models, our approach facilitates model transformations. In the future, we aim to automatically transform the problem descriptions into the problem-solution feature model to reduce the effort of manually modeling the problem-solution feature model. Moreover, we will formalize the derivation of design alternatives from the problem-solution feature model, using UML model transformations. Eventually, we strive for an evaluation of the approach in a larger scope.

Acknowledgments.

We would like to thank Stephan Faßbender, Rene Meis, and Kristian Beckers for useful discussions.

7. REFERENCES

[1] A. Alebrahim, D. Hatebur, and M. Heisel. A method to derive software architectures from quality requirements. In *APSEC'11*, pages 322–330. IEEE Computer Society, 2011.

[2] R. Ali, Y. Yu, R. Chitchyan, A. Nhlabatsi, and P. Giorgini. Towards a Unified Framework for Contextual Variability in Requirements. In *IWSPM'09*, 2009.

[3] H. d. Bruijn and J. C. v. Vliet. Scenario-Based Generation and Evaluation of Software Architectures. In *GCSE'01*, pages 128–139. Springer Verlag, 2001.

[4] L. Chung, B. Nixon, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[5] M. Clauß. Modeling variability with UML. In *GCSE – Young Researchers Workshop*, 2001.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications: Methods, Techniques and Applications*. Addison-Wesley, 2000.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *SPLC'4*, pages 266–283. Springer Verlag, 2004.

[8] T. M. Dao, H. Lee, and K. C. Kang. Problem frames-based approach to achieving quality attributes in software product line engineering. In *SPLC'11*, pages 175–180. IEEE, 2011.

[9] D. Fey, R. Fajta, and A. Boros. Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In *SPLC'2*, pages 198–216. Springer Verlag, 2002.

[10] D. Hatebur and M. Heisel. Making Pattern- and Model-Based Software Development more Rigorous. In *ICFEM'10*, pages 253–269. Springer Verlag, 2010.

[11] B. Hui, S. Liaskos, and J. Mylopoulos. Requirements Analysis for Customizable Software: A Goals-Skills-Preferences Framework. In *RE'03*, pages 117–126, 2003.

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

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[14] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[15] K. Lee and K. C. Kang. Usage Context as Key Driver for Feature Selection. In *SPLC'10: going beyond*, pages 32–46. Springer Verlag, 2010.

[16] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer Verlag, 2005.

[17] "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.

[18] Y. Yu, J. C. S. do Prado Leite, A. Lapouchnian, and J. Mylopoulos. Configuring Features with Stakeholder Goals. In *SAC'08*, pages 645–649. ACM, 2008.

[19] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML Profile for Software Product Lines. In *PFE'03*, pages 129–139. Springer Verlag, 2003.

[20] H. Zuo, M. Mannion, D. Sellier, and R. Foley. An Extension of Problem Frame Notation for Software Product Lines. In *APSEC'5*, pages 499–505. IEEE Computer Society, 2005.