# Towards a Computer-Aided Problem-Oriented Variability Requirements Engineering Method*

Azadeh Alebrahim, Stephan Faßbender, Martin Filipczyk,
Michael Goedicke, Maritta Heisel, and Marco Konersmann

Paluno – The Ruhr Institute for Software Technology, Germany
firstname.lastname@paluno.uni-due.de

**Abstract.** In theory, software product lines are planned in advance, using established engineering methods. However, there are cases where commonalities and variabilities between several systems are only discovered after they have been developed individually as single systems. In retrospect, this leads to the hindsight that these systems should have been developed as a software product line from the beginning to reduce costs and effort. To cope with the challenge of detecting variability early on, we propose the PREVISE method, covering domain and application engineering. Domain engineering is concerned with exploring the variability caused by entities in the environment of the software and the variability in functional and quality requirements. In application engineering, the configuration for a concrete product is selected, and subsequently, a requirement model for a concrete product is derived.

**Keywords:** Variability modeling, problem frames, software product lines (SPL), orthogonal variability modeling (OVM), UML profile.

## 1 Introduction

In our ongoing project GenEDA[1], we aim at extending our method for deriving design alternatives from quality requirements [11], which supports a single-system development to a product-line development addressing quality requirements. Software product line engineering (SPLE) represents an emerging paradigm to develop software applications which are tailored to individual customer's needs [12].

Software product lines (SPL) involve a set of common features as well as a set of variable ones. The first challenge we are facing is how to utilize and adjust conventional requirements engineering techniques for modeling and engineering SPL. Modeling and managing variability is the central concept in SPLE. Beyond the variability which is caused by variable requirements, there exist further variabilities, which might emerge because of changes in the environment in which the software will be located. Such kind of variability should be taken into consideration when developing SPL.

In this paper, we propose the PREVISE (**PR**obl**E**m-oriented **V**ar**I**ability Requirement**S** **E**ngineering) method, which conducts requirements engineering in software

* This research was partially supported by the German Research Foundation (DFG) under grant numbers HE3322/4-2 and GO774/5-2.

[1] www.geneda.org

L. Iliadis, M. Papazoglou, and K. Pohl (Eds.): CAiSE 2014 Workshops, LNBIP 178, pp. 136–147, 2014.
© Springer International Publishing Switzerland 2014

product lines considering quality requirements. Our method is composed of four phases. It covers domain engineering (phases one and two) as well as application engineering (phases three and four).

The PREVISE method uses the problem frames approach [10] as a basis for requirements engineering and extends it for developing SPL. We use the problem frames approach, because 1) it takes the surrounding environment of the software into consideration. Therefore, it allows identifying variability, which is caused by the environment, 2) it allows decomposing the overall software problem into subproblems, thus reducing the complexity of the problem, 3) it makes it possible to annotate problem diagrams with quality requirements, 4) it enables various model checking techniques, such as requirements interaction analysis and reconciliation [1] or quality requirements elicitation [6] due to its semi-formal structure, and 5) it supports a seamless transition from requirements analysis to architectural design (e.g. [3]).

The remainder of this paper is organized as follows. An alarm system as a running example is introduced in Sect. 2. Section 3 gives a brief overview of the OVM, problem frames, and problem-oriented requirements engineering. Section 4 describes how we extend problem frames with a notation for variability. We introduce the PREVISE method in Sect. 5. Section 6 presents related work, while Sect. 7 concludes the paper and points out suggestions for future work.

## 2 Running Example

As our running example, we have chosen an alarm system. We will not elaborate on a full alarm system, but a very small and simple one, blanking many functionalities that such a system normally embodies. An initial problem description is given as follows: the *alarm system* is installed within a defined perimeter, such as a building. In this building alarm *buttons* and *signal horns* are installed. Whenever a person in the building witnesses a critical situation such as a fire, he / she shall warn others. A *witness* can *alert* others in the building, using the alarm buttons. The *alarm is given* using the signal horn. The alarm shall be given within one second. Additionally, every *alarm raised* is forwarded to an *alarm central*. The notification is repeated every 30 seconds. The *broadcast* to the alarm central is optional as not every owner of the alarm system needs or can afford using such an alarm central. When a communication to an alarm central is established, no third party shall be able to tamper with the communication. From this small scenario, we can derive two functional, one performance and one security requirement:

**R1.** A witness can alert others in a building using the alarm buttons. The alarm is given using the signal horn.
**R2.** Every alarm raised is forwarded to an alarm central. The notification is repeated every 30 seconds.
**PR1.** The alarm shall be given within one second.
**SR1.** When a communication to an alarm central is established, no third party shall be able to tamper with the communication.

## 3 Background

In this section, we give an overview of the concepts and methods our method relies on. OVM is described in Sect. 3.1, while the problem frames approach is given in Sect. 3.2.

### 3.1 Orthogonal Variability Modeling

In SPLE, OVM describes an approach to capture a product line's variability. In contrast to other approaches, which integrate variability into existing design artifacts, OVM explicitly captures variability in distinct models. Using traceability links, elements from OVM models can be connected to arbitrary design or development artifacts or elements within these artifacts, e.g. requirements, a state within a UML state machine, or implemented classes [12].

OVM comprises a set of model elements that allow for modeling variability. The central model element is the abstract *variation point (VP)*. A VP defines a place where single products may differ.

Since an OVM model defines the variability of an entire SPL, it provides a concept to derive products. Several model elements (including VPs) support a selection concept. A single product is defined through all elements that have been selected. To indicate a choice for the developer, selectable VPs may be *optional*. In contrast, if a VP is considered essential, it is declared *mandatory*. A mandatory VP must be selected for every product.

While VPs define where products may differ, *variants* define how they differ. Variants and VPs are linked through *variability dependencies (VD)*, while a variant has to be associated with at least one VP (in turn, a VP must be associated with at least one variant). Similar to VPs, variability dependencies may be either *optional* or *mandatory*. If a VP is selected and is associated with a variant through an optional VD, this very variant may be selected. However, if the association is a mandatory one, the variant must be selected in this case.

To ensure flexibility in the product derivation, OVM offers the possibility to define *alternate choices*. An alternate choice groups a set of variants that are associated with the same VP through optional dependencies and defines a minimum and a maximum value. Within product derivation, a number of $n$ with $minimum \leq n \leq maximum$ variants have to be selected if their corresponding VP has been selected.

Since in practice relationships and interactions between variants and VPs can be observed, OVM allows for defining these relationships through *variability constraints*. Variability constraints can be set up between two variants, two VPs, or a variant and a VP. OVM provides two types of variability constraints: *requires* and *excludes*. The requires constraint is directed from a source to a target element and requires the target to be selected if the source has been selected. The excludes constraint is undirected and prevents selecting one element if the other element has been selected.

### 3.2 Problem Frames

Problem frames [10] proposed by Michael Jackson 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

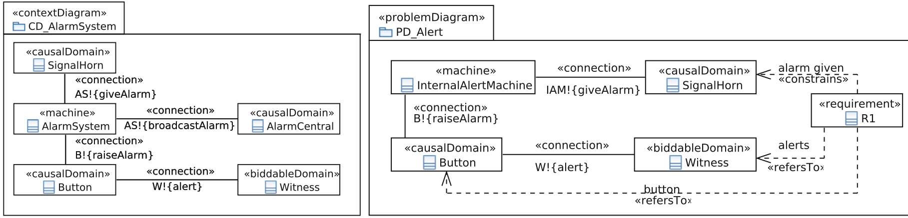


**Fig. 1.** Context diagram for the Alarm System (left) and problem diagram for R1 (right)

them, and a requirement. Domains describe entities in the environment. *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 "!". For example, the notation *W!*{*alert*} (between *Witness* and *Button*) in Fig. 1 (right) means that the phenomenon *alert* is controlled by the domain *Witness*. The software to be developed is called *machine*.

We describe problem frames using UML class diagrams, extended by a specific UML profile for problem frames (UML4PF) proposed by Hatebur and Heisel [9]. 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.

In UML4PF, requirements are a special kind of statement. When we state a requirement, we want to change something in the world with the machine to be developed. Therefore, each requirement expressed by the stereotype ≪`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 these domains with the stereotype ≪`refersTo`≫. The requirement *R1* on the right-hand side of Fig. 1 constrains the causal domain *Signal Horn*, and it refers to the causal domain *Button* and the biddable domain *Witness*.

**Problem-oriented Requirements Engineering.** Our method for problem-oriented requirements engineering involves the steps *problem context elicitation*, *functional requirements*, and *quality requirements modeling*.

The first step *problem context elicitation* aims at understanding the problem the system-to-be shall solve, and therefore understanding the environment it should influence according to the requirements. We obtain a problem description by eliciting all domains related to the problem, their relations to each other and the system-to-be. To elicit the problem context, we set up a *context diagram* consisting of the machine (system-to-be), related domains in the environment, and interfaces between these domains. The context diagram for our example is shown on the left-hand side of Fig. 1.

The second step *functional requirements modeling* is concerned with decomposing the overall problem into subproblems, which describe a certain functionality, as expressed by a set of related functional requirements. We set up *problem diagrams* representing subproblems to model functional requirements. A problem diagram consists of one submachine of the machine given in the context diagram, the relevant domains, the interfaces between these domains, and a requirement referring to and constraining problem domains. The problem diagram describing the functional requirement *R1* in our example is shown on the right-hand side of Fig. 1.

To analyze quality requirements in the software development process, they have to be addressed as early as possible in the requirement models. The functionality of the software is the core, and all quality requirements are related in some way to this core. Modeling quality requirements and associating them to the functional requirements is achieved in the step *quality requirements modeling*. We represent quality requirements as annotations in problem diagrams. For more information, see our previous work [4].

## 4 Extending Problem Frames with a Variability Notation

We extend the problem frames notation by introducing new elements for modeling variability in software product lines. We base our extension on the OVM terms. In Sect. 3.2, we briefly described the UML4PF profile, which enables us to use the problem frames notation in UML models. Our extension is a UML profile relying on the UML4PF profile.

The detailed usage of the stereotypes[2] will be explained in Sect. 5. The profile allows the creation of new kinds of diagrams and statements. The first new kind of *UML4PF diagrams* are *variability diagrams*. They capture the actual variation points. There are *requirement variability diagrams*, *domain variability diagrams*, and *phenomenon variability diagrams* as the variability can stem from requirements, domains, and phenomena. One special variability diagram is the *constraint variability diagram*, which captures constraints to variability. To the *context diagram* we add two new sub-types. First of all, a *variability context diagram*, which describes the context containing the variability. In contrast, the *product context diagram* describes the context regarding a particular product, which is defined by a *configuration*. The same distinction is made for *problem diagrams*. For problem diagrams we also have *variability problem diagrams* and *product problem diagrams*. The latter diagram is the *configuration diagram*, which describes a particular configuration for a product.

The first new statement introduced is the *variation point*. One can distinguish between *mandatory variation point* and *optional variation point*. Related to variation points are *variants*, which can represent an *optional variation* or a *mandatory variation*. A variation point indicates by its *min* and *max* properties how many of the variants have to be chosen for the variation point. The type of variation relation is indicated by a *variation dependency*. Variants and variation points can be related by a *constraint dependency*. The relation can be an *excludes* or a *requires* dependency.

[2] The meta-model is available in `http://www.geneda.org/pub/TechnicalReportPREVISE.pdf`

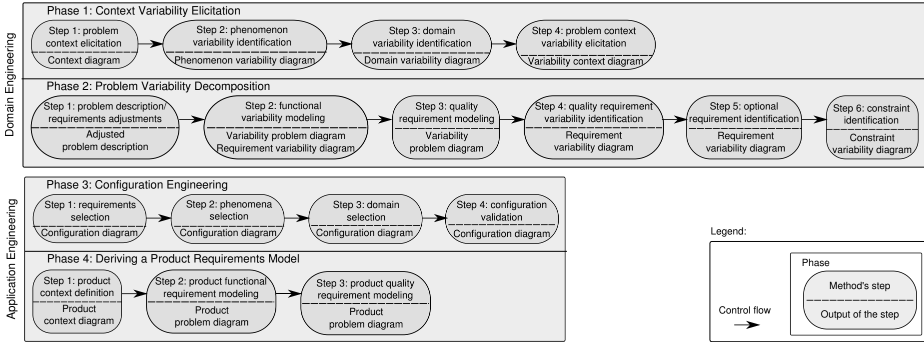


**Fig. 2.** PREVISE method and the outputs of each step

# 5 Problem-Oriented SPL Requirements Engineering Method

In this section, we present the PREVISE method, which defines the activities in the first phase of the domain and the application engineering, namely the requirements engineering. We describe how we extend our current problem-oriented requirements engineering method described in Sect. 3.2 for SPL. In Sect. 5.1, we describe the phases of domain engineering and the subsequent steps, in which we create a requirement model for the SPL. Then, we describe the phases of application engineering, in which we derive a concrete SPL product from the SPL requirement model in Sect. 5.2. Figure 2 shows an overview of the steps to be conducted in the PREVISE method and the corresponding outputs.

## 5.1 Product Line Requirement Model Creation

**Phase 1: Context Variability Elicitation.** In this phase, the context of the system-to-be is analyzed, and variation points in the environment of the machine are identified.

Step 1 - Problem context elicitation. For our method, it is not necessary to have a problem description, which already includes variability. Instead, one can start by giving a problem description for one possible product. The variability is identified and added in later steps. Hence, in step one we derive a context diagram from the problem description as proposed by Jackson [10]. *The context diagram is shown on the left-hand side of Fig. 1 and was already explained in Sect. 2.*

Step 2 - Phenomenon variability identification. In this step, every phenomenon of the context diagram has to be analyzed for two things. First, if the phenomenon at hand is a generic one, which has more than one possible concrete instances. For the case that it is not a generic one, there may be other alternatives for the phenomenon at hand. If one of these two cases holds, the generic phenomenon has to be added as a variation point and the concrete phenomena as variants. Additionally, one has to model if a variant or variation point is optional or not. Second, if a phenomenon is shared using a dedicated connection domain, this connection domain has to be added to the context diagram. *For our example, the phenomenon* alert *turns out to be a generic phenomenon, which*

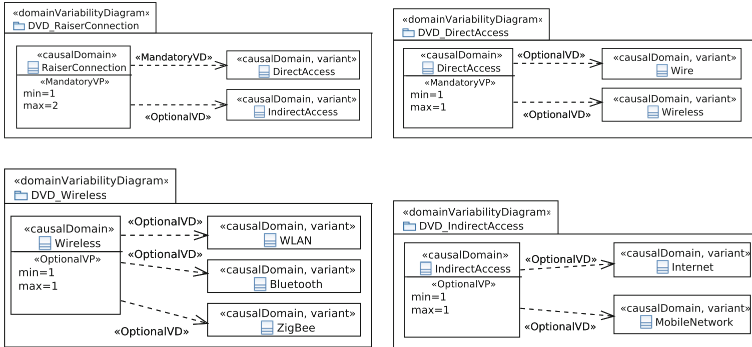


**Fig. 3.** Domain Variability Diagrams

*has two variants. First, one can* push *something to give the alert. Second, one can* shout *to give an alarm, which is a more advanced option for an alarm system.*

Step 3 - Domain variability identification. Similar to the phenomena, the domains of the context diagram have to be checked for variation points and variants. Note that it can occur that one variant is a variation point as the variant can be further refined.

*One example for domain variability is shown in Fig. 3. The starting domain for this variability is the causal domain* wire*. It connects the alarm raiser with the machine. The domain wire is abstracted to the causal domain* raiser connection*, which is a mandatory variation point. Variants for the raiser connection are a* direct access *connection, which is mandatory or an* indirect access *connection, which is optional. For direct access, one variant is the wire. The other variant is a* wireless *solution, which can be a* WLAN*, a* bluetooth*, or a* ZigBee *connection. The indirect access can be realized via* internet *or a* mobile network*.*

Step 4 - Problem context variability elicitation. This step uses the context diagram and the domain variability diagrams to generate the variability context diagram. The variability context diagram enables us not only to elicit all domains related to the problem to be solved, but also to capture, which domains represent variability and which ones commonality. The structure of the variability context diagram is similar to the *context diagram* from step 1. It differs from it in the way that we represent variation points for the problem domains and phenomena, which involve variability. The variability context diagram represents a context diagram for the SPL. Note that the variability context diagram can be automatically generated using the context diagram and the domain variability diagrams.

*Figure 4 on the left-hand side shows the resulting variability context diagram for our example. The domains* alarm system *and* witness *are directly taken from the context diagram as they are not variable. The signal horn is replaced by the variation point* notifier*. The alarm button is replaced by the variation point* raiser*. Additionally, the connection domains and their abstract variation points* raiser connection*,* notifier connection*, and* alarm central connection *are added to the variability context diagram.*

**Phase 2: Problem Variability Decomposition.** In this phase, the overall problem is decomposed into smaller subproblems according to the requirements of the system-to-be. The quality and functional requirements are adjusted in a way that they reflect the variability of the problem.

Step 1 - Problem description/ requirements adjustment. In this step, the textual requirements of the machine are derived from the problem description. As the initial problem description does not contain the variability identified in phase one, the textual description of the requirements has to be adjusted. *In Sect. 2 we already derived the textual requirements from the initial problem description. Now the wording has to be adjusted to the variability context diagram. For example, requirement R1 changes to "A witness can* [alert] *others in a building using* [raisers]. *The alarm is given using* [notifiers]."

Step 2 - Functional variability modeling. This step is concerned with decomposing the overall problem into subproblems, which accommodate variability. Each functional requirement has to be modeled as a problem diagram. Whenever the problem diagram contains at least one variation point, the requirement is variable, too. But variability in a requirement cannot only stem from phenomena or domains, which are variable. Sometimes requirements contain further variation points, which do not show up in the structure of a problem diagram. One reason might be a variability in behavior, for example in the sequence of phenomena. Hence, each requirement has to be checked for such variations not visible in the problem diagrams. Such variabilities are represented by a *requirement variability diagram (RVD)*, which represents the requirement as variation point and its alternatives as variants. *For our example, the functional requirement R2 contains further variability. The repetition of the alarm notification is optional. The according requirement variability diagram is shown in Fig. 4 on the right-hand side. Note that requirement R2.1 contains further variability regarding the time span between the repetitions. Figure 5 on the left-hand side shows the variability problem diagram for requirement R2.*

Step 3 - Quality requirement modeling. This step is concerned with annotating quality requirements, which complement functional requirements. In contrast to functional

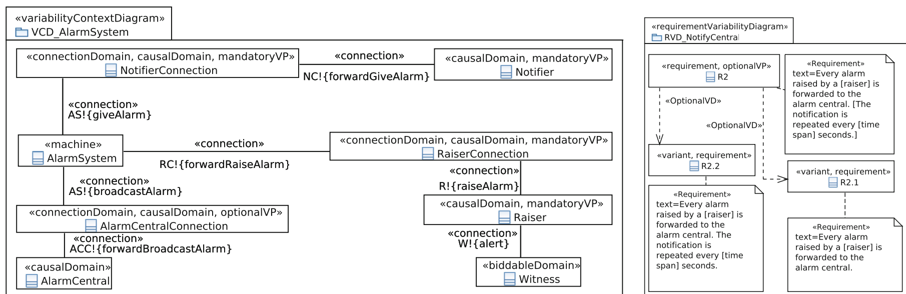


**Fig. 4.** Variability Context Diagram for the Alarm System (left) & Requirement Variability Diagram for R2 (right)

requirements, quality requirements are not modeled as problem diagrams on their own. Instead, they augment existing functional requirements.

Step 4 - Quality requirement variability identification. Variability in quality requirements can be caused when making trade-offs among quality requirements of different types. Such requirements are subject to interactions. Interactions among quality requirements can be detected by applying step 1 of the QuaRO method proposed in our previous work [1]. To resolve interactions, we generate requirement alternatives by relaxing the original requirement. To obtain such a variability in quality requirements, we apply the second step of the QuaRO method. The generated quality requirement alternatives provide variants for the original requirement. The requirement variability diagrams have to be updated according to the results of the QuaRO method. Sometimes, quality requirements introduce new domains, e.g., an attacker for security, and phenomena. Thus, one has to check these domains and phenomena for variability, too. *For our example, we have the security requirement SR1. It complements the functional requirement R2. It adds the biddable domain* attacker. *The domain attacker is a variation point as there can be different attackers distinguished by their abilities (see [1] for more information).*

Step 5 - Optional requirement identification. In this step, one has to identify the requirements, which are optional. They have to be modeled as optional variation point. *For the alarm system, the notification of the alarm central is optional, which is already reflected in Fig. 4 on the right-hand side, as* R2 *is annotated as an optional variation point (*optionalVP*).*

Step 6 - Constraint identification. This step is concerned with identifying constraint dependencies among requirement, phenomena, and domain variants. Dependencies caused by quality requirements interactions are identified as a result of the first step of the QuaRO method [1]. For functional requirements, one can use the RIT (Requirements Interaction Tables) as proposed in previous work [2]. Other kind of dependencies have to be checked manually. We distinguish between two types of dependencies, namely *requires* in which one variant or variation point requires another variant or variation point for a valid configuration, and *excludes* in which one variant or variation point is not allowed together with another variant or variation point in a valid configuration. *For example, the phenomenon* shoutToAlert *requires a* voice sensor. *The according constraint variability diagram is shown in Fig. 5 on the right-hand side.*

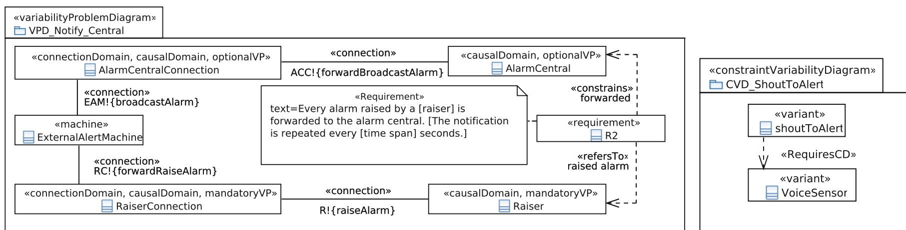


**Fig. 5.** Variability Problem Diagram for R2 (left) & Constraint Variability Diagram for alert to shout (right)

## 5.2 Deriving a Concrete Product Requirement Model

To derive requirements for a concrete SPL product, we make use of the artifacts generated in domain engineering. The aim of the application engineering is to get a coherent subset of requirements for a particular product from the overall set of requirements containing the variability. The application engineering is divided into two phases, which are explained in the following. Note that for the product engineering we do not elaborate the example for reasons of space. The example is explained in the accompanying technical report[3]

**Phase 3: Configuration engineering.** In this phase the configuration for the concrete product is selected. The following steps can be supported by a feature diagram and OVM diagrams derived from the domain requirements model. Note that this phase can be repetead to define more than one configuration. Step 1 - Requirements selection: The first step towards a configuration is to select the desired requirements among all optional requirements. This selection may reduce the phenomena and domains to select from in the next steps. The reason is that phenomena and domains, which are only bound to optional requirements that are not selected can be left out. For all requirements, which represent an optional variation point, one has to decide whether to include the requirement or not. Next, one has to select a variant for all requirements, which represent a variation point and which are included in the desired set of requirements. The desired set contains the selected optional and all mandatory requirements. The selected variants have to be documented in a configuration diagram. Step 2 - Phenomena selection: The second step is to select the variants for all phenomena, which are variation points. The reason for going first for the phenomena is that phenomena are the starting point of the interaction of end users with the system-to-be. Thus, we have the end user in focus. Additionally, the selected phenomena often constrain the set of domains to be chosen from. In many cases, specific phenomena exclude or require specific domains. Step 3 - Domain selection: In this step, one has to select for all domain variation points the according desired variants. Step 4 - Configuration validation: Last, one has to check if the constraints defined in the constraint variability diagrams are all satisfied. Additionally, one has to check whether the variation dependencies given by the variation diagrams and the min / max constraints of the variation points are satisfied.

**Phase 4: Deriving a Product Requirements Model.** In this phase, the concrete product requirements model is derived based on a given configuration. Note that one can define more than one configuration at a time and derive product requirement models for them. Step 1 - Product context definition: This step is concerned with deriving a *product context diagram* for a concrete product. To this end, we make use of a configuration diagram that defines, which requirement variants have to be achieved by the concrete product. Then, we derive the concrete SPL context diagram from the variability context diagram, replacing all variation points by the variants defined by the configuration. Variation points, which are not addressed by a variant in the configuration are removed. Step 2 - Product functional requirement modeling: In this step, we derive *product problem diagrams* for a concrete product. By means of the configuration we know which functional requirements have to be involved in the requirement models for the concrete SPL product. We use the *variability problem diagrams* for deriving *product problem diagrams*. The activities to

[3] `http://www.geneda.org/pub/TechnicalReportPREVISE.pdf`

be performed are like the ones for step 1. One additional step is the textual adjustment of the requirements. Step 3 - Product quality requirement modeling: For the product quality requirement modeling one has to perform the same activities as given for step 2.

## 6 Related Work

There exist several methods connecting SPL with requirements engineering approaches. We focus on methods, which connect problem frames and variability. Zuo et al. [14] introduce an extension of the problem frames notation that provides support for product line engineering. The extension for problem frames only supports variability in requirements and machines. In contrast to the PREVISE method, the authors do not consider the variability, which can be caused by domains and phenomena. Furthermore, the authors only provide a notation for domain engineering.

Ali et al. [5] propose a vision for dealing with variability in requirements caused by the environment. The authors propose an idea for a framework, which relates the three requirements engineering methods goal models, feature diagrams, and problem frames to the environmental context in order to use context information for product derivation. In contrast to PREVISE, it does not pay attention to the variability caused by the requirements and relies on preliminary knowledge about variability.

Variability, which emerges due to changes in the environment (contextual variability), is discussed by Salifu et al. [13]. The authors first set up problem diagrams and then identify a set of variables representing the contextual variations. Using the contextual variables, variant problem diagrams are derived. In their work, the authors provide no systematic approach on how to identify contextual variations in the environment and Application engineering is not considered.

An approach for integrating SPLE and the problem frame concept is proposed by Dao et al. [8]. The starting point is a feature model, which is mapped to a problem frames model to elicit functional requirements and domain assumptions. To take quality requirements into account, a goal model is adopted. The three different notations feature models, problem frames, and goal models are used, which might cause consistency problems among different models. In contrast, we provide one single model, which enables consistency checking and tool support.

Similar to our method, the approach proposed by Classen et al. [7] considers variability in requirements and phenomena. However, the authors do not treat variability in domains. Furthermore, quality requirements are not considered.

## 7 Conclusion

In this paper, we have presented an extension of the problem frames notation to enable variability modeling. The notation extension for variability is accompanied by a method called PREVISE for discovering variability, modeling variability, and deriving products from the variability models. The contributions of this paper are providing 1) an OVM-based notation for adding variability to requirements, which are expressed in the problem frames notation (see Sect. 4), 2) a method, which can be conducted without any previous knowledge about variability, 3) a structured method for conducting domain engineering

in the requirements phase, which includes (see Sect. 5.1) discovering and modeling variability, 4) a structured method for conducting application engineering in the requirements phase, which includes (see Sect. 5.2) setting up configurations for products and deriving requirement models for products according to the configurations. For the future, we plan to implement and improve the tool support.[4] We also plan to integrate PREVISE and QuaRO. We will also integrate PREVISE into the GenEDA method, which will provide the software engineer with a method, which closely integrates requirements engineering, architecture and design, and patterns. Hence, the variability will not only be reflected in the requirements, but will also be integrated in the architecture generation. For the validation of the method , we will apply the method to a bigger case study.

## References


1. Alebrahim, A., Choppy, C., Faßbender, S., Heisel, M.: Optimizing functional and quality requirements according to stakeholders' goals. In: Mistrik, I. (ed.) Relating System Quality and Software Architecture. Springer ( to appear, 2014)
2. Alebrahim, A., Faßbender, S., Heisel, M., Meis, R.: Problem-Based Requirements Interaction Analysis. In: Salinesi, C., van de Weerd, I. (eds.) REFSQ 2014. LNCS, vol. 8396, pp. 200–215. Springer, Heidelberg (2014)
3. Alebrahim, A., Hatebur, D., Heisel, M.: A method to derive software architectures from quality requirements. In: APSEC, pp. 322–330. IEEE Computer Society (2011)
4. Alebrahim, A., Hatebur, D., Heisel, M.: Towards systematic integration of quality requirements into software architecture. In: Crnkovic, I., Gruhn, V., Book, M. (eds.) ECSA 2011. LNCS, vol. 6903, pp. 17–25. Springer, Heidelberg (2011)
5. Ali, R., Yu, Y., Chitchyan, R., Nhlabatsi, A., Giorgini, P.: Towards a Unified Framework for Contextual Variability in Requirements. In: IWSPM 2009, pp. 31–34. IEEE (2009)
6. Beckers, K., Faßbender, S., Heisel, M., Meis, R.: A problem-based approach for computer-aided privacy threat identification. In: Preneel, B., Ikonomou, D. (eds.) APF 2012. LNCS, vol. 8319, pp. 1–16. Springer, Heidelberg (2014)
7. Classen, A., Heymans, P., Laney, R.C., Nuseibeh, B., Tun, T.T.: On the Structure of Problem Variability: From Feature Diagrams to Problem Frames. In: VaMoS 2007, pp. 109–117 (2007)
8. Dao, T.M., Lee, H., Kang, K.C.: Problem frames-based approach to achieving quality attributes in software product line engineering. In: SPLC 2011, pp. 175–180. IEEE (2011)
9. Hatebur, D., Heisel, M.: A UML profile for requirements analysis of dependable software. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 317–331. Springer, Heidelberg (2010)
10. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley (2001)
11. Konersmann, M., Alebrahim, A., Heisel, M., Goedicke, M., Kersten, B.: Deriving Quality-based Architecture Alternatives with Patterns. In: SE. LNI, vol. 198, pp. 71–82. GI (2012)
12. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques, pp. 1–467. Springer (2005)
13. Salifu, M., Nuseibeh, B., Rapanotti, L., Tun, T.T.: Using Problem Descriptions to Represent Variabilities For Context-Aware Applications. In: VaMoS 2007, pp. 149–156 (2007)
14. Zuo, H., Mannion, M., Sellier, D., Foley, R.: An Extension of Problem Frame Notation for Software Product Lines. In: APSEC 2005, pp. 499–505. IEEE (2005)


[4] For more details see:
`http://www.geneda.org/pub/TechnicalReportPREVISE.pdf`