Technical report

Nr.: FIN-04-2014

Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines

Reimar Schröter

Arbeitsgruppe Datenbanken und Software Engineering

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-04-2014

Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines

Reimar Schröter

Arbeitsgruppe Datenbanken und Software Engineering

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Reimar Schröter
Postfach 4120
39016 Magdeburg
E-Mail: reimar.schroeter@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 15.10.2014

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Using Multi-Level Interfaces to Improve Analyses of Multi Product Lines

Reimar Schröter
University of Magdeburg, Germany
rschroet@ovgu.de

ABSTRACT

Software product lines (SPLs) enable an efficient development of similar programs based on a common code base. Although the number of SPLs in practice increases, the development and maintenance of an SPL is still challenging. In detail, a developer has to consider a huge amount of variability in each step of the development cycle that can become unmanageable. Therefore, *multi software product lines (MPLs)* were introduced that divide this variability in smaller more manageable parts. However, as result, the dependency between these SPLs becomes challenging as well as the analysis that should ensure the correctness of the whole systems. We propose multi-level interfaces that consist of a set of interfaces that encapsulate the modeling level, the implementation level as well as the behavioral level between cooperating software product lines. Using these interfaces, we want to investigate whether it is possible to simplify analyses according to these different levels so that we do not need the knowledge about the whole MPL. Preliminary results show that multi-level interfaces facilitate the domain comprehension for the developer, and reduce the complexity of product-line analyses. We present an overview of hypotheses and research methods as well as a concrete working plan to investigate multi-level interfaces in the software product-line lifecycle.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—Modules and interfaces

General Terms

Design, Reliability

Keywords

Software product lines, multi product lines, modularity, interfaces

1. INTRODUCTION AND MOTIVATION

The reuse of software artifacts is an important concept to reduce the effort through product development. Since the beginning of the software-engineering age, many concepts and mechanism were proposed to reuse software artifacts. One of the most promising concepts for the reuse of software artifacts are *software product lines (SPLs)* that allow us to generate tailored products based on a common code base [10, 3]. The commonalities and differences between these products are described by features that can be selected to generate a product according to the special requirements of a stakeholder [18].

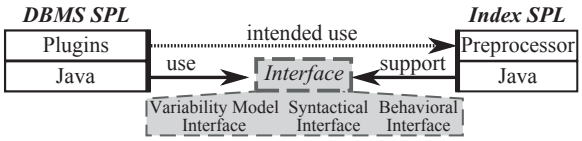


Figure 1: Reuse of an index-structure SPL in a variable database management system (adapted version of [38]).

Several techniques and tools exist that allow a developer to design, develop, and analyze SPLs. Nevertheless, industrial SPLs tend to be very large and, thus, the developer has to handle thousands of features (e.g., more than 11 000 features in the linux kernel [40]). This huge amount of variability is still challenging for each development step of an SPL. One way to handle this variability is based on a divide and conquer strategy in which the problem is described in a set of dependent SPLs. The result is a *multi software product line (MPL)*, which is an arbitrary composition of SPLs [33].

Let us consider a small example to clarify the main ideas of MPLs. In Figure 1, we depict the dependency of an SPL that represents a *database management system (DBMS)* to an SPL for *index structures (Index)*. We call this resulting system of dependent SPLs an MPL. In detail, the *DBMS* is a variable system based on plugins that intends to reuse parts of a variable *Index* to optimize the performance of the data access. By contrast to the *DBMS*, the SPL *Index* is implemented based on preprocessors. If we want to combine both SPLs, we have to ensure that each resulting product of the DBMS works correct and use a tailored *Index* according to the specific DMBS requirements. Thus, the complete system is high variable (cf. if we consider a huge set of possible variable *Indexes* that we can use in the SPL *DBMS*) and it is difficult to describe the dependencies between these SPLs.

In previous research, the problem of the overwhelmed variability was mainly addressed according to one development step of SPLs (i.e., variability modeling, implementation). In the following, we classify these approaches as *local approaches*. In general, in these development steps exist a direct dependency (see Figure 1, dashed arrow) between the involved SPLs (i.e., on the modeling, implementation and on the level of runtime behavior). Therefore, if we want to

analyze the MPL (e.g., consistency of the variability model), we have to consider all dependent SPLs. This results in the same complexity as for one huge SPL that realize the whole variability. Thus, the question arise how can we efficiently analyze the correctness of MPLs? Is it possible to find a general concept that we can use in each development step?

By contrast to the existing approaches to handle variability, we focus on an approach that can be used in each development step of MPLs and supports modular analyses. Therefore, we suggest using the well-known concept of interfaces that was applied with success in other areas of software engineering, such as module systems and *object-oriented programming (OOP)*. In detail, we introduce multi-level interfaces for MPLs that detach the direct dependency between SPLs. In detail, the multi-level interfaces consist of a *variability-model interface*, a *syntactical interface*, and a *behavioral interface*. In Figure 1, we use these interfaces between the SPL *DBMS* and the SPL *Index*. Here, the SPL *Index* supports an interface for each development step that can be used without further details of the underlying structure behind it (e.g., implementation or modeling details). Furthermore, these interfaces depend on each other so that, for instance, only syntactical artifacts are represented that rely on features represented in the variability-model interface. In this thesis, we investigate the usage of our interface concept to reduce the complexity of analysis processes on each development step. For instance, we are interested in modular automated analyses of variability models, an easy manual analysis of code artifacts by the developer for reuse, and reduced complexity of analysis techniques for detection of runtime violations of MPLs.

In the following, we present an overview of state-of-the-art techniques for the development and analysis of SPLs and MPLs including their advantages and drawbacks. Afterwards, we present the research issues and hypothesis for the thesis with respect to existing approaches. We present our general approach of interfaces and give an insight into our proposed interfaces. Based on this description, we present our research methodology and give an overview of completed as well as open tasks of our working plan.

2. STATE-OF-THE-ART

In the following, we present existing work that tackles the problem of the overwhelmed variability only partially. Therefore, we divide the section according to the SPL's development steps in which we plan to introduce our concept of multi-level interfaces. Afterwards, we give an overview about existing analyses in MPLs.

Variability Modeling

Feature models are the most commonly used concept to describe the dependencies between features in an SPL [5]. Feature models were introduced by Kang et al. [18] and were extended by several authors. Czarnecki et al. presents an overview of these extensions that, for instance, introduce cardinalities for feature groups and attributes specifying further details of features [11].

To reduce the complexity of large feature models and to improve the manageability, several authors propose feature-model views [26, 17, 37]. In general, these views present a tailored excerpt of the relevant features according to the requirements of the stakeholder. Reiser and Weber introduce multi-level feature trees to improve the manageability of highly complex product families by feature and feature-model references [31]. Using permission attributes, the user can define allowed changes of a feature model compared to a referred feature model.

In general, these techniques allow us to get an improved overview of the basic elements of variability models, but these concepts do not reduce the complexity of analyses. In particular, it is not possible to locally analyze the correctness of each model part.

Product-Line Implementation

The concepts to implement software product lines can be divided in two parts, composition-based approaches and annotation-based approaches [3]. Whereas annotation-based approaches (e.g., C-preprocessor) use annotations to describe the implementation variability of an SPL, composition-based approaches locally separate the different programming artifacts according to a feature.

Annotation-based approaches are well-known concepts in industrial SPLs [3]. However, with increasing variability it is challenging to maintain the code artifacts and to comprehend the code. Furthermore, the annotations can be very fine-grained, so that in the worst case only one additional character is added by an annotation. Therefore, the fine-graininess is also curse and results in the "ifdef hell" [15, 25]. Approaches such as CIDE, which uses colors as markups for different features, improve the maintainability of #ifdef code [19]. Based on these techniques, Kästner et al. present several views that can be used as support for the SPL implementation [21].

Besides the annotation-based approaches, several composition-based approaches were proposed, such as *feature-oriented programming (FOP)* [29, 6], *aspect-oriented programming* [23], or *delta-oriented programming* [35]. These techniques allow a developer to define all code artifacts according to one feature locally separated (e.g., in feature modules).

The mentioned techniques to implement SPLs ease the implementation of features. However, the implementation is nonetheless challenging because it is not clear which code artifacts of other features can be reused in the currently developed or maintained code artifact. Approaches that introduce views to the code (e.g., realization view for preprocessors [21]) or restrict the code access (e.g., special access modifiers for FOP [4]) decrease the set of usable implementation artifacts. Nevertheless, these approaches still present elements that cannot be reused in the current implemented feature.

Multi Software Product Line

MPLs are a general concept to tackle the variability problem. According to Holl et al., an MPL is "a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system" [16]. However, the proposed concepts according to MPLs address in general the modeling step and do not present concepts to develop MPLs that can be used on each development step [16].

Using the concept of MPLs for the modeling step, it is possible to decompose feature models in smaller parts and to combine the resulting models using composition-based approaches [8, 1, 34]. Therefore, several variability-model languages were introduced, such as VELVET [34], TVL [9],

and FAMILIAR [2]. Besides these languages, Eichelberger and Schmid present an overview about existing languages and describe their capabilities [14].

Similar to the concepts of MPLs, module systems divide the large-scale problem into smaller pieces and allow us to combine the components in a flexible way. The Koala component model defines *provide* and *require interfaces* for the description of valid combinations of components [45]. Furthermore, we can define Koala components in a hierarchical fashion so that it is possible to build more complex components and hide internal complexity. In addition, Reiser et al. use feature models to describe the inner variability of such hierarchical components and propose a set of pattern for the variability specification [30]. By contrast to these concepts, Kästner et al. present a module system with a variable interface on code as well as modeling level [20]. This concept is more flexible compared to Koala components and allows us to present internal variability of the module to the interface. Thus, a module is similar to an SPL and the composition of multiple modules is similar to a composition of multiple SPLs that describe an MPL. However, the approach does not take the behavioral level into account.

Further techniques investigate the flexible evolution and configuration of MPLs. In detail, Dhungana et al. present combinable model fragments with public and private model elements [12]. It is possible to merge these fragments semi-automatic and save a merge history to automate the merge process in future. Furthermore, several authors investigate the configuration of MPLs. For instance, Rosenmüller and Siegmund propose a stepwise configuration from the high-level SPLs to the low-level SPLs to reduce unnecessary number of decisions [33]. By contrast, Dhungana et al. introduce the tool Invar which is a web service system that allows us to combine different variability modeling approaches and their configurations [13]. As result, vendors and suppliers can use their own modeling technique that is connected to Invar to configure the final MPL's product.

Product-Line Analyses

Analysis techniques vary among the different development steps of SPLs. Thus, there are a set of specialized analysis techniques for the modeling, implementation, and the behavioral level to ensure the correct functionality of each SPL's product. In the following, we give a short overview about existing approaches and their basic ideas that we plan to improve by our modularization concept according to each development step.

Benavides et al. present an overview of existing automated analyses for feature models [7], such as the analyses *dead features*. The knowledge about the existence of these features is very important because a programmer can neglect these features for the implementation. By contrast, false-optional features are included in each product, in which the parent is included and, thus, the programmer has to pay attention on this feature during the implementation step. Approaches, such as views, help to ease the modeling effort, but fail to modularize the analysis step of the whole SPL's feature model. Furthermore, it is not clear whether it is possible to modularize feature-model analyses.

Besides analysis techniques according to variability models, several approaches were published that aim to ensure the correctness of a specific product-line implementation technique [41]. For instance, TypeChef is a variability-aware type checker for preprocessors [22] and Fuji is a compiler and type checker for the composition-based approach FOP [4]. These approaches help us to detect errors in SPL's products in a more efficient way, but it is not possible to prevent errors during the implementation. In detail, a developer is interested in reusable code artifacts during the implementation (e.g., a method introduced in another feature), whereas current techniques check the syntactical correctness only after the implementation. Therefore, we need further code-analysis techniques to infer that we can reuse a specific code artifact.

Furthermore, Ribeiro et al. propose emergent interfaces to ease the maintenance of annotation-based SPLs [32]. These interfaces were created by means of a dataflow analysis and present *requires* and *provides* information related to a currently maintained code artifact. Therefore, the interfaces were created on demand to the current selected code artifact. This approach helps to prevent errors during the code maintenance but does not present safely accessible code artifacts. Therefore, the developer has to search for reusable artifacts for himself.

Design by contract is a well-known technique in object oriented programming that facilitates checks to ensure correct behavior of a program using pre-conditions (i.e., expected method input) and post-conditions (i.e., expected method output) [28]. Several authors proposed adapted approaches of design by contract for SPLs. For instance, Thüm et al. propose concepts to apply design by contract to FOP [44]. This allows developers of SPLs to verify the correct behavior of each SPL's product using, for instance, the theorem prover KeY [43]. But, it is an open question how these approaches scale to large systems and whether it is possible to modularize these analyses in MPLs. For further details, we refer to a survey on general analysis strategies for SPLs and on an overview of related tool support [41, 27].

3. RESEARCH ISSUES AND HYPOTHESES

According to the proposed work, especially for the development of multi software product lines (MPLs), we identified a lack of modularized analysis strategies for MPLs. Thus, we consider the research on modular analysis strategies that ensure the correct functionality of MPLs as main research issue of the thesis. Thus, the management of the variability in large-scale and ultra large-scale systems consisting of thousands of features is challenging. Therefore, we need analysis strategies to ensure the correctness of MPLs and support their development.

In general, we want to tackle the problem of the variability management with a general approach that is applicable on each development step of MPLs and allows a simplification of the analysis processes. Therefore, we are interested in an approach supporting a modular analysis so that we can efficiently check the correctness of the SPL on each development step.

We propose multi-level interfaces according to the different development steps to tackle the overall problem and to reach our objectives. Interfaces are a well-known technique in module systems and object-oriented programming for the modularization of artifacts. By default, interfaces facilitate information hiding. Therefore, we assume that it is possible to reduce the complexity of the analysis processes, because we have to consider only the dependencies to the specific interface.

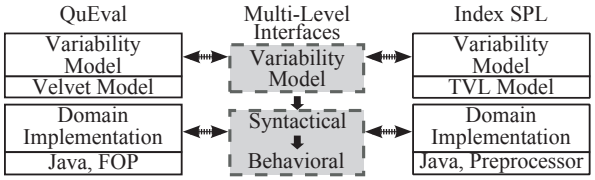


Figure 2: Interfaces for the reuse of SPLs in MPLs (adapted version of [38]).

The proposed multi-level interfaces concept includes dependent interfaces for variability models, the programming interface, and the runtime behavior of SPLs. In general, we expect a significant improvement of existing analyses on each development step that we can improve the performance of the specific analyses. For each development step, we formulate a hypothesis that we want to investigate during the thesis.

H 1. *The variability-model interface enables a performance improvement of automated analysis on variability models in evolving MPLs.*

H 2. *The syntactical product-line interface helps to detect reusable code artifacts and reduces the development time compared to state-of-the-art techniques.*

H 3. *The behavioral product-line interface based on design by contract enables a time-efficient modular analysis to detect violations in MPLs using verification techniques.*

Different strategies are necessary to investigate each hypothesis. In Section 5, we describe the strategies according to our research methodology.

4. OVERVIEW OF THE MULTI-LEVEL INTERFACE CONCEPT

By contrast to state-of-the-art approaches that mainly focus on a local approach for one specific development step to manage variability, we propose a general approach using the concept of interfaces to ease the analysis of MPLs. In detail, we propose multi-level interfaces and expected benefits (e.g., modular analysis) that we want to reach in each development step of MPLs [38]. For the thesis, we plan to refine our initial ideas so that we reach our overall goal of a modular analysis on each development step of an MPL. In the following, we give an insight of our main concept of multi-level interfaces and present one of our case studies for motivation purposes.

Case Study

We intend to use a case study to support the conceptual description of the thesis and to investigate the complete approach of multi-level interfaces. Similar to our initial example of the SPL *DBMS* and SPL *Index*, we identified our evaluation framework QuEval [36] as a possible subject. QuEval itself is a framework that allows us to identify an optimal index for a special use case in a DMBS (e.g., a concrete data distribution to access data as fast as possible). QuEval uses different products of an index as input and searches for an optimal solution. In our previous work, we emphasize that it is necessary to reorganize the framework *QuEval* and the *Index* as dependent SPLs so that it is possible to address and evaluate different index implementations [24]. For instance, if the input data of the use case is based on double values, the framework itself, all index implementations, and other components have to support double values. This affects all development steps so that the correctness of the dependencies in the variability model (i.e., selection of the specific feature), the implementation (i.e., method calls with different data types) and the runtime behavior has to be ensured (e.g., support of `null` values).

The framework QuEval including all dependent index implementations is complex and it is hard to ensure a correct combination of all features. Furthermore, if we change an SPL *Index* or if we add a new one, the whole MPL with all features must be checked again for correctness. Therefore, we introduce our multi-level interfaces to ease the development and to reduce the analysis complexity.

Multi-Level Interfaces

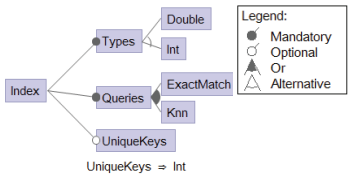
In Figure 2, we illustrate our interface concept using the dependencies between SPL *QuEval* and SPL *Index*, which supports our multi-level interfaces that introduce interfaces on each development step. Based on these interfaces, we can evaluate the dependency to the SPL *Index* without the knowledge of the specific instantiated index implementation. In detail, we introduce the concept of multi-level interfaces and define the (1) variability-model interface, (2) syntactical product-line interface, and (3) behavioral product-line interface. Using these interfaces, we aim to modularize the dependencies between SPLs, such as the dependencies of *QuEval* to the *Index* [24]. In Figure 3, we give a detailed overview of each interface according to an SPL *Index* (this is an adapted example of [38]).

As illustrated in our previous work, the variability-model interface is a variability model itself [38], which we plan to use as an agreement between the involved SPLs. For instance, we define a new variability model as interface according to the variability model of the SPL *Index*. We consider the variability-model interface as specialized variability model of the *Index*. This means that the set of valid configurations of the interface is a subset of all valid configurations according to the variability-model of the SPL *Index*. Therefore, the original variability model supports range-queries, but this feature is not available in the resulting variability-model interface (see Figure 3) because it is not provided by each SPL *Index*. Furthermore, the complexity of the resulting variability model of the SPL *Index* is reduced and, thus, the combination with the variability model of SPL *QuEval* is also reduced. Therefore, we expect a significant performance improvement of analyses if we use dependencies to the variability-model interface instead of direct dependencies to the original variability model of SPL *Index*.

The syntactical product-line interface is an application programming interface with variability information [38]. In detail, the interface consists of classes, methods, and fields that are supported by different configurations of the SPL *Index*. For instance, the availability of each member (e.g., a method) depends on the configuration of the SPL *Index* and can be filtered accordingly. Furthermore, the syntactical product-line interface is based on the variability-model interface. Thus, we only use the existing features of the variability-model interface and collect the available members (see Figure 3). All other variability (i.e., other features

that are not included in the variability model) is hidden for the SPL that wants to reuse the functionality. For instance, the SPL *QuEval* has no knowledge about the implementation details of an SPL *Index* and knows only the variability according to the variability-model interface. This means, we present all method and field signatures with the information in which feature the members are defined. All other methods that exist in the underlying SPL *Index* (e.g., according to range queries) are not included. Furthermore, we assume that a variability-aware programming interface can be used to support the manual analysis of the source code by the developer. As result, it is easier for the developer to comprehend the source code and, thus, to develop the SPL in a more efficient way.

The behavioral product-line interface is the third interface of our multi-level interface approach. This interface is based on the syntactical product-line interface as well as the variability-model interface and specifies the behavior of each available method [38]. Therefore, we propose to use the methodology of design by contract that we plan to extend to fulfill the variability requirements. In Figure 3, we present a small example. If we want to insert a point and the feature *UniqueKeys* is selected, the method has to check that the point is not already inserted in the point-container `cont`. If the point is unique or the feature *UniqueKeys* is not selected, the method has to return `true` and otherwise `false`. Using the behavioral product-line interface, we assume that it is possible to enable a modular analysis that detects runtime violations more efficiently. Therefore, we want to investigate the application of the behavioral product-line interface with theorem proving.



Variability-model interface

```
1 interface IIndex{
2   //Features: Int, UniqueKeys
3   ArrayList<int[]> cont;
4   //Features: Int, UniqueKeys
5   boolean insert(int[] point);
6   //Features: Double
7   boolean insert(double[] point); /* ... */
8 }
```

Syntactical product-line interface

```
9  interface IIndex{
10   ArrayList<int[]> cont; //Point container
11   //@ requires point != null
12   //@ if(UniqueKeys)
13   //@  ensures (exist(point) => \result = false)
14   //@  && cont.size() == \old(cont.size())
15   //@ else
16   //@  ensures cont.size() == \old(cont.size())+1
17   //@  && \result = true
18   boolean insert(int[] point); /* ... */
19 }
```

Behavioral product-line interface

Figure 3: Multi-level interfaces (adapted version of [38]).

5. RESEARCH METHODOLOGY

In this section, we present the main research methods that we plan to utilize in the PhD thesis to investigate our hypotheses. Afterwards, we discuss respective threats to validity.

Research Methods

In general, we use our presented hypotheses of Section 3 as guideline for our research procedure. Because of the various targets according to the hypotheses, we need different strategies to validate each hypothesis. We use different kinds of evaluations; quantitative evaluations of real world SPLs on the one side and user studies on the other side.

The presented hypotheses strongly depend on our multi-level interfaces. Therefore, we plan a stepwise procedure, in which we introduce each of the respective interfaces. For each interface, we have to (1) refine the base concept (cf. [38]), (2) create a prototypical implementation and (3) validate our approach according to the presented hypotheses.

First of all, we have to search for an approach that can be used as basis for the integration and development of our interface concept according to the specific development step. For instance, if we focus on our variability-model interface, we search for an adequate modeling language that makes an integration of our interface concept as easy as possible. Afterwards, we can refine the specific interface concept with respect to the identified approach and the results of the previous development interface.

Second, because of our aim to combine all interfaces to a combined solution, we need an *integrated development environment (IDE)*, in which we can integrate our multi-level interfaces. Thus, we have to search for an environment that is suitable for this integration. Afterwards, we use the refined concept of the previous step, and integrate our interface approach in this IDE.

Third, based on the IDE integration of the specific interface approach, we plan to validate our hypotheses. Whereas Hypothesis H2 can only be validated by a user study, the Hypothesis H1 and H3 can be investigated using quantitative evaluations. The detailed evaluation strongly depends on the specific interface and is partly an open research question.

Threats to Validity

The threats to validity strongly depend on the evaluation procedure and, thus, we divide our threats to validity according to a user study and a quantitative evaluation.

User Study. To investigate Hypothesis H2, we plan an exemplary user study using our syntactical product-line interfaces [39]. We already published our concept for these interfaces and present a quantitative evaluation [39]. We use the results of this evaluation to select an appropriate product line for the planned user study that is able to present meaningful results. Nevertheless, in a user study it is only possible to investigate one or two SPLs and, thus, the generalizability of the results is limited. However, using the results of our previous qualitative evaluation, we expect significant results according to the SPL that we investigate. Furthermore, the results of a user study strongly depend on the subjects of the experiment and the complexity of the presented tasks.

Therefore, we have to ensure that we select the subjects and the tasks carefully.

Quantitative Evaluation. The threats to validity according to the investigation of Hypotheses H1 and H3 strongly depend on the applied subject system. In general, it is hard to find real-world subject system in the domain of MPLs. However, we have access to a lot of open-source SPLs that are available in public repositories. Thus, it is possible to divide these SPLs in smaller dependent SPLs that we can use as base for our evaluation. The open-source SPLs were also used by other researchers and, thus, we hope to increase the acceptance by the research community.

Referring to the internal validity in both evaluation procedures, we have to ensure a correct prototypical implementation. Therefore, we plan to investigate our implementation using some small self-created case studies.

6. PRELIMINARY KEY RESULTS

In this section, we present our preliminary results according to our main concept of multi-level interfaces.

First of all, we proposed the overall idea of multi-level interfaces between SPLs [38]. In this context, we discussed the main reasons for the usage of interfaces in MPLs. For each interface of our multi-level interface approach, we present first ideas according to the concept of the specific interface. The paper is the starting point of the thesis and presents a guideline and initial concepts that we can refine step by step.

Although the interfaces of our multi-level interface approach strongly depend on each other, we start to refine the concept of our syntactical product-line interface. Therefore, we present a technique that can be used in single SPLs without a variability-model interface [39]. However, we are sure that the concept itself is easy adaptable according to MPLs and will profit by the knowledge of the variability-model interface. In detail, we proposed feature-context interfaces as a non-variable interface that presents all code artifacts that a developer can reuse in an SPL based on FOP. We published the concept including a quantitative evaluation that illustrates the benefits of feature-context interfaces and a detailed description of the prototypical implementation in FeatureIDE[1] [42].

In our second step, we focus on the refinement, implementation, and evaluation of the variability-model interface. The development of the variability-model interface itself is still a remaining task, but we already published our ideas related to the analysis of depending feature models [38]. We plan to reuse these concepts for the analysis, in which we evaluate our concept for the variability-model interface. In detail, we plan to compare analyses with and without our variability-model interfaces.

Besides the concepts of multi-level interfaces that describe the core assets of the thesis, we also published a paper according to our evaluation framework QuEval that presents the necessity to reorganize the framework to an MPL [24]. In addition to the description of the running example, we present problems using state-of-the-art techniques for the implementation [24]. Thus, we consider QuEval as motivating example for the thesis.

[1] `http://www.fosd.de/featureide`

7. REMAINING WORK

The first remaining research part of our thesis investigates the variability-model interfaces and their facility to improve automated analysis. We published first ideas in our previous work [38] and plan to improve the main concept in the next month and to include the approach in FeatureIDE. Afterward, we will execute evaluations that compare the results of automated analyses according to the variability model with and without interfaces (see Hypothesis H1). Using these results, we plan a conference submission. The remaining work for this main part of the thesis will require at least six additional months.

Furthermore, we plan to execute a user study to evaluate the benefits of feature-context interfaces as syntactical product-line interfaces. In detail, we plan to develop programming tasks that should be solved by undergraduate students. We plan to divide the students in two groups and compare the time to solve these tasks with and without the help of our feature-context interfaces. For this user study, we want to select subjects of our SPL lecture that takes places every winter term. Therefore, we plan the execution of the user study in the winter term 2014/2015. We plan to submit the results of the study to a journal as extended version of our previous work on feature-context interfaces [39].

Another remaining part investigates the concepts for the behavior product-line interface. In this area, we already published first ideas in our overview paper in which we describe the whole concept for the usage of interfaces between SPLs [38]. The detailed concept is a still open task as well as the detailed evaluation. Similar to the other main parts of our multi-level interfaces, we plan a conference submission with an evaluation part that investigates Hypothesis H3. These concepts strongly depend on the syntactical product-line interface and, thus, it will be the last part of the thesis. For the development of the main concept including the evaluation, we plan to invest half a year.

8. CONCLUSION

Although software product lines improve the development effort of similar products, their implementation and maintenance is still a challenging task. Especially the overwhelming variability in very large-scale systems complicates the development task. Multi software product lines, which are an arbitrary composition of software product lines, help to support developers through the development and maintenance tasks. Although the concepts of multi software product lines are promising, the support for analyses to ensure correctness in each development step is insufficient. Therefore, we propose the concept of multi-level interfaces between software product lines to close the gap. In this proposal, we present our overall idea of multi-level interfaces and present our strategies to investigate their facility to improve analyses on each development step. In detail, we present a set of hypotheses that we plan to investigate throughout the thesis process. Our first evaluations according to the syntactical product-line interface present promising results that we want to validate by a user study. We expect similar improvements in all other steps of the product-line development.

Acknowledgments

We thank Thomas Thüm and Gunter Saake for suggestions and constructive discussions of previous versions of the pa-

per. This proposal was accepted for presentation at SPLC 2014 Doctoral Symposium. The work is partially funded by German Research Foundation (DFG, SA 465/34-2).

9. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. Composing Feature Models. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, pages 62–81. Springer, 2009.

[2] M. Acher, P. Collet, P. Lahire, and R. B. France. A Domain-Specific Language for Managing Feature Models. In *Proc. ACM Symposium Applied Computing (SAC)*, pages 1333–1340. ACM, 2011.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[4] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access Control in Feature-Oriented Programming. *Science of Computer Programming (SCP)*, 77(3):174–187, 2012.

[5] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[7] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[8] M. Bošković, G. Mussbacher, E. Bagheri, D. Amyot, D. Gašević, and M. Hatala. Aspect-Oriented Feature Models. In *Proceedings of the International Conference on Models in Software Engineering (MODELSWARD)*, pages 110–124. Springer, 2011.

[9] A. Classen, Q. Boucher, and P. Heymans. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming (SCP)*, 76(12):1130–1143, 2011.

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

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[12] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *Journal of Systems and Software (JSS)*, 83(7):1108–1122, 2010.

[13] D. Dhungana, D. Seichter, G. Botterweck, R. Rabiser, P. Grünbacher, D. Benavides, and J. A. Galindo. Configuration of Multi Product Lines by Bridging Heterogeneous Variability Modeling Approaches. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 120–129. IEEE Computer Science, 2011.

[14] H. Eichelberger and K. Schmid. A Systematic Analysis of Textual Variability Modeling Languages. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 12–21. ACM, 2013.

[15] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? 18(4):699–745, 2013.

[16] G. Holl, P. Grünbacher, and R. Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *J. Information and Software Technology (IST)*, 54(8):828–852, 2012.

[17] A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards Multi-view Feature-Based Configuration. In *Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 106–112. Springer, 2010.

[18] 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 CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[19] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[20] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 773–792. ACM, 2012.

[21] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proceedings of the International Workshop Visualisation in Software Product Line Engineering (ViSPLE)*, pages 303–313, 2008.

[22] A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

[24] V. Köppen, M. Schäler, and R. Schröter. Toward Variability Management to Tailor High Dimensional Index Implementations. In *Proceedings of the International Conference on Research Challenges in Information Science (RCIS)*. IEEE Computer Science, 2014.

[25] D. Le, E. Walkingshaw, and M. Erwig. #Ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Proc. Int'l Symposium Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150. IEEE Computer Science, 2011.

[26] M. Mannion, J. Savolainen, and T. Asikainen. Viewpoint-Oriented Variability Modeling. In *Proc. Computer Software and Applications Conf. (COMPSAC)*, pages 67–72. IEEE Computer Science, 2009.

[27] J. Meinicke, T. Thüm, R. Schöter, F. Benduhn, and G. Saake. An Overview on Analysis Tools for Software Product Lines. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*. ACM, 2014. To appear.

[28] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.

[29] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

[30] M.-O. Reiser, R. T. Kolagari, and M. Weber. Compositional Variability - Concepts and Patterns. In *Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS)*, pages 1–10. IEEE Computer Science, 2009.

[31] M.-O. Reiser and M. Weber. Managing Highly Complex Product Families with Multi-Level Feature Trees. In *Proceedings of the International Conference on Requirements Engineering (RE)*, pages 149–158. IEEE Computer Science, 2006.

[32] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. In *Proc. Int'l Conf. Object-Oriented Programming Systems Languages and Applications Companion (SPLASH)*, pages 11–18. ACM, 2010.

[33] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. Universität Duisburg-Essen, 2010.

[34] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–22. ACM, 2011.

[35] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 77–91. Springer, 2010.

[36] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. *PVLDB*, 6(14):1654–1665, 2013.

[37] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 252–268. Springer, 2012.

[38] R. Schröter, N. Siegmund, and T. Thüm. Towards Modular Analysis of Multi Product Lines. In *Proceedings of the International Software Product Line Conference co-located Workshops*, pages 96–99. ACM, 2013.

[39] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2014. To appear.

[40] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, 2012.

[41] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 2014. To appear.

[42] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 2014.

[43] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM, 2012.

[44] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 255–269. Springer, 2012.

[45] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.