Towards Modular Analysis of Multi Product Lines

Reimar Schröter
University of Magdeburg
Germany

Norbert Siegmund
University of Magdeburg
Germany

Thomas Thüm
University of Magdeburg
Germany

ABSTRACT

Software product-line engineering enables efficient development of tailor-made software by means of reusable artifacts. As practitioners increasingly develop software systems as product lines, there is a growing potential to reuse product lines in other product lines, which we refer to as multi product line. We identify challenges when developing multi product lines and propose interfaces for different levels of abstraction ranging from variability modeling to functional and non-functional properties. We argue that these interfaces ease the reuse of product lines and identify research questions that need to be solved toward modular analysis of multi product lines.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Reusable Software—*Domain engineering*

1. INTRODUCTION

In the last decades, many complex software systems have been developed as *software product line (SPL)*. SPL engineering is a methodology to develop tailor-made products based on common artifacts [2]. This allows vendors to develop many similar software systems cost-efficiently and with reduced time-to-market. The increasing trend to develop SPLs instead of single programs naturally raises the question: How to reuse product lines?

Let us consider an example of an SPL of *database management systems (DBMS)* in Figure 1, which is implemented in Java and reuses an existing SPL of search indexes. We can generate one product of the Index SPL to reuse its functionality in the DBMS SPL. However, it might be the case that there is not a single product fulfilling all functional and non-functional requirements of the DBMS SPL. Consequently, a suitable product of the Index SPL may depend on the DBMS configuration, and we need to expose the variability of the reused SPL to the reusing SPL. SPLs with such dependencies are also referred to as *multi product line (MPL)* [9].

SPLC 2013 workshops, August 26 - 30 2013, Tokyo, Japan

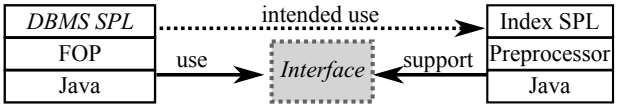

Figure 1: Reusing a preprocessor-based Index SPL in a DBMS SPL implemented with FOP.

Reusing a suitable product of our Index SPL related to the configured DBMS SPL is far from trivial and thus, a challenging task of MPLs. Holl et al. present a literature review about existing approaches to support MPLs [6]. The approaches are categorized related to their capabilities, but none of these address all of the detected capabilities. Our goal is to present a holistic view on the development of an MPL. Similar to Bosch [4], we consider different development levels, but not limited to a closed world scenario and extended by important aspects, such as non-functional properties and specification of MPL behavior. We argue that the correct combination and communication among SPLs of an MPL must be ensured on several levels related to the development process of the MPL. These levels involve variability modeling, implementation, specification, and non-functional properties of MPLs.

At the level of variability modeling, there are approaches such as Velvet [9], TVL [5], and Familiar [1] that support reuse of variability models. However, it is unclear how these models relate to the variable implementation underneath. Furthermore, to the best of our knowledge, there is no approach supporting interfaces in variability models.

At implementation level, compositional approaches, such as *feature-oriented programming (FOP)*, and annotative approaches, such as preprocessors, enable to reuse code of artifacts in an SPL [2], but do not support reusing variability that is provided by other SPLs. Several approaches exist that treat dependencies between components by (variable) interfaces (e.g., [13, 7]). In contrast to these approaches, we focus on interfaces on different levels that depend on each other. Furthermore, our approach can combine SPLs realized with different implementation techniques. Reconsider the DBMS of Figure 1. It is implemented with FOP, but reuses an existing preprocessor-based SPL. Allowing code reuse across different implementation techniques and programming languages is a challenge that is to be resolved.

At specification level, we can define the intended behavior of an SPL by means of formal specifications, such as temporal logic or contracts [11]. Based on these specifications, there are efficient techniques for formal verification [11]. However, it is not clear how to scale these approaches to MPLs. First, specifications need to made avail-

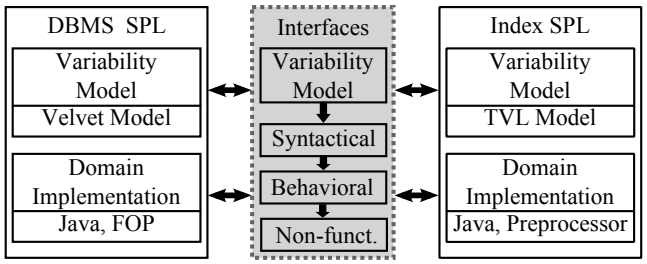

Figure 2: Interfaces for the reuse of SPLs in MPLs.

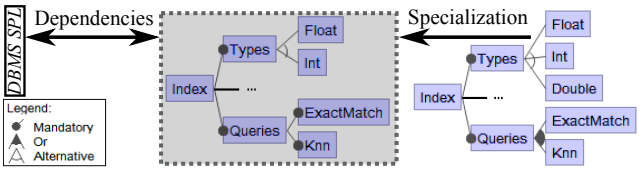

Figure 3: Variability-model interface is a specialization of the Index feature model.

able to reusing SPLs. Second, we need efficient techniques for verification avoiding the redundant verification of reused SPLs.

Finally, at the level of non-functional properties, developers can estimate the influence of an SPL's features on different non-functional properties, such as performance, memory consumption, and footprint [10]. Considering the huge variant space, this quantification is already challenging, but when it comes to MPLs, we face additional problems regarding interactions between features belonging to different SPLs. Clearly, to use a suitable product of the Index SPL in our DBMS SPL, we need to determine which configuration of *both* SPLs results in a product fulfilling all non-functional requirements; again, an open challenge.

We argue that a holistic view on the development of MPLs is required to facilitate modular analysis. We propose interfaces for variability modeling, implementation, specification, and non-functional properties. These interfaces enable a modular view to different levels of SPLs and, thus, improve the reusability of SPLs. Using these interfaces, direct dependencies among SPLs are avoided, unused functionality can be hidden, and evolution is simplified, because it is sufficient to analyze the conformance to these interfaces. In the remaining paper, we investigate all levels in detail.

2. PRODUCT-LINE INTERFACES

When combining multiple SPLs to develop an MPL, we have to guarantee that the collaboration works correctly. To this end, we propose interfaces on multiple stages of the MPL development process. In detail, we distinguish between variability-model interfaces, syntactical product-line interfaces, behavioral product-line interfaces, and non-functional property interfaces (see Figure 2). Syntactical interfaces build on the variability-model interface. That is, a successful check at the modeling level is the precondition to check the correctness of the SPLs against the syntactical interfaces. Similarly, both remaining interfaces build on the syntactical interface and are exchangeable related to given requirements. This way, we handle errors due to reusing SPLs as early as possible and fix them at the stage at which the error originates. Furthermore, the proposed interfaces depend each other, such that, for example, the syntactical interface provides only features that exist in the variability-model interface. In the following, we describe each interface in detail using our running example.

2.1 Variability-Model Interface

The variability-model interface addresses the allocation of functionality provided by one variability model that is needed by another variability model. The interface is a variability model itself that can be seen as an agreement between the using and the reused SPL. Thus, the feature model of the DBMS can rely on the variability-model interface, such that all configurations that are represented by this interface can be reused. To this end, the interface defines a subset of the configurations available in the variability model of the reused SPL (a.k.a. specialization [12]).

In Figure 3, we illustrate a variability-model interface for the Index feature model. In this example, the interface is a feature model in which the alternative feature `Double` is removed, and both features of the or group are specialized to the mandatory features `ExactMatch` and `Knn`. We use this feature model as variability-model interface, and refer to it when modeling the DBMS SPL.

Several variability modeling languages, such as Velvet [9], TVL [5], and Familiar [1] allow us to combine feature models to model MPLs. However, the resulting composed feature models can become complex and hard to understand. Instead, by using our variability-model interface, we can significantly reduce the complexity of these models and, enabling domain engineers to focus on the features and dependencies that are relevant to their MPL (i.e., we leave out the irrelevant features in the interface).

Another benefit of variability-model interfaces is a reduced complexity of automated analysis. Automated analyses are used to detect inconsistencies (e.g., dead features) and calculate statistics [3]. They become especially important in the context of MPLs, because variability models are inherently larger than for single SPLs and dependencies among several SPLs introduce further complexity. In previous work, we analyzed MPLs based on a propositional formula of the complete MPL [9]. This required to create the combined propositional formula of the DBMS and Index SPL. Using our interface, we analyze the DBMS model in a modular way, because, due to the specialization of the Index SPL, the complexity of the propositional formula is reduced to the visible features of the interface.

Variability-model interfaces simplify evolution of MPLs. For instance, when changing the feature model of the Index SPL, we need to check whether the variability-model interface is still a specialization of the evolved feature model. If so, there is no need to redo any automated analysis in the MPL. If it is not a specialization, we have to adapt either the interface or the evolved variability model. Only if the interface itself needs to be changed, we have to redo the analyses in the MPL.

However, there are open research challenges that need to be addressed by future work. It is not clear to what extent we can and should automate the process of creating a variability-model interface. In case a manual process is preferred, it is questionable whether these interfaces shall be created by the domain engineer of the reusing SPL, the reused SPL, or as an agreement of both. Finally, the benefit of variability-model interfaces for the automated analysis of evolving MPLs should be evaluated quantitatively.

```
1 interface IIndex{
2   //constraint: Tuplewise && Int
3   boolean insert(int[] point);
4   //constraint: Bulkload && Int
5   boolean insert(int[][] points);
6   //constraint: KNN && Int
7   int[][] searchKNN(int[] point);
8 }
```

Figure 4: Syntactical product-line interface of the Index SPL.

```
1 interface IIndex{
2   ArrayList<int[]> cont; //Point container
3   //@ requires point != null
4   //@ if(UniqueKeys)
5   //@   ensures (exist(point) => \result = false)
6   //@   && cont.size() == \old(cont.size())
7   //@ else
8   //@   ensures cont.size() == \old(cont.size())+1
9   boolean insert(int[] point);
10 }
```

Figure 5: Behavioral interface of the Index SPL.

2.2 Syntactical Product-Line Interface

A syntactical product-line interface is an *application programming interface (API)* that contains variability. The interface represents a view to the reusable code artifacts of an SPL without containing implementation details, such as method bodies. In detail, the interface contains those classes and their members of one SPL, which are intended to be accessible by another SPL. Since classes and class members may not be available for each configuration, we annotated a propositional formula indicating the availability. The propositional formula may contain only features that are defined in the variability-model interface to enable modular reasoning.

In Figure 4, we illustrate a syntactical product-line interface of the Index SPL. It contains classes and methods that can be used in the DBMS SPL. Each annotation is indicated by the keyword `constraint`, which is followed by a propositional formula, describing the availability related to features defined in the variability-model interface (because of limited space, features `Bulkload` and `Tuplewise` are not illustrated in Figure 1). We can use these annotations to check the compatibility of the interface with the DBMS implementation and the Index implementation. For the DBMS, we need to check whether methods are only accessed from DBMS products that exist in the interface. Similarly, we need to check whether all methods that are defined in the interface, exist at least in those Index products. For instance, method `insert(int[] point)` is available if features `Tuplewise` and `Int` are selected in combination. If a feature of the DBMS SPL requires both features, this method is available and can be reused.

A benefit of syntactical product-line interfaces is information hiding. The developer of the DBMS SPL does not need to know the complete implementation of the Index SPL. Instead, the developers can look-up methods and their annotations directly in the interface. Furthermore, the interface refers only to features defined in the variability model interface, which further reduces the complexity of SPL reuse.

Furthermore, the syntactical product-line interfaces enable modular type checking and compilation. In our example, we can compile the DBMS SPL without having the source code of the Index SPL, and vice versa. However, the price for this modularity is that the interface must be checked for compatibility with both SPLs. Separate compilation is an important property for MPLs, because each SPL may be developed by a different vendor, and may rely on a different variability implementation mechanism. Separate compilation is also beneficial for evolving MPLs. When changes the Index implementation do not change the interface, there is no need to recompile the DBMS.

However, there are still open research problems for syntactical product-line interfaces. A challenge is to define a suited interface, which contains all and only necessary members. Missing members are problematic as the developer needs to adapt the interface to add them. Unused members are problematic, since the interface might become frequently adapted caused by changes in the reused SPL. Hence, changes to the interface should be avoided to enable modular analysis. Further research is needed to evaluate to what extent the creation of stable interfaces can be automated especially in the presence of heterogeneous implementation techniques.

2.3 Behavioral Product-Line Interface

A behavioral product-line interface is an agreement on the behavior of different methods to guarantee a correct communication between multiple SPLs. This is especially important in MPLs, because multiple features coming from different SPLs can influence the behavior of one method, which is difficult to foresee. We propose to use the methodology of design by contract to specify the behavior of methods. A method is annotated with a method contract consisting of a precondition defining under which condition the method may be called and a postcondition defining what the caller can rely on [8]. We extend method contracts by statements indicating for which configurations a precondition or postcondition must be established.

In Figure 5, we illustrate a behavioral product-line interface for the Index SPL in an extension of the Java Modeling Language. Method `insert` has the parameter `int[] point`. The boolean return value indicates whether the point is correctly inserted. The behavior of this method differs depending on the configuration. If feature `UniqueKeys` is selected, the method does not allow us to insert identical points and, thus, the method returns `false`. By contrast, if feature `UniqueKey` is not selected, we are able to insert the key even if the key already exists.

The behavioral product-line interface documents the intended behavior of the reused SPL. Hence, we can understand methods of the reused SPL to a certain degree by looking into the interface rather than the actual implementation. Consequently, the behavioral product-line interface enables modular understanding of the MPL. When reusing closed-source SPLs, we cannot examine the method behavior by looking into the source code, so that such an interface is even more valuable.

Furthermore, the behavioral product-line interface is an agreement between SPLs that enables blame assignment. Given that the vendors of two SPLs have agreed on a behavioral product-line interface, it can be used to assign blame on one of the vendors if a contract is not fulfilled for a given configuration. The agreement may even be useful to develop SPLs of an MPL in parallel.

Similar to the other interfaces, we see some research challenges regarding behavioral product-line interfaces. Again, it is the question whether this interface shall be written manually or could also be generated automatically to a certain extent. Having these formal specifications, it is also an open research challenge how to detect violations in the MPL in a

```
1 interface IIndex{
2   ArrayList<int[]> cont; //Point container
3   // Performance:
4   // Time = point.length * 10 ns
5   // if(UniqueKeys)
6   // + ln(cont.size()) * point.length * 10 ns
7   // Footprint:
8   // 350 KB
9   // if(UniqueKeys)
10  // + 25 KB
11  boolean insert(int[] point);
12}
```

Figure 6: Non-functional interface of the Index SPL.

modular way; this may involve theorem proving, static analysis, model checking, or even runtime assertion checking. It is not clear to what extent existing techniques [11] can be extended for modular analysis of MPLs.

2.4 Non-Functional Product-Line Interface

A non-functional product-line interface describes the non-functional properties of an SPL that other SPLs can rely on. The non-functional properties specified in such an interface depend on the modeled product line and the requirements of stakeholders (e.g., performance, memory consumption, or footprint). The non-functional product-line interface is based on measurements for the reused SPL or on predictions based on domain knowledge. As the configuration of the reused SPL highly influences the non-functional properties of the final product, the interface specifies the properties with respect to features of the reused SPL. Similar to the other interfaces, only features of the variability-modeling interface may be used.

In Figure 6, we illustrate a non-functional product-line interface for the Index SPL, describing the performance and the binary footprint. We use annotations similarly to the previous interfaces to ease tool support by providing a consistent description. For example, the response time depends on the number of data points to be inserted. However, if we select feature `UniqueKeys`, we have to add the time required to search for duplicates within the index container (`cont`). Having this specification, we can compute the expected performance depending on the feature selection of the MPL. Similarly, we can predict the binary size of the resulting MPL program, which is, for instance, important for embedded systems.

A non-functional product-line interface is especially useful for MPLs that have to satisfy non-functional constraints such as performance policies. In an MPL, non-functional properties emerge in a complex interaction of the SPLs involved. The proposed interface enables modular prediction of non-functional properties, when we rely on the properties in the interface rather than doing a prediction for reused SPLs. This way, we can save effort for redundantly analyzing non-functional properties of the reused SPL. When reusing closed-source SPLs, we can even use white-box techniques requiring the source code for measurement; each vendor can measure its own SPLs in isolation.

The main research challenge is how to predict the non-functional properties of the MPL based on these interfaces rather than measuring the reused SPLs again. Specifically, how can we determine interactions among SPLs, affecting the outcome of non-functional properties? Furthermore, when an MPL does not meet the overall non-functional requirements, we need approaches to identify the SPL that is responsible for the violation.

3. CONCLUSION AND FUTURE WORK

Multi product lines (MPLs) facilitate the reuse of software product lines (SPLs) in other SPLs. We argue that MPLs require sophisticated techniques to enable high-level reuse and modular analysis. As a solution, we propose interfaces for different levels of the development process: variability modeling, syntactical, behavioral, and non-functional product-line interface. We exemplified how these interfaces enable modular development and analysis of MPLs.

We identified open research challenges regarding interfaces in MPLs. For instance, it is unclear how to design or generate interfaces that change less frequently than the involved SPLs, but are still sufficient for the communication of SPLs. We are currently working on tool support for all interfaces in the development environment FeatureIDE.[1] Given the tool support, we plan an evaluation based on publicly available SPLs.

ACKNOWLEDGMENTS This work is partially funded by DFG grant SA 465/34-2, and BMBF grant 01IM10002B.

4. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. A Domain-Specific Language for Managing Feature Models. In *SAC*, page 1333–1340. ACM, 2011.

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

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[4] J. Bosch. Software Product Lines: Organizational Alternatives. In *ICSE*, page 91–100, 2001.

[5] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-Based Feature Modelling Language. In *VaMoS*, page 159–162, 2010.

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

[7] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *OOPSLA*, page 773–792, 10 2012.

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

[9] R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated Analysis of Dependent Feature Models. In *VaMoS*, page 9:1–9:5. ACM, 2013.

[10] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In *SPLC*, page 160–169. IEEE, 2011.

[11] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, 2012.

[12] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *ICSE*, page 254–264. IEEE, 2009.

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

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