Supporting Heterogeneous Compositional Multi Software Product Lines

Sascha El-Sharkawy, Christian Kröher, Klaus Schmid
Software Systems Engineering, Institute of Computer Science, University of Hildesheim, Germany
{elscha, dederichs, schmid}@sse.uni-hildesheim.de

Please cite this publication as follows:

The corresponding BibTeX-entry is:

```bibtex
@INPROCEEDINGS{El-SharkawyKroeherSchmid11a,
  author = {Sascha El-Sharkawy and Christian Kr{"o}her and Klaus Schmid},
  title = {Supporting Heterogeneous Compositional Multi Software Product Lines},
  year = {2011},
  editor = {Ina Schaefer and Isabel John and Klaus Schmid},
  volume = {2},
  publisher = {ACM},
  doi = {10.1145/2019136.2019164}
}
```

ACM, 2011. This is the authors version of the work. It is posted here by permission of the ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the Joint Workshop of the 3rd International Workshop on Model-driven Approaches in Software Product Line Engineering and the 3rd Workshop on Scalable Modeling Techniques for Software Product Lines (MAPLE/SCALE 2011) at the 15th International Software Product Line Conference (SPLC ’11), DOI: 10.1145/2019136.2019164.
Supporting Heterogeneous Compositional Multi Software Product Lines

Sascha El-Sharkawy, Christian Kröher, Klaus Schmid
University of Hildesheim, Institute of Computer Science
Marienburger Platz 22, D-31141 Hildesheim, Germany
{elscha,kroeher,schmid}@sse.uni-hildesheim.de

ABSTRACT
Software Product Line Engineering is inherently complex. This complexity increases further if multiple product line infrastructures are composed to yield the final products, an approach sometimes referred to as Multi Software Product Lines (MSPL). In this paper, we present an approach that targets this development scenario. The approach we present here aims at a lightweight, scalable, and practical approach to variability management for multi software product lines.

Our approach explicitly supports heterogeneous product lines, i.e. situations where the various product lines use different generation approaches. The approach has been implemented in the EASy-Producer tool set and applied on some case studies.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]; D.2.13 [Reusable Software].

General Terms
Design, Management

Keywords
Multi Software Product Lines, Variability Modeling, Decision Modeling, Heterogeneous Composition, Distributed Work

1. INTRODUCTION
Software Product Line Engineering (SPLE) typically relies on composing variable artifacts that are part of a reuse infrastructure [2, 8]. Often a variability model is used to support this composition process [7]. It enables to ignore the complex dependencies that may exist on the artifact level, as the variability model makes them explicit (in the form of constraints).

In software development in large organizations, respectively in multi-organizational software development we need to face an additional problem: products are often composed of components from various development organizations, each of them developing their parts as an independent product line. Such a situation, where products are derived as the result of multiple product lines is also called multi software product lines [10]. In such a situation not only variability within the various product lines need to be addressed, but also interactions among the different product lines must be taken into account. These interactions must be addressed both on the level of the variability model as well as on the level of the artifact model. At this point already several approaches exist which deal with multi product lines [4, 5, 10]. We stipulate, however, that it is possible to provide more lightweight approaches to composing products from multiple product lines. In this paper, we present a scalable approach we developed, which we regard as better suited for minimizing the complexity of multi software product line development. This approach has been implemented in the EASy-Producer tool set and was tested on some case studies.

2. REQUIREMENTS OF MSPL
In this section, we will detail what we regard as key requirements for any approach to support multi software product lines. Thus, we assume these requirements are more generic than our specific approach and tool, while they are certainly build on this. We will then discuss in Section 4 how our approach addresses these requirements.

A core goal of any product line engineering approach is to support efficient development. Thus the approach and tools must minimize the effort and complexity of the development task. An important motivation for multi software product lines is that the individual product lines are developed in a distributed manner by multiple, more or less independent organizations. In such a case the contributing organizations may hardly exchange any information, except for the information required to instantiate the individual parts.

Related to the requirement of distribution is the notion of heterogeneity. Mostly independent organizations will use different tool infrastructures; they will make different assumptions about the development, etc. Even within a product population, as discussed by van Ommering [14], different generation processes and tools will be used. In such a situation, it may be possible to integrate the resulting instances, but it might be quite difficult to reconcile the individual techniques for generating them. The resulting product instances must then be composed to form the final product. Please note that we do not need to compose the sub-product lines per se, but it is sufficient to compose the derived products. Actually, different products may compose instances in completely different ways. For example, in Figure 1 the hospital system and the research system may compose the instances rather
differently. Thus, a single composition model would not be appropriate to govern them both. It is important to integrate the instantiated product line infrastructures into a single product. **Composability** implies that the product engineer can easily handle the resulting, instantiated infrastructures.

From a perspective of minimizing complexity it is well important to handle product-specific functionality. Modeling this as a variability leads to significant overhead. In the context of multi software product lines this is a particular problem as it is unclear to which product line to assign this functionality.

The high-level requirements can be refined as follows:

- **Heterogeneity and distribution** require that the product lines can be treated as independent projects. They are only integrated as needed in the context of the derived product. This requires that each product line has its own variability description, its own artifacts and its own approach to instantiating these.

- **Handling composability with minimal complexity** requires that the instantiated product line artifacts can be easily handled in combination by the product developer. Also the composition on the artifact side and on the variability model side should happen in a synchronized way and be easily accessible to the developer who works on the product level.

- **Supporting product-specific functionality** with minimal complexity requires that the product developer can handle product-specifics independently of concerns about the reused product lines.

In order to show in more detail how we address these challenges in our approach, we will introduce in the next section some basic concepts and terminology.

### 3. Basic Concepts

This section defines some basic concepts relevant to our approach. This provides the terminology for further discussions below.

The **Variability Space** denotes all issues concerning variability modeling in software product line development. This includes, defining and maintaining the variability model, selecting variants for the variation points and defining the resulting model configurations. We differentiate:

- **Variability Model (VM)** defines the variability of the software product line. This covers all variation points including their variants as well as the associated constraints.

- **Configuration** denotes the process of making decisions or selecting variants to achieve one model representing the final product.

- The resulting model is called a **Product Configuration**. This conforms to the constraints on the variability model.

The **Artifact Space** includes all relevant to software artifacts in software product line development, like developing and managing product line artifacts, instantiating artifacts for use in a specific product and the resulting artifact instances.

- **Product Line Artifacts** are any artifacts that are needed to create the products of the product line. They are generic in the sense that they contain variation points that allow the selection of variants. Different product lines will typically use different formats, e.g., different programming languages.

- **Product Artifacts** are specific parts of concrete products. Variability has been as far as possible removed.

- **Instantiation** refers to the process of creating these product artifacts as defined by the product configuration.

![Figure 2: Elements of the derivation process](image)

Variability Space and Artifact Space are conceptually different, but strongly related to each other. Configuration defines semantically what should happen to the artifacts while instantiation then executes the corresponding transformations. In this paper, we call such a semantically consistent combination of configuration and instantiation a derivation. The relationship of these three processes is depicted in Figure 2. As we will discuss in the following section, our approach will rely on a homogenous variability model representation while supporting heterogeneous derivation approaches.

## 4. OUR APPROACH

In this section, we will discuss our approach and how it relates to the core requirements we discussed in Section 2. We will discuss the specific design decisions we took in the next section, while Section 4.2, will discuss the specifics of the current implementation.

### 4.1 Key Design Decisions

There are many ways to address the requirements outlined in Section 2. Thus, we will now discuss the main decisions that influenced our approach:

- We decided to use a single variability language across all product lines and products. This simplifies the configuration process both on a technical level as well as from the developer perspective. Thus, we support the efficient composition of distributed MSPLs. We decided to use decision modeling as described in [12] as a foundation of our approach. With respect to the example given in Figure 1, this means that both **PL_Cabin** and **PL_Control** will be configured in terms of decision models and that the product Hospital System will be defined as product configuration in terms of decision values.

- Variability exposed by the constituting product lines is exposed in terms of decision variables for the products. The composition on the level of the decision variables is rather straightforward as it is fully homogeneous (all product lines use the same language). All variables are currently combined. The chosen approach significantly simplifies the handling of the variability models. It thus supports efficient composition.

- Constraints can semantically support composition of the different variability model parts that are inherited from the various product lines. This supports the efficient and correct composition of distributed product lines, as it removes any dependencies among the individual product lines and puts it in the realm of the products where the dependencies matter. In our example, both **PL_Cabin** and **PL_Control**, may relate to a decision on number of floors, named displayed floors and floors, respectively. In the context of a specific product a constraint can be added that ascertains that both are identical.
Within the development environment products and product line infrastructure are mapped to projects. This simplifies the development and makes them individual configuration management entities. This greatly improves the handling for distributed development, as we can also update the various parts individually. In the current implementation the Eclipse environment is used as a basis. This means that in our example, PL_Cabin, PL_Control, and Cabin would be individual projects.

• Derivation of a product line to yield a part of a product is mapped to a part of a project (i.e., a sub-project). This is a sub-tree (e.g., a package in Java). This supports distributed development and simplifies the composition. It also helps to insulate in some degree the heterogeneous parts. In our example, this would lead to a substructure for the instantiated PL_Cabin and PL_Control, respectively. These would be part of the Cabin project.

• Instantiators are first-class entities. They are self-contained and work through a well-defined interface. They may also be external tools that are included via a wrapper. As they are individual entities they can be explicitly managed. This helps to scale the approach and make it better manageable in a heterogeneous situation. In our example, PL_Cabin and PL_Control would define their own instantiators and different instantiators might even handle the same types of artifacts.

• The instantiated product parts keep their relation to the parent product line. They also keep the relation to the responsible instantiators. Thus, the corresponding instantiator is used to (re-)instantiate a parent product line. This particularly helps to support heterogeneity. In our example, this means that if the part resulting from PL_Cabin needs to instantiated or updated, the instantiators defined in PL_Cabin are used.

• Product-specific functionalities are part of the product-project. This reduces complexity of the product lines as these need not be concerned with this on the artifact side. As a consequence, domain engineers do not need to handle such product-specific parts. This supports the efficient development. In our example, this means that if a product specific functionality that does not result from PL_Cabin or PL_Control is simply developed within the project.

Figure 3 gives an overview of our approach. The product lines PL_Cabin, PL_Control, and PL_Environment are used as a basis for the derivation of the product Research System. This is used to handle the variabilities of these three product lines. Upon selecting the corresponding decision values, the different product lines are instantiated. They all have different artifacts, but also their mechanisms for instantiating the artifacts may differ. Correspondingly, each of these product lines has its own instantiators as illustrated in the figure. Each of them is activated separately to instantiate the artifacts according to the composed product configuration. The product is also extended by product-specific functionalities that are represented by a cross, e.g., code that integrates the heterogeneous parts.

4.2 Instantiation Approach
The instantiators are a key part of the approach. We will thus discuss some of their key properties in this section:

P_1 All instantiators are controlled by the decision values.
P_2 Different artifacts could use different markup languages and thus different instantiators can be used. For instance, the same decision could be implemented in various ways depending on whether the source file is Java, XVCL or a make file.
P_3 A configuration interface is provided that supports the product line engineer in relating instantiators to the product line, respectively specific parts of it. This is needed as already a single product line can consist of many artifact types.

The support for distributed and collaborative work is also a major part of our work, as this is a major driver of project complexity:
P_c1 It must be possible to prevent interactions among different instantiation processes. This is currently the responsibility of the respective domain engineers, which must organize their product lines in appropriate packages to avoid undesidered interactions.
P_c2 It must be possible to update the product infrastructure to a certain time. Thus, it is possible to perform the instantiation process at any time, if the product line can be accessed. It also causes to update the product configuration, while keeping the decision values.
P_c3 It must be possible to work on the products without having direct access of the full product line infrastructure. After initial product derivation, the product contains the product configuration and instantiated artifacts, together with the capability of updating artifacts. It is thus possible to work distributed and independent on the product line as well as on the product.
P_c4 The sole composition of variability spaces must not causes an inconsistent product configuration. Thus, during the combination variability models name spaces are introduced to handle name clashes. As a consequence, the variability models of the parent product line are disjoint.

5. RELATED WORK
The challenge of supporting and simplifying MSPL development has been previously by van Ommering [14]. He introduced the term product population to denote the situation that products are built from assets derived from multiple product lines. This early approach did not include an explicit variability management approach; rather variability was mostly represented on the architectural level using the ADL Koala, which supports the description of variation.

Most approaches to multi software product lines rely on feature modeling. Notable exceptions, besides the work discussed in this paper, is the work by the Dopler group, which is presented in [6] and the work by Elsner et al. [4]. While the former relies on decision modeling, the second aims at an integration approach that supports the combination of multiple variability models that can be represented in different ways. In [11] also several approaches from industrial and open source practice are discussed that support variability management in a distributed situation.
So far several ways have been defined to deal with the integration of multiple variability models. Discussion relevant to the problem can be found in [9]. There different concepts and patterns for composing variability models are discussed. Early approaches, like FeaturePlugin [1] or Feature-Oriented Domain Analysis Engineering development method (FORE) [13] supported the notion of variability links. In this case one feature model could link another one. As the relation to artifacts remains unclear, it is not clear whether we can regard this as an approach to support multi product lines or whether it is only an approach to partition the variability model of a single product line. The later would be comparable to decision model fragments as introduced in [3]. However, such an approach where integration of some fragments is given by references is a rather lightweight approach and is thus desirable, given the criteria we defined in Section 2.

More recently some approaches introduced the notion of composition models as a basis for defining the combination of multiple software product lines [10, 5]. While these approaches have their advantages, e.g., in [10] the automatic derivation of a combined configurator is described, the construction and management of the additional model also incurs some overhead.

While the previous approaches rely on a homogenous representation of variability, Elsner et al. introduce in [4] an approach that aims at integrating variability in case the different sub-product lines use different variability representations. Their approach relies on the transformation of an external, heterogenous variability description into an internal model.

6. CONCLUSION AND FUTURE WORK
Multi software product lines result from the composition of various (independent) product lines. In this paper, we presented a lightweight and practical approach to handle variability in multi software product lines. This approach is tool supported and has been applied in some case studies. We derived guiding principles in a systematic way from the general demands of MSPL. Based on the requirements, we derived, we show how our approach supports distributed product line development and composition while minimizing development effort and complexity.

The presented approach is characterized by the use of a common variability modeling language while supporting heterogenous product line implementation techniques. A common variability language simplifies product configuration as each product line is expected to bring its own instantiators that are able to interpret the product configuration and correspondingly instantiate the specific product line artifacts. Simplicity in integration of product-specific functionalities is achieved by shifting the management of these functionalities from the product line level to the product level. Thus, a product can be extended by specific functionalities without influencing the variability management.

In the future, we will extend our approach to support a more complex variability modeling language. While the current version of the tool supports decision modeling, some advanced constructs are not yet well represented. A further extension will be to extend the support for more complex instantiators.

7. ACKNOWLEDGMENTS
This work is partially supported by the INDENICA project, funded by the European Commission grant 257483, area Internet of Services, Software & Virtualisation (ICT-2009.1.2) in the 7th framework programme.