Challenges in Software Product Line Composition*

Horst Schirmeier and Olaf Spinczyk
Technische Universität Dortmund, Informatik 12
Otto-Hahn-Str. 16, D-44221 Dortmund, Germany
{Horst.Schirmeier,Olaf.Spinczyk}@tu-dortmund.de

Abstract

The idea to develop applications and infrastructure software as software product lines (SPLs) is continuously growing in acceptance throughout the software industry. The ability to customize software to customer needs or a specific application scenario, and the advantages arising from code reuse throughout the product line will further enhance this trend in the future. Although the construction and the automated configuration of single SPLs is already well understood, new challenges come into being from the composition of multiple product lines from potentially different software producers. This article discusses several problems originating in multi-layer and multi-instance composition of SPLs, which we expect to be a common situation in the future. Focusing on infrastructure SPLs and the resource-constrained embedded systems domain, we describe possible approaches and starting points for prospective research.

1. Introduction

Specific application scenarios or customer groups call for tailored software systems that match their exact needs. Especially in the resource-constrained embedded systems domain every unneeded piece of code and every redundant CPU cycle directly maps to additional hardware costs, which is among the highest priorities of factors to be minimized. Statically configurable systems, in particular tailor-able infrastructure software such as embedded operating systems [9, 8] and embedded data management [11, 10], have been in the focus of our research group for several years.

Software product lines (SPLs) and feature-driven product derivation are widely adopted approaches to meet the demands for highly configurable software systems [7, 4]. Providing guidelines for the software engineering process, implementation techniques to cope with variability [1, 13], and a variety of tools supporting variability management and product configuration [2], these approaches have already proven to be viable for developing and instantiating single SPLs. The trend goes towards more and more software being developed as SPLs nowadays, particularly in embedded systems areas such as the automotive domain.

Also a very common approach in the embedded systems domain is the composition of complex software systems from several modules, typically developed by different software producers. Modular software composition is a classical research field with years of experience, but little research has gone into this emerging necessity to be able to integrate multiple SPLs in a single system. Therefore, we want to concentrate on aspects that are unique for the composition of SPLs, which include

- the consequences of composition on variability modeling,
- functional and non-functional constraints crossing the boundaries between the SPL instances that compose the overall system,
- means to express these relationships,
- and advantages resulting from composition, including automatic configuration.

The remainder of this paper is organized as follows: Section 2 gives an overview on SPLs and their configuration and instantiation process. Section 3 discusses different composition architectures and the fundamental difference between composing SPL layers and ordinary software layers. Sections 4 and 5 elaborate on aspects specific for vertical respectively horizontal composition of product lines. The article concludes with a discussion of related work and a summary in sections 6 and 7.

2. SPLs and Product Instantiation

Feature-driven product derivation [14] begins with a feature model, which consists of features and sub-features in
In the configuration and instantiation process, a product variant is defined by selecting the needed features. This feature selection is usually carried out by a human user, interacting with a configuration tool and finally triggering the product generation, resulting in an actual SPL instance. In the following, the term “instance” is used for a single product of a SPL.

3. SPL Composition

At first glance, composing products of SPLs is not much of a difference to classical software module composition. After all, the result of feature-driven product derivation is a piece of software like any other, communicating with neighbouring modules through some kind of interface.

So the interesting part lies within the variability that pervades the SPLs, and the configuration process that selects concrete product line instances that make up the composed software system.

SPL instances can mainly be composed in three different ways:

**Vertical Composition** An application SPL is built upon several layers of different infrastructure SPLs (such as, for example, an embedded database SPL, a libc product line, and an embedded operating system SPL) that incorporate a “uses” relationship from top to bottom (Figure 3). Usage in this context means API calls from one layer to a layer below, as embedded software is usually compiled and linked to a single binary, ignoring large-scale operating system concepts like shared libraries or dynamic loading.

**Horizontal Composition** Several instances of the same SPL are organised in a horizontal manner, sharing the same environment (e.g. a network or an embedded device). The binding is much looser than in vertical composition, usually limited to higher-level communication primitives. Sharing the same environment also can mean having to share resources, such as limited RAM on an embedded device (Figure 2).

**Hybrid forms** In practice, the aforementioned extremes rarely occur in the described purity, but in hybrid forms that have properties of both Vertical and Horizontal Composition. This includes the common case of two or more (similar) applications running on top of a collectively used stack of infrastructure software layers, or neighbouring sensor nodes with composed software built from the very same SPLs.

Nevertheless, the next two chapters will elaborate on the first two (“pure”) forms of SPL composition, as we believe that the concepts learned from these extremes can be combined when dealing with real-life, hybrid forms of composed SPLs.

4. Vertical Composition

A stack of vertically composed SPLs typically has a single application SPL on the top layer, with one or more infrastructure SPLs in the layers below (cf. Figure 3). In this context it is a crucial fact that every contributing SPL may come from a different software producer, and the layers are supposed to be interchangeable with product lines from other producers.

The logical relationship from top to bottom can roughly be described as “uses and needs functionality of”; technically this simply means API usage in terms of function calls and data interchange. This relationship only points downwards, implying that an SPL instance on layer \( n \) may depend on features (or: functionality) provided in the layers
$n-1$ to $1$ below, and not the other way around. This might seem obvious, but has some interesting consequences described in the following subsections.

Beyond functional constraints (“Layer $n$ needs feature X of layer $n-1$”), possibly additional non-functional constraints (such as: “The application layer needs the database layer to guarantee reaction times below 50ms”) may exist between the SPL layers, but this is already the subject of current research [12].

4.1. Modeling Feature Constraints

A feature model describing the variability is key to the configuration process of a single SPL. Usually organised in a feature tree with features and sub-features, constraints like optional/mandatory features or alternative/cumulative feature groups (see also Kang et al. [7]) are accompanied by additional feature dependencies like requires or excludes, which connect features orthogonally to the feature hierarchy.

In a vertically composed stack of SPLs, features on one layer may depend on features of a lower layer, e.g. demanding a specific functionality or behaviour: Constraints of all types may cross the layer boundaries, connecting the feature models from top to bottom. These constraints can be modeled in different ways:

Explicit modeling The designers of an SPL can decide to explicitly model the dependencies on product lines on lower layers by referring to particular features of an infrastructure layer below (cf. Figure 4). The obvious implication is, that the feature model designers needs to know the feature model of the referred layer. These constraints need to be manually modeled, which can be a very error-prone process—but another, much more problematic implication emerges from explicit modeling: As the layers are supposed to be interchangeable, now not only a layer’s API needs to be standardised, but also its feature model. Stepping on virgin soil, we propose some approaches to tackle this problem.

- The top levels of an infrastructure SPL’s feature tree could be standardised, allowing for proprietary subtrees at the leaf nodes.
- Alternatively, a flat set of features or feature groups (e.g. alternative/cumulative groups) could be used for this purpose, which may be freely positioned within the proprietary feature tree.
- Standardising feature semantics is another open problem, which could probably be approached by incorporating the already standardised infrastructure APIs (e.g. a POSIX feature model).

Implicit modeling As we described in [11], features of a lower-layer SPL could be enhanced with annotations that describe what properties in higher-layer SPL instance code imply the need for specific features in the lower layer. The clear advantage of this approach over explicit modeling is, that higher-layer feature model designers do not need to know the particular features of product lines below—no feature model standardisation is necessary. Unfortunately, this is only true for the ideal case; in reality only a subset of features can be automatically derived from higher-layer instance code.

Although explicit modeling seems quite disadvantageous—the need for infrastructure manufacturers on each layer to agree on a standard feature set could pose a severe obstacle, especially in highly
specialised niches of the embedded systems market—the idea of **implicit modeling** has not been fully explored yet. Probably future research will reveal that both approaches need to be used in combination, in order to automate the configuration process to an extent that makes large-scale SPL composition feasible and reasonable.

As outlined, the consequence of vertical product line composition on variability modeling is highly increased complexity, in terms of inter-layer dependencies and feature model standardisation needs. Nevertheless, transferring the advantages from automatic configuration of single SPLs to the multi-layer case may reward for taking this hurdle.

### 4.2. Automatic Configuration

The configuration process preceding the product generation in a single SPL is nowadays supported by configuration tools, such as for example pure::variants [2]. Using the constraints (e.g., feature X requires feature Y) in the SPL’s feature model, the configuring user is supported by automatic dependency resolution which maintains a valid product configuration.

Regardless of how the inter-layer constraints in the multi-layer SPL situation (see Subsection 4.1) are modeled, in principle these dependency resolution mechanisms could as well be applied to the case with multiple feature models, connected by inter-layer constraints. But in our opinion, the resolution process should be adapted to the new situation, choosing among several different strategies.

**Top-down Configuration** One straightforward approach would be to first allow the user to manually configure the top layer \( n \) and to resolve the dependencies within that layer until the user is satisfied (cf. Figure 5). Afterwards, effects on the directly connected layers should be calculated, resolving the dependencies within layer \( n-1 \) while leaving the layer \( n \) configuration untouched. The layers should be iteratively configured downwards, until the SPL stack is completely configured, or an unresolvable conflict occurs. The user may be allowed to express additional wishes for lower layers’ configuration, e.g., enforcing the selection of a certain feature although not explicitly needed by layers above, or manually specifying one of multiple or features.

This configuration direction would work for both **explicit** and **implicit** cross-layer constraints.

**Bottom-up Configuration** Working in the opposite direction, the **lowest** layer (layer 1) would initially be configured manually (or has a fixed configuration because it models given hardware). Then iteratively the layers upwards “adjust to provided features”: Starting from a default configuration (possibly empty, possibly with some default features selected), constraints are resolved leaving lower layer configurations untouched. An interesting observation regarding **implicit dependency modeling** is that this technique has been practiced in Open Source Software for a long time: So-called **configure** scripts determining capabilities of libraries and other infrastructure create a configuration file (in the majority of cases called “config.h”), which in turn determines the features enabled in the program about to be compiled. Another interesting aspect in **Bottom-up Configuration** is that it models current practices in embedded systems development: Modifying/switching the hardware platform (e.g., to a smaller RAM/Flash memory size due to pricing pressure) after the product is already in the market is common and would directly lead to software adaptation (by repeating the configuration process **bottom-up**, taking the changed hardware into account).

One major problem with the **top-down** configuration approach is the fact that in practice lower infrastructure layers are additionally affected by environmental constraints that need to be manually specified by the user (e.g. given hardware setup, business decisions selecting a specific communication protocol).

The **bottom-up** approach is also far from ideal: As the upper layers’ configuration tend not to be sufficiently specified, the configuration tool needs to fall back to a default configuration. Moreover, the user is probably most interested in the top layer (application) SPL (as it contains the actual business logic the whole software composition is in existence for), which cannot be directly influenced anymore.

A logical consequence would be a **combination of both**: The user could, for example, configure the topmost and the bottommost layers (or: the bottom layer is determined by some given hardware setup), and the iterative configuration process then should proceed in both directions. As additional conflict potential emerges where both directions meet, more thought would have to be put into conflict resolution, probably propagated back to the layers where the user conducted manual configuration.

Generalised further, the user should be able to make configuration decisions on all layers he is proficient in. The explicitly manually selected or deselected features could be called “configuration seeds”, as they are the origin for a dependency resolution process propagating through the layer-spanning web of feature constraints.

### 5. Horizontal Composition

Orthogonally to vertical composition, instances of the **same** SPL may exist side by side (cf. Figure 2), still con-
straining each other because they will be deployed ...

- ... on the same machine: In such a scenario, a possible constraint could be, e.g., that the combined memory consumption must not exceed a certain amount (a classical non-functional constraint), or that at most one application may access the display device (functional).

- ... on the same network: Instances must, e.g., use the same communication media, use the same communication protocol, or offer a compatible set of functions to a client (e.g., sensor nodes with master nodes). SQL sent across the network: If the master node may issue a certain JOIN type, data management in the sensor nodes must be able to serve that request.

- ... or generally, in the same environment.

Generalised, the following analysis is also valid for instances of different SPLs that share common properties (and a common standard feature model, cf. Subsection 4.1 “explicit modeling”) and a common environment.

5.1. Modeling Feature Constraints

For a very limited set of cases, “normal” feature constraints as described in Subsection 4.1 also suffice for horizontal composition of SPLs, using explicit modeling. In these cases, the number of SPL instances to be composed must be known in advance, and these instances must be uniquely identifiable (by giving them names, for instance). Thereby, feature constraints can be formulated referring to a specific, neighbouring instance (“Feature ‘TCP/IP’ of ‘Client1’ requires Feature ‘TCP/IP’ of ‘Server1’, ‘Server2’ and ‘Server3’.”). Additionally, nesting of SPL instances is not possible.

Unfortunately, this scenario is unrealistic, and the outlined feature constraint modeling technique is very restricted and inflexible. The following issues must be addressed by an extension to feature modeling that supports horizontal composition of SPLs:

- The total number of SPL instances to be composed may be unknown at constraint modeling time.

- The system architecture may call for SPL instance nesting, and respectively SPL instance hierarchies.

- Constraints may not only exist among SPL instances, but among groups of SPL instances. These groups or subgroups must be specifiable. Groups also include specific named groups (roles), e.g., “All instances that have the role ‘server’.”.

- As non-functional properties are especially important in the embedded systems domain, non-functional constraints on the composed system must also be expressible. This could mean that these constraints (“The byte size of the flash image containing all application instances for node X must not exceed 256KiB.”) must be reducible to (previously measured [12] or otherwise computable) non-functional information of the single SPL instances that comprise the whole (“The size of the composed flash image is the sum of all instance sizes.”).

As classical feature modeling does not provide a solution for these issues, additional expressiveness needs to be introduced.

5.2. Discussion

An interesting approach would be to enhance feature modeling in a way that only minimal changes to current variant management tools (such as pure::variants [2]) are necessary to deal with composition of multiple SPLs. Although dedicated composition models, such as Entity Relationship Models or UML notations, have been proven to be
applicable in this area [5], they lack support in the tools that are already widely in use.

Predicting non-functional properties especially of composed systems may very well not always be possible: E.g., giving real-time guarantees for communicating processes or predicting the resulting binary size when an optimising linker might remove commonly used/unused functionality seems infeasible. Alternatively, a worst-case approach could be considered (similar to Worst-Case Execution Time in RT systems), giving upper bounds for these properties.

6. Related Work

Frieß et. al. describe feature models with cardinalities [5], allowing for horizontal/vertical composition, nesting (“part of” relationship) of SPL instances, and group conditions. In their approach, the composition model is a combination of feature configurations, attribute restrictions and group conditions with “uses”/“part of” relationships and cardinalities. They do not discuss the possibility to enhance classical feature modeling to support SPL composition.

Clements and Northrop discuss both COTS (Components Off The Shelf) integration and the need for variability within SPL components [3], but do not go into detail how variability in COTS should be dealt with, especially in regard with feature modelling and automatic configuration.

Streitferdt et. al. [15] use the Object Constraint Language to formally describe complex inter-feature constraints, but lack a discussion on how to extend that approach on the composition of multiple SPLs. Genßler and Zeidler describe an approach for composition of software modules in the embedded systems domain [6], also focusing on the composition formalisms. Especially their component model, which addresses non-functional properties and containment relations, should be considered in future SPL composition research.

7. Conclusion

Although composition of software is a classical research field, little is known about composition of SPLs and the implications for feature modeling, inter-SPL constraints, and the implications for the configuration process. As we believe that in the future even more software is going to be developed as an SPL, multi-layered SPL architectures will be in common use. Facing a vastly growing configuration complexity, we believe that the experiences from single SPL variability modeling and automatic configuration need to be transferred to the multi-layer situation to be controllable at all.

We showed different architectures of composed SPLs and proposed means to extend existing variability meta models and configuration processes for complex software systems composed from several SPLs. The identified problems in variability modeling and automatic configuration for vertical and horizontal SPL composition pose challenges for research and experiments in this area. Should they prove feasible, the solutions we outlined will take the field forward towards composition of SPL instances in a similar way single SPLs were configured and normal software components were composed before.

More interesting future work includes a closer analysis of the lower layers in an multi-tier SPL architecture, including (statically or dynamically) configurable hardware and a software stack building upon and adapting to it.

References


[8] Daniel Lohmann, Fabian Scheler, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. PURE embedded


