

# Automating the Configuration of Multi Software Product Lines

Marko Rosenmüller, Norbert Siegmund  
School of Computer Science  
University of Magdeburg, Germany  
{rosenmue,nsiegmun}@ovgu.de

**Abstract**—The increased use of software product lines (SPLs) results in complex software systems in which products of multiple SPLs communicate and interact with each other. Such a system of interdependent SPLs has to be considered as a whole to achieve compatibility between different SPL instances. In this paper, we present an approach to design and configure *multi software product lines (MPLs)*, i.e., product lines that consist of multiple interdependent SPLs. Therefore we use *composition models* that describe how an MPL is composed from multiple SPL instances. This allows us to automate the configuration of MPLs which is required to handle the resulting complexity. We also show how to automatically derive configuration generators to further simplify the configuration process and we report from experiences of applying the presented approach.

## I. INTRODUCTION

*Software product lines (SPLs)* enable reuse by generating software from a common set of assets, e.g., by composing components [5]. The *instances* of an SPL, i.e., the products, can be programs, libraries, and also components. Hence, products of an SPL can also be used for building more complex SPLs [17]. For example, a component can be developed as an SPL and can be combined with other components in a larger SPL. Due to the success of SPLs, more and more programs are developed as product lines and integrated in complex systems. This results in *product lines of product lines* or *nested product lines* [11]. We call arbitrary compositions of SPLs *multi software product lines (MPLs)*.

As an example for an MPL, consider a sensor network (SNW) that consists of network nodes (SNW-Nodes) which are small embedded devices. The software running on network nodes differs in functionality because the nodes have to accomplish different tasks: there are *sensor nodes* for sensing data, *access nodes* that provide access to the sensor network, and *data nodes* that aggregate and store data [12], [14]. Developing an SPL for network node software, allows a user to generate tailor-made variants for the different node types. The software of a single network node consists of multiple programs and libraries. For example, a data management system might be used for data storage, cryptographic libraries ensure confidentiality of data, and communication libraries might be used to provide basic communication functionality. These programs or libraries are more and more developed as SPLs to increase reuse and to minimize the functional overhead especially needed in the domain of embedded systems. This results in SPLs that *use* other SPLs for their realization.

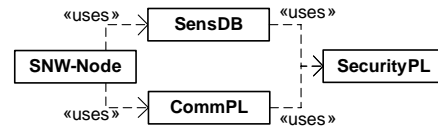


Fig. 1. Uses relationships between different product lines.

An example for this *uses* relationship between SPLs in a sensor network is shown in Figure 1. The SNW-Node SPL uses an SPL for data storage (SensDB) and a communication product line (CommPL). Product lines CommPL and SensDB encrypt transferred and stored data using the security product line (SecurityPL) which provides cryptographic algorithms. Hence, there are four interdependent SPLs and each of them has to be configured according to the requirements of the other SPLs.

In contrast to a single SPL, we have to consider the functional dependencies between the different instances of all involved SPLs. That is, modifying the configuration of one instance might require a different configuration of other SPL instances. Already manual configuration of single SPLs is highly complex and error prone. Manual configuration of large networks of interdependent SPLs might be impossible if many features and dependencies between features are involved. Furthermore, the configuration process has to be repeated when the configuration or the implementation of one SPL changes. Ideally, a user only has to configure one SPL that describes the whole application scenario, e.g., a sensor network product line, and does not have to care about implementation details of underlying SPLs. A solution could be to integrate multiple feature models into a single feature model. This, however, results in a large feature model that mixes different domains and provides configuration options that are not important for the problem domain.

In this paper, we present an approach to automate the configuration of MPLs based on *composition models* as described in [13]. A composition model integrates multiple SPLs by describing for each SPL which instances of other SPLs it *uses*. It thus describes dependencies between concrete SPL instances. In order to integrate composition modeling in the SPL engineering process, we show how a composition model can be semi-automatically derived from the domain models

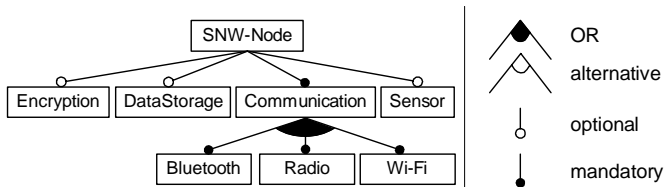


Fig. 2. Feature diagram of a sensor network node (SNW-Node) SPL.

of the SPLs of an MPL. Finally, we present an approach to generate configuration generators for MPLs, i.e., programs that are used to derive configurations for all SPL instances of an MPL.

## II. MODELING MPLS

While SPL engineering is well understood and programs can be automatically generated from SPLs, the configuration of multiple interdependent SPLs (i.e., MPLs) is mostly not considered. In the following, we describe how the feature modeling approach can be used for modeling MPLs. However, we will show that feature models do not provide a suitable solution for describing dependencies in arbitrary MPLs, e.g., when multiple instances of the same SPL are used in an MPL. Furthermore, using only feature models for MPL modeling results in complex solutions. For that reason we introduce composition models and show how they overcome existing problems.

### A. Feature Models

An SPL is used to create similar programs that share a common set of *features*. The features of an SPL are distinguishable characteristics that are of interest to some stakeholder [5]. SPLs can be described using *feature models* that are often visualized using *feature diagrams* [9], [5]. An example for a sensor network node (SNW-Node) product line is depicted in Figure 2. A concrete program or *instance* of an SPL is defined by a selection of required features. *Domain constraints* of a feature model are used to ensure only valid feature combinations in an SPL configuration. For example, *requires* and *mutual-exclusion* relations are used to describe dependencies between features [5]. In general, arbitrary propositional formulas might be used [2].

Domain constraints can not only be used to describe dependencies within a single SPL but also between different SPLs [6]. For example, a *requires* constraint

$$SnwNode.Bluetooth \Rightarrow CommPL.Bluetooth \quad (1)$$

describes that when a user selects feature `BLUETOOTH` of the SNW-Node SPL (*SnwNode.Bluetooth*; cf. Fig. 2) also feature `BLUETOOTH` of the communication framework SPL (*CommPL.Bluetooth*; cf. Figure 1) has to be selected. However, if multiple instances of the same SPL are required, there can also be constraints between SPLs that cannot be described on the domain level. For example, communication between nodes in a sensor network, requires support for the same

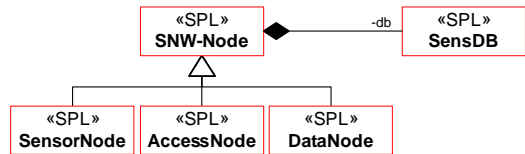


Fig. 3. Composition model with an SPL for sensor network nodes (SNW-Node) and specialized SPLs SensorNode, AccessNode, and DataNode. The SNW-Node SPL uses an instance of a sensor database (SensDB).

communication protocol in the nodes (e.g., `RADIO` in Fig. 2). That is, two instances of the same SPL (e.g., one instance for sensor nodes and one instance for access nodes) have to be configured to support the same protocol. A domain constraint cannot describe this dependency because SPL instances cannot be distinguished in the domain model.

A similar problem occurs when an SPL  $A$  uses two different instances  $b_1$  and  $b_2$  of SPL  $B$ . Since the domain model does not include SPL instances, it is hard to use domain constraints to describe such dependencies. For example, to describe that feature  $f_3$  of instance  $b_2$  has to be selected whenever feature  $f_1$  of SPL  $A$  is selected, we might use a listing of the distinguishing features of  $b_2$  as part of the constraint:

$$A.f_1 \wedge (B.f_1 \wedge B.f_2) \Rightarrow B.f_3 \quad (2)$$

where  $(B.f_1 \wedge B.f_2)$  describes instance  $b_2$ . At first glance, this correctly represents the required constraint; however, it is also valid for other instances of  $B$  that include features  $f_1$  and  $f_2$  which is not intended. A direct representation of SPL instances could avoid such problems and simplify the configuration at the same time. Furthermore, a constraint as shown in Equation 2 is not part of the problem domain of SPL  $A$ . Actually, it is an implementation issue of  $A$  and for a different scenario SPL  $B$  might be replaced by a different SPL. Hence, this implementation knowledge should be separated from the domain model of  $A$ .

### B. Composition Models

In order to overcome the presented problems, we introduced *composition models* that describe an MPL by modeling a composition of multiple SPL instances [13]. This allows us to describe the dependencies of all SPLs and to provide means for automatically configuring the SPLs according to these dependencies. In the following, we review composition models and show how these are used to describe MPLs.

*SPL Instances:* A composition model uses the concept of aggregation of classes known from OOP to represent the uses relationships between SPLs. Each class represents an SPL and a class instance represents an instance of an SPL. In Figure 3, we depict the UML representation of a composition model for the sensor network example. SNW-Node and SensDB are SPLs and the aggregation relation between them denotes that SNW-Node uses an instance of SensDB with name `db`.

*Specialization:* In product line engineering, *specialized SPLs* are used to describe a subset of the variants provided by an SPL [6]. We include SPL specialization into composition

models using inheritance between SPL classes. This allows us to easily reuse SPL configurations in different MPLs. In Figure 3, *SensorNode*, *AccessNode*, and *DataNode* are specializations of the *SNW-Node* SPL. For example, *SensorNode* is a partial configuration of *SNW-Node* in which feature *SENSOR* is mandatory.

*Constraints:* In order to achieve compatibility between SPL instances of an MPL, we use composition model constraints. A constraint in a composition model is a propositional formula<sup>1</sup> consisting of features similar to a feature model constraint [2]. A feature in that expression can be referenced using the name of an SPL or the name of an SPL instance. When using an SPL name, it refers to the feature in all instances of that SPL (i.e., a usual domain constraint between feature models) and when using the name of an SPL instance, it refers only to a feature in that concrete instance which we call *instance constraint*. The constraint shown in Equation (2) thus simplifies to  $A.f_1 \Rightarrow b_2.f_3$ , where  $A$  denotes an SPL and  $b_2$  denotes an SPL instance. If needed, instances can be fully qualified with the name of the SPL an instance belongs to. For example,  $A.b_2$  refers to instance  $b_2$  defined in SPL  $A$ . Constraints might also include specialized SPLs. For the sensor network example in Figure 3, we can specify constraints:

$$SensorNode.Radio \Rightarrow AccessNode.Radio \quad (3)$$

$$SnwNode.Encryption \Rightarrow SnwNode.db.Encryption. \quad (4)$$

Constraint (3) describes a dependency between *SensorNode* and *AccessNode* specializations of the *SNW-Node* SPL. Selecting feature *RADIO* in *SensorNode* requires to select feature *RADIO* in all instances of *AccessNode* to ensure a compatible communication protocol between sensors and access nodes. In contrast, instance constraint (4) addresses only the *SensDB* instance  $db$  defined in *SNW-Node*. Hence, different instances of *SensDB* can be configured differently, e.g., with or without encryption, depending on the kind of the node.

Constraints are part of an SPL and are inherited when creating a specialized SPL. For example, SPL *SensoreNode* in Figure 3 inherits all constraints stored in *SNW-Node*, e.g., constraint (4). Constraints can also be redefined in a specialized SPL. However, since a specialization represents a subset of the variants represented by its parent, a constraint can only be redefined by adding propositions using a conjunction. That is, constraint redefinitions can only reduce the number of possible variants.

*Conditional Dependencies:* Sometimes an SPL instance of an MPL is only required when some optional feature is available in a configuration. For example, the instance of *SensDB* in Figure 3 is only needed when feature *DATAS-*

<sup>1</sup>First order logic might also be used, e.g., to provide constraints for sets of SPL instances. However, we think that propositional logic might be sufficient because a limited number of SPL instances should be the usual case and thus quantification is not required. Nevertheless, when using more complex constraints in domain models [7] the same kind of constraints should be used for composition models.

*TORAGE* of *SNW-Node* is selected (cf. Fig. 2). To describe this, we use *conditional dependencies* which define optional SPL instances that are only needed when a particular feature or a set of features is present in a configuration [13]. This simplifies the configuration process because the SPL only has to be configured when the according feature is selected.

### III. AUTOMATING THE CONFIGURATION OF MPLS

In order to automate MPL configuration and to integrate MPL modeling into the development process for SPLs, we extend the product line engineering process with composition modeling which is part of the domain design process. A composition model connects domain model and implementation of an MPL by describing the SPL instances used to implement an MPL. As illustrated in Figure 4, the following steps are required for creating MPLs:

- creating a feature model for an MPL,
- generating and refining composition models,
- deriving a configuration generator.

In the following, we describe the required steps in detail and discuss experiences when creating the example sensor network MPL in Section IV.

#### A. Feature Models for MPLs

We describe an MPL using the feature models of all contained SPLs and a composition model that defines how the SPL instances are combined. Depending on the SPLs, we can differentiate between *hierarchical* and *flat* MPLs.

*Hierarchical MPLs:* The dependencies between the SPL instances of an MPL often result in a *hierarchy of SPLs*. In Figure 3, *SNW-Node* uses an instance of *SensDB* and thus defines a hierarchy between both SPLs. The dependency between the SPLs is directed: *SNW-Node* depends on *SensDB* but *SensDB* is independent of *SNW-Node*. An SPL hierarchy may have multiple levels, e.g., as shown for *SNW-Node* in Figure 1, resulting in a three level hierarchy. We call *SNW-Node* the *top-level SPL* of the hierarchy because there is no other SPL that depends on *SNW-Node*. Usually, we can describe the variability of a whole MPL using the feature model of the top-level SPL.

*Flat MPLs:* There can also be MPLs that do not exhibit a hierarchy which we call *flat* MPLs. Examples are MPLs of communicating programs such as in a client-server architecture where client and server are developed as distinct SPLs and have to be configured to achieve compatibility. For example, a mail client and a mail server have to support the same communication protocol, e.g., IMAP. Flat MPLs also occur when multiple instances of the same SPL are combined in an MPL. For example, an MPL of replicated DBMS stores data in a master DBMS and in a slave for replication. Such a system can be developed as a single SPL from which differently configured instances for master and slave can be generated. The MPL thus consists of multiple instances of the same SPL. Usually, the dependencies between the SPLs of a flat MPL are not directed but each of the SPLs depends on the other. In contrast to hierarchical SPLs, there is no top-level

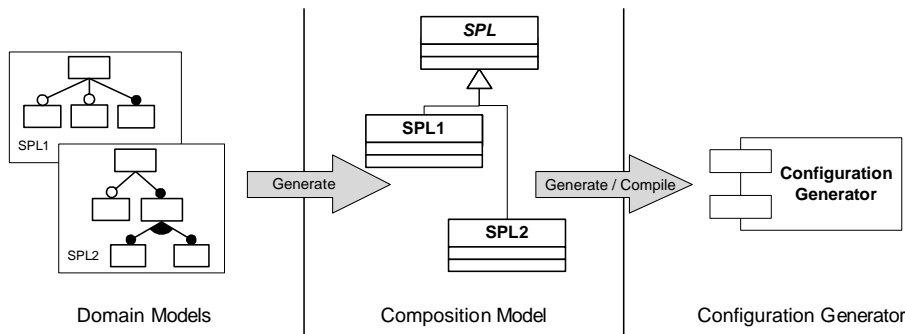


Fig. 4. Generating composition models and configuration generators for MPLs.

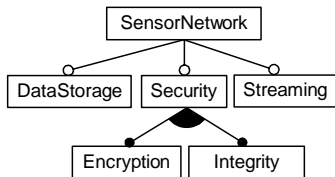


Fig. 5. Feature diagram of the SensorNetwork MPL.

feature model that describes the MPL. Imposing a hierarchy by defining one or the other SPL as the top-level SPL is usually not sufficient because one SPL does not describe the whole MPL and variability of the other SPL is hidden. For example, the feature model of a mail server is inappropriate for configuring the whole MPL, i.e., mail server and mail client. A better solution is to introduce a hierarchy by creating a new top-level feature model for the MPL. This way, we can describe the variability of the whole MPL using only a single feature model while variability of the constituent SPLs is hidden, e.g., a feature model for a replicated DBMS that describes the whole MPL and abstracts from underlying SPL instances for master DBMS and slave for replication.

In the sensor network example, a product line of sensor network nodes (cf. Fig. 1) is a hierarchical MPL since it uses multiple other SPLs. In a complete sensor network scenario, however, there are different specializations of network nodes used (SensorNode, AccessNode, DataNode) which results in a flat *sensor network MPL*. We thus create a new top-level feature model that represents the whole sensor network, as depicted in Figure 5. It describes which functionality the sensor network provides and abstracts from the underlying specializations of the SNW-Node SPL. We use features like `DATA STORAGE` to represent an optional data node of the sensor network which is implemented by the specialized DataNode SPL. Which features are to be included in an MPL feature model we discuss in Section IV.

### B. Creating Composition Models

Based on the feature models of an MPL, the composition models can be created in a step-wise manner. In fact, this will be the usual scenario because a developer of an SPL will create a composition model for her SPL that can be

reused in higher level SPLs and hides underlying SPLs. Each composition model defines the directly used SPL instances and is independent of higher level SPLs which is important for reusing the model. For example, we can easily replace SensDB (cf. Fig. 3) with a different DBMS product line. This requires to store the composition model of each SPL separately. The composition models of different SPLs are implicitly connected via the uses-relationships between SPLs defined as instance variables (e.g., instance `db` in Figure 3). Hence, the compound composition model of an MPL is the union of all composition models of the underlying SPLs. Replacing an SPL thus only requires to modify the instance variable that defines the type of an SPL instance, e.g., modify the `db` instance of SNW-Node to be of a type other than SensDB.

As shown in Figure 4, we use a composition model generator to create initial composition models and their UML representation. The model can be created at any time in the design process and can be updated when SPLs change or different specialized SPLs become available. For generating composition models, we use the integration of C# and UML in Microsoft Visual Studio 2008. The generator creates a C# class for every SPL and an inheritance relation to represent SPL specialization. The use of the partial class concept of C# and the integration with UML class diagrams allows an SPL developer to edit the composition model using either the UML representation or the C# code while the generated corresponding representation is automatically updated. The classes of the composition model are subclasses of an abstract SPL class, as shown in Figure 6. This abstract class implements generic functionality of the configuration generator as we describe later.

A composition model can be visualized using a UML diagram which is also used by the SPL developer to define SPL instances. An integrated visualization of a complete MPL composition model can be generated by including all SPL classes in a single UML diagram, as shown for the sensor network example in Figure 7. Such an integrated view of the whole model is usually not necessary, since it is sufficient to edit the models of all SPLs separately.

*SPL Instances and Constraints:* The initially generated composition model is refined by an SPL developer who creates instance variables to represent the uses-relationships between

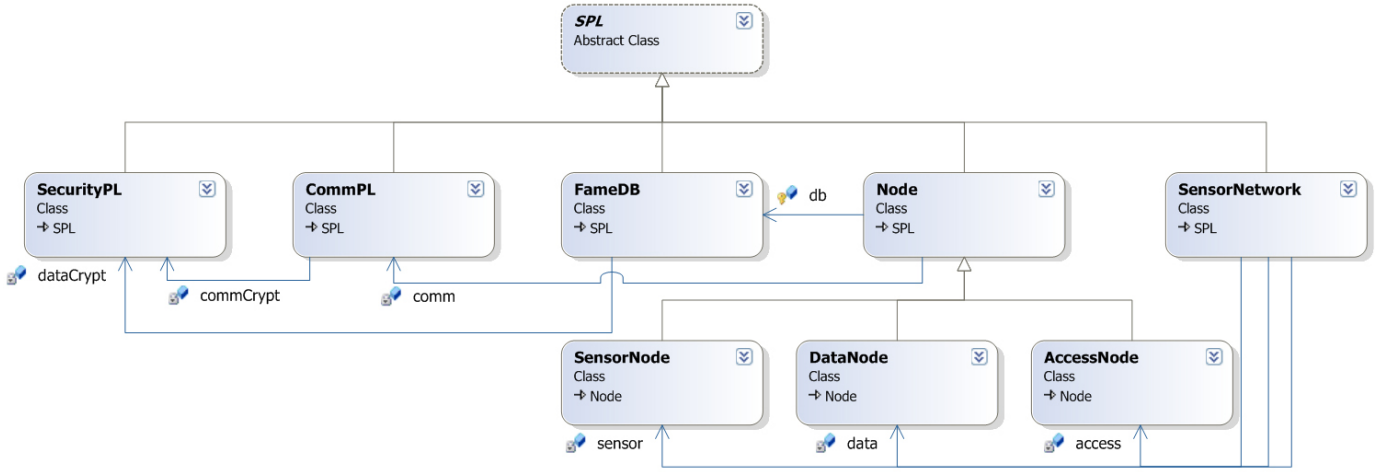


Fig. 7. Screenshot of the compound composition model for the SensorNetwork MPL.

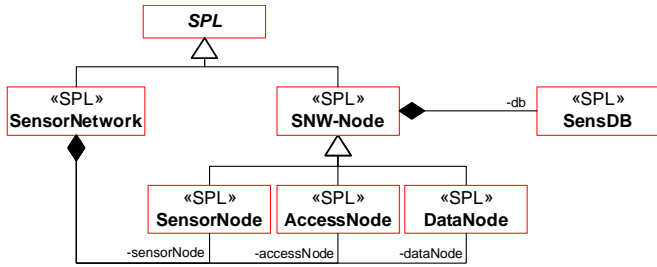


Fig. 6. A composition model for a SensorNetwork SPL that uses a network node SPL (SNW-Node).

SPLs. The SPL instance variables connect individual composition models. For example, instance variable `sensorNode` of the SensorNetwork SPL in Figure 6 represents an instance of the specialized SPL `SensorNode` and connects both SPLs. SPL instances can be added via the UML representation of a composition model or by directly changing the source code of the SPL classes.

A domain modeler creates composition model constraints to define which features from used SPL instances are required by a higher level SPL depending on its configuration. As discussed above, constraints are arbitrary propositional formulas between features of the involved SPLs but may refer only to concrete SPL instances if required. For example, we can define constraints that refer only to the `dataCrypt` instance of `FameDB` and do not affect the `commCrypt` instance of `CommPL` (cf. Fig. 7). Finally, conditional dependencies are created to represent optional SPLs that are only needed when a particular feature is selected. For example, a data node is only needed when a user selects the `DATA STORAGE` feature of the SensorNetwork SPL.

### C. Configuration Generators for MPLs

Having an MPL, described by feature models of all SPLs and an integrated composition model, we can automatically derive a configuration generator. This configuration generator

is used in an interactive configuration process and asks a user for configuration decisions (i.e., feature selections) required to configure all SPL instances of an MPL and checks for violation of constraints.<sup>2</sup>

We implemented a generic configuration generator using C#. The configuration generator provides a graphical user interface that asks a user for feature selections of all required SPL instances for an MPL scenario. It checks for violations of model constraints and creates the SPL instances required for an MPL configuration. Parts of the generic implementation are provided by abstract class `SPL`, as shown in Figure 4. The generic implementation of the configuration generator is extended by MPL specific code which is the composition model of the MPL (subclasses of `SPL` in Fig. 4 and 7). Each MPL specific class extends the functionality of the abstract SPL class by defining instance variables for the used SPL instances. The configuration generator code (generic code plus MPL specific composition model) is compiled into an executable program.

## IV. EXPERIENCE REPORT AND DISCUSSION

In the following, we shortly report about our experience of modeling and configuring MPLs using the sensor network example. Since we observed different possible ways for creating feature models for MPLs and defining constraints, we discuss benefits of the respective solutions.

### A. Handling MPL Variability

For the sensor network example, we defined specialized SPLs of the SNW-Node SPL (`SensorNode`, `DataNode`, and `AccessNode`), as shown in Figure 6. Since it is a *flat* MPL, we created a feature model that represents the whole MPL as already described (cf. Fig. 5). We think that such an additional feature model for flat MPLs is often useful for several reasons:

<sup>2</sup>Currently, the configuration generator does not support redefinition of model constraints in specialized SPLs.

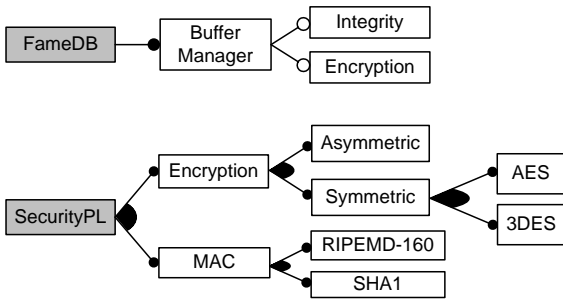


Fig. 8. Excerpts of the feature diagrams of FameDB and SecurityPL.

- It provides a simple structure for an MPL and simplifies the configuration process because only a single feature model is needed. Moreover, some of the features of the SPLs are not important for configuring the MPL, e.g., because they are mandatory in the particular MPL.
- It reduces configuration options, e.g., by creating features that imply a set of features of lower level feature models.
- It avoids invalid configurations, e.g., it implicitly defines constraints that otherwise had to be expressed as propositional formulas.
- It allows developers to define new features representing features of lower level feature models. For example, it is simpler for a domain expert not familiar with cryptography to configure a feature ENCRYPTION instead of a feature AES that represents the encryption algorithm.

An MPL feature model is useful to reduce the configuration space to valid combinations of SPL instances and to provide only configurations that are meaningful in a special context, e.g., to simplify testing and maintenance of an MPL by supporting only a predefined subdomain. However, restricting the variability using an MPL feature model also means that features important for a particular application scenario might not be available in the MPL's feature model. In order to be able to create configurations also for such scenarios a user can additionally configure the underlying SPLs.

For some MPLs, it might be useful to avoid an additional feature model, e.g. when most of the features of the lower level feature models have to be repeated in the MPL's feature model. Including all features in a single feature model is usually not a solution because it results in large feature models that are probably unmanageable. Furthermore, it is a question of additional effort for creating and maintaining the additional feature model.

### B. Composition Model and Constraints

Based on the feature models, we generated four composition models for SPLs CommPL, FameDB, Node, and SensorNetwork consisting of eight classes representing SPLs or specialized SPLs (cf. Fig. 7). We extended the composition model to define SPL instances and constraints. SPL instances can be defined by editing each model separately but also using an integrated view, as shown in Figure 7.

We use several constraints to ensure valid configurations of the sensor network SPLs. For example, the FameDB SPL requires only AES and SHA1 algorithms of the SecurityPL (cf. Fig 8) and does not use all provided algorithms. We describe this with *instance constraints* between SPLs FameDB and SecurityPL:

$$FameDb.Encryption \Rightarrow FameDb.dataCrypt.AES \quad (5)$$

$$FameDb.Integrity \Rightarrow FameDb.dataCrypt.SHA1 \quad (6)$$

Constraint (5) defines that the AES encryption algorithm of the SecurityPL is required when feature ENCRYPTION is selected in FameDB. Constraint (6) defines that the SHA1 hash algorithm is required when feature INTEGRITY is selected in FameDB. Instance constraints are stored as part of the composition model of the according SPL which simplifies their reuse. For example, constraints (5) and (6) are stored as part of FameDB and can be reused in other compositions of FameDB.

*Instance Constraints vs. SPL Specialization:* As an alternative solution to instance constraints, we can create specializations of an SPL. For example, we can create a specialization of SecurityPL that is used for data encryption and integrity only in FameDB. This specialization can already include features AES and SHA1 thus avoiding instance constraints.

In contrast to instance constraints, SPL specialization provides better means for reuse in other MPLs. The reason is that instance constraints are defined per instance variable (e.g., `FameDb.dataCrypt`) and cannot easily be reused. On the contrary, specialized variants can be reused simply by defining the corresponding instance variables. However, specialization does not scale because different application scenarios with different constraints result in an exponentially growing number of specializations. Hence, there is a tradeoff between simpler reuse of specializations and an increasing effort for defining and managing an increasing number of specializations. In general, it seems to be more favorable to create a specialized SPL when it can be used multiple times and includes many configuration decisions. Constraints are to prefer when reuse is limited and many specializations with few design decisions can be avoided. In many cases, this probably results in a mixture of specialization and instance constraints as we observed it for the sensor network.

### C. The Configuration Process

Based on domain and composition models, we automatically derive a configuration generator and use it to configure all SPLs used in an MPL. The configuration process starts with the top-level feature model and an empty feature selection. In our example, this is the SensorNetwork feature model. Based on mandatory features, the configuration generator creates a configuration for each instance variable and continues recursively. For example, SensorNode and AccessNode are always required in a sensor network and thus a `sensor` and `access` configuration is created according to the composition model in Figure 7. Initially there is no configuration for SPL DataNode because a conditional dependency defines that instance `data` of SensorNetwork is only available when

feature `DATA_STORAGE` is selected. This configuration is only added when the feature is selected by the user. Similarly, selecting feature `ENCRYPTION` adds an instance of `SecurityPL` to SPLs `FameDB` and `CommPL`.

When the user has provided a configuration for all `SensorNetwork` features, the configuration of the `SensorNetwork` SPL is finished. This should be the usual scenario, but a detailed configuration process of lower-level SPLs might be needed. For example, the `FameDB` SPL can be manually configured with features that influence the performance of the DBMS, like special index data structures. Such additional configurations are also required if configuration decisions are missing. Each time a user changes a configuration, the configuration generator checks the validity of the whole composition.<sup>3</sup>

#### D. Avoiding Code Duplication

For some application scenarios it is important to avoid code duplication of SPL instances that are used multiple times in an MPL. For example, `SensDB` and `CommPL` use an instance of `SecurityPL` for encrypting and decrypting data (cf. Fig. 7). In a scenario without product lines, e.g., using a library for security algorithms, we could reuse this library for data storage and communication, e.g., to reduce the binary size or used working memory of the resulting compound system. Reusing product lines in the same way is not always possible, because the required configurations of the instances might be contradicting. For example, when two alternative features of `SecurityPL` are needed (cf. Fig. 7) two different instances of the same SPL are required, one for `SensDB` and one for `CommPL`.

In order to optimize an MPL configuration with respect to code reuse, the configuration generator detects possible SPL instances that can be reused. However, it cannot always be decided automatically whether an instance can be reused or not. For example, even when reusing an SPL instance with more features than needed, the semantics of the reused SPL might be different, e.g., it might be required to initialize additional features. Furthermore, due to the feature interaction problem [4], the behavior of one feature might change when adding another feature. Finally, reuse is problematic if it means to reuse a running program or component including its internal state. In order to avoid such errors, a user can select whether an SPL can be reused or not. However, to provide a practical solution, there should be more sophisticated configuration mechanisms included in the future, e.g., using insights from component based software development to solve issues regarding the internal state of an SPL instance.

## V. RELATED WORK

As described in Section II, approaches to model SPLs can also be applied to model MPLs. Czarnecki et al. use cardinality-based feature models with constraints to specify specializations and constraints in feature models where multiple selections of one feature are possible [6], [10]. The used *feature model references* and *feature cloning* can also

be applied to model compositions of SPL instances as needed for MPLs. The tool *FeaturePlugin* furthermore allows for configuring multiple feature models that are integrated via feature model references [1]. Including multiple feature models into a single MPL feature model via feature model references results in highly complex domain models for large MPLs. Furthermore, it mixes domain modeling with SPL implementation because lower level SPLs are used for *implementing* higher level SPLs and the domains are only related due to the implementation. An approach that integrates feature models of different hardware and software product lines was presented by Streitferdt et al. [15]. The presented integration of multiple product lines does not consider SPL instances or constraints between them which is not needed in their context.

In contrast to the modeling approaches presented above, we propose to model SPL instances and dependencies between them. Our approach is an extension of existing SPL modeling techniques and we think that their combination is required to sufficiently model MPLs. We think that domain constraints are required for describing dependencies between the SPLs of an MPL but implementation issues like constraints to lower level SPLs, used for implementation of higher level SPLs, should be handled separately, i.e., in composition models.

Czarnecki et al. showed that feature models provide less descriptive power than ontologies but are easier to use due to their specialization [7]. The relationship of composition models to ontologies is similar: Compositions of SPL instances can also be modeled using ontologies and a composition model is a special view representing dependencies between related SPLs. Hence, feature models and composition models are special techniques with a focus on certain aspects of SPL modeling. Ontologies might be used to integrate both for a more complete description of an MPL.

*Product populations* built from Koala components are described by van Ommering [17]. Koala components can be built by composing smaller components at configuration time which is different from our work. We aim at describing how SPLs have to be composed to build a larger product line, i.e., an MPL, and not concrete products, i.e., Koala components. With our approach, the composition of a complex product (e.g., a component), built from other products, automatically changes depending on a feature selection, which is a modification of the composed architecture. This is different from manual composition of components to derive a larger component.

Fries et al. present an approach to model SPL compositions for embedded systems [8]. They use *feature configurations*, a selection of features, to describe a group of instances that share these features. Feature configurations are similar to specialized SPLs in staged configuration but do not allow a user to describe multiple configuration steps. A composition model, as described in [8], is defined for a complete composition of product line instances. We create a composition model for each SPL of an MPL and integrate them to describe MPLs which eases reuse of composition models. Implementation issues or reusing SPL instances are not addressed in [8].

SPLs consuming different other SPLs in a SOA environment

<sup>3</sup>Currently, we detect violation of constraints and do not check satisfiability.

are described by Trujillo et al. [16]. Their focus was on modeling the interfacing between SPLs in a service-oriented environment. This includes service registration and service consumption.

Tools like *pure::variants*<sup>4</sup> and *Gears*<sup>5</sup> allow a domain engineer to build feature models and describe dependencies among them. Both tools support modeling dependencies between SPLs and *Gears* explicitly supports nested product lines that can be reused between different feature models. *guidsl* is a tool to specify composition constraints for feature models using a grammar [2]. It provides means to check models and interactively derive a configuration for a feature model. All these tools support model checking using constraints similar to our approach. However, they do not consider different instances of an SPL within one composition and *guidsl* does not consider the integration of multiple SPLs at all.

Batory et al. have shown that SPL development using layered designs scales to *product lines of program families* [3]. The focus of their work is on generating families of programs from a single code base and reasoning about program families. The work does not address composition of SPLs developed independently or compositions of SPL instances.

## VI. SUMMARY

With increasing importance of SPLs, techniques to model, develop, and configure compositions of multiple interacting SPLs have to be provided. In this paper, we presented an approach to model and configure such *multi software product lines* (MPLs) that are built from multiple interdependent SPLs. The presented work is a first step to extend current SPL engineering with a new process that describes the implementation of MPLs on an abstract level using a composition model of involved SPL instances. We thus bridge the gap between domain modeling and implementation of MPLs using a high-level abstraction.

We have shown that composition models for MPLs can be generated which reduces the effort for modeling MPLs and integrates composition modeling into product line engineering. Based on domain and composition models of an MPL, we can automatically create configuration generators that help to automate the configuration process of MPLs. In order to reuse composition knowledge, we use a separate composition model for each SPL that can be easily used in different MPLs. The composition model of a whole MPL is derived by combining the composition models of all constituent SPLs.

As a next step, we want to evaluate the approach using existing product lines and analyze how the configuration process of large MPLs can be further simplified. For example, by checking satisfiability we could provide early feedback to the user that configures an MPL.

## ACKNOWLEDGMENT

The work of Marko Rosenmüller is funded by German Research Foundation (DFG), project number SA 465/34-1.

<sup>4</sup><http://www.pure-systems.com>

<sup>5</sup><http://www.biglever.com>

Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C. This work is part of the projects MultiPLe<sup>6</sup> and ViERforES<sup>7</sup>.

## REFERENCES

- [1] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-in for Eclipse," in *Eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. ACM Press, 2004, pp. 67–72.
- [2] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proceedings of the International Software Product Line Conference (SPLC)*, ser. Lecture Notes in Computer Science, vol. 3714. Springer Verlag, 2005, pp. 7–20.
- [3] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating Product-Lines of Product-Families," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society Press, 2002, pp. 81–92.
- [4] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast," *Comput. Netw.*, vol. 41, no. 1, pp. 115–141, 2003.
- [5] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [6] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged Configuration Using Feature Models," in *Proceedings of the International Software Product Line Conference (SPLC)*, ser. Lecture Notes in Computer Science, vol. 3154. Springer Verlag, 2004, pp. 266–283.
- [7] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg, "Feature Models are Views on Ontologies," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society Press, 2006, pp. 41–51.
- [8] W. Friess, J. Sincero, and W. Schroeder-Preikschat, "Modelling Compositions of Modular Embedded Software Product Lines," in *Proceedings of the 25th Conference on IASTED International Multi-Conference*. ACTA Press, 2007, pp. 224–228.
- [9] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.
- [10] C. H. P. Kim and K. Czarnecki, "Synchronizing Cardinality-Based Feature Models and Their Specializations," in *European Conference on Model Driven Architecture Foundations and Applications (ECMDA)*, 2005, pp. 331–348.
- [11] C. W. Krueger, "New Methods in Software Product Line Development," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society Press, 2006, pp. 95–102.
- [12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.
- [13] M. Rosenmüller, N. Siegmund, C. Kästner, and S. S. ur Rahman, "Modeling Dependent Software Product Lines," in *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, no. MIP-0802. Department of Informatics and Mathematics, University of Passau, Oct. 2008, pp. 13–18.
- [14] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake, "FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems," in *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, Mar. 2008, pp. 1–6.
- [15] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow, "Configuring Embedded System Families Using Feature Models," in *Proceedings of Net.ObjectDays*. Gesellschaft für Informatik, 2005, pp. 339–350.
- [16] S. Trujillo, C. Kästner, and S. Apel, "Product Lines that Supply Other Product Lines: A Service-Oriented Approach," in *SPLC Workshop: Service-Oriented Architectures and Product Lines - What is the Connection?*, Sep. 2007.
- [17] R. van Ommering, "Building Product Populations with Software Components," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM Press, 2002, pp. 255–265.

<sup>6</sup>[http://www.witi.cs.uni-magdeburg.de/iti\\_db/research/MultiPLe](http://www.witi.cs.uni-magdeburg.de/iti_db/research/MultiPLe)

<sup>7</sup><http://www.vierfores.de>