Modeling Dependent Software Product Lines

Marko Rosenmüller, Norbert Siegmund, Christian Kästner, Syed Saif ur Rahman

School of Computer Science,
University of Magdeburg, Germany
{rosenmue, nsiegmun, kaestner, srahman}@ovgu.de

Abstract

Techniques to model software product lines (SPLs), using feature models, usually focus on a single SPL. Larger SPLs can also be built from smaller SPLs which results in a dependency between the involved SPLs, i.e., one SPL uses functionality provided by another SPL. Currently, this can be described using constraints between the involved feature models. However, if multiple differently configured instances are used in a composition of SPLs, dependencies between the concrete instances have to be considered. In this paper, we present an extension to current SPL modeling based on class diagrams that allows us to describe SPL instances and dependencies among them. We use SPL specialization to provide reuse of SPL configurations between different SPL compositions.

1. Introduction

Reuse in *software product lines (SPLs)* is achieved by combining assets, e.g., components, to produce a number of similar programs [4]. The resulting concrete products of an SPL (*SPL instances*) are variants tailored to a specific use-case or environment. Large SPLs can be built by reusing functionality provided by smaller SPLs and sometimes functionality of multiple SPLs is integrated into one SPL [18]. This results in a composition of SPLs where compatibility between interacting SPLs has to be ensured. As an example, consider a mail application developed as an SPL (MailClient in Figure 1). The client uses mail communication functionality provided by a MailFramework SPL (e.g., different mail protocols) and two differently configured instances of list SPLs (SortedList and SynchronizedList). To ensure correct composition the MailFramework has to be configured according to the requirements of the MailClient. For example, using the IMAP mail protocol in the MailClient requires the MailFramework to provide this protocol. This is getting more complex if multiple product lines are involved, e.g., the mail client in Figure 1 uses two additional instances of a product line of list data structures that also have to be configured appropriately. Such systems can be seen as large SPLs composed from smaller SPLs, i.e., *product lines of product lines* or *nested product lines* [13]. Proper configuration of such *dependent SPLs* not only ensures compatibility but also reduces consumed resources by removing unneeded functionality, avoids unneeded dependencies to other programs, and can reduce the user interface.

A user who configures an SPL that depends on other SPLs is usually only interested in configuration decisions of her problem domain and not in the configuration of underlying SPLs. For example, configuring the MailClient should not involve configuration of the underlying MailFramework SPL. Hence, SPLs used within other SPLs should be automatically configured to match the requirements of the enclosing SPL and only functionality a user is interested in has to be configured manually. This is possible by defining constraints between dependent SPLs that enforce only valid combinations and can be automatically resolved at configuration time. Such constraints (e.g., *requires* constraints between MailClient SPL and MailFramework) can be described as constraints between the feature models of these SPLs [4]. However, if multiple similar variants of one SPL are used, constraints between concrete SPL instances (*instance constraints*) are needed. For example, the MailClient uses two different instances of a list SPL (cf. Fig. 1). These instances have to be be configured differently, i.e., one as a sorted list and one as a synchronized list. A domain level constraint between mail client SPL and list SPL as used in current domain modeling cannot describe this dependency.

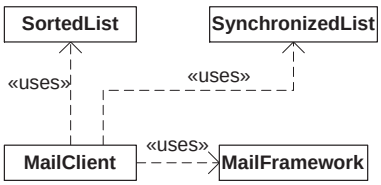


Figure 1. A MailClient SPL using a MailFramework SPL and different instances of a List SPL.

In this paper, we extend existing product line modeling with an approach that aims at modeling compositions of dependent SPLs. Our goal is to connect domain modeling and domain implementation: while feature models describe the features of an SPL we use SPL instance models to describe the composition of SPLs. Furthermore, we want to separate dependencies needed for SPL configuration, i.e., the uses-relationship between SPL instances, from concrete SPL implementation. Furthermore, we integrate domain modeling and SPL instances by mapping a feature of an SPL to instances of SPLs that are referenced by this feature. This is in line with *feature-oriented software development* where all software artifacts are decomposed with respect to the features of a domain [2]. By including *SPL specialization* [5] we are able to reuse SPL configurations in different SPL compositions. A combination of domain modeling and the presented instance modeling can be used to derive configuration generators that create instances of all dependent SPLs of a composition and thus provide the basis for an automated configuration process.

2. Software Product Line Engineering

In the following, we shortly present foundations of *software product line engineering (SPLE)* and the current state of techniques used to model and implement SPLs.

Domain Modeling. An SPL is used to create similar programs that share some common *features*. The features of an SPL are distinguishable characteristics of software that are of interest to some stakeholder [4]. As part of *feature-oriented domain analysis*

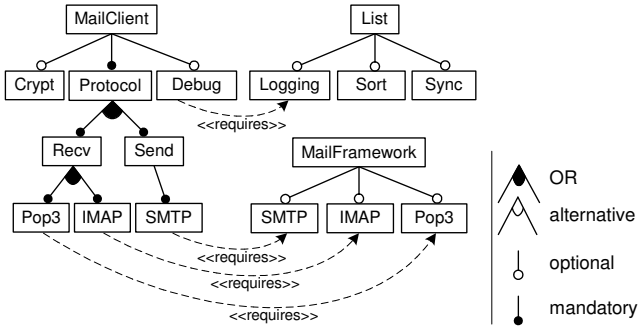


Figure 2. Feature diagram of a mail client SPL that uses a mail framework SPL with requires-constraints between SPLs (shown as dashed arrows).

(FODA), SPLs can be described using *feature models* [10, 4]. These are usually visualized using *feature diagrams* [10] as shown for a MailClient product line in Figure 2. The MailClient SPL uses other SPLs: a MailFramework SPL that provides different mail protocols and another small SPL of list data structures. The root of a feature diagram (e.g., node MailClient) represents the SPL itself and remaining nodes represent features of that SPL (e.g., feature IMAP represents the IMAP mail protocol). Features can be optional (depicted with an empty dot) or mandatory (depicted with a filled dot). Variability introduced by features provides means to create tailor-made applications. For example, mail clients using different protocols are created by including the according features IMAP, POP3, and SMTP.

Domain Constraints. Feature models often contain *domain constraints* that ensure only valid feature combinations on the domain level. For example, *requires* (shown as dashed arrows in Fig. 2) and *mutual-exclusion* relations are used to describe dependencies between features [4]. Domain constraints can also be used to describe dependencies between different product lines [6, 16]. For example, if feature IMAP is used in the MailClient, also feature IMAP of the MailFramework SPL is required (cf. Fig. 2). A user of the MailClient SPL usually only wants to configure the MailClient itself and not all accompanied SPLs which she might not have any domain knowledge of. This can be achieved by automatically resolving constraints between SPLs, e.g., between MailClient and MailFramework.

Product Line Implementation. SPLs are implemented using a variety of technologies. Examples are components that are combined to build large systems [3] or C/C++ preprocessor definitions used to build SPLs in the embedded domain. New paradigms like *aspect-oriented programming (AOP)* [11] and *feature-oriented programming (FOP)* [14, 2] can also be used to implement SPLs. The approach that we present in this paper is independent of the used implementation technique.

Based on the SPL implementation a user derives a concrete product by selecting the needed features from an SPL. The resulting *SPL configuration* (i.e., feature selection) is used to compose the corresponding software assets that implement an SPL resulting in a tailored *SPL instance*. The created SPL instance might be a library, a component, a program, or a collection of programs. The concrete composition mechanism depends on the implementation technique.

3. Dependent Software Product Lines

By using domain constraints, dependencies within an SPL and between different SPLs can be modeled. In the following, we show

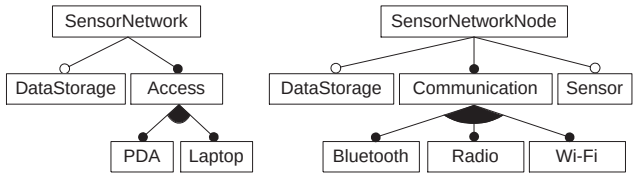


Figure 3. Feature diagrams of an SPL for a sensor network (left part) and an SPL for software used on sensor network nodes (right part).

that existing models have to be extended to completely describe arbitrary compositions of product lines and present requirements needed for an extension of current product line modeling.

Large Scale Product Lines. Complex and distributed systems, e.g., sensor networks, can be developed as product lines built from a number of heterogeneous SPL instances. For example, a SensorNetwork SPL as shown in Figure 3 may consist of different sensor nodes, data storage nodes, and access nodes each of them being an instance of a SensorNetworkNode SPL. Additionally, a client application accessing the sensor network might be developed as an SPL to support different client hardware (e.g., Laptops and PDAs) to interface with sensor network nodes. Dependencies between network nodes and client applications may exist to ensure a valid sensor network as a whole. Communication between sensor nodes, for instance, requires the same communication protocol and the access node of a sensor network might additionally require Bluetooth to communicate with clients (Laptop or PDA).

In contrast to the MailClient SPL, the SensorNetwork SPL is not an SPL from which a program is created but a number of interacting programs (i.e., the software running on nodes of the network and the client software to access the network). Hence, there might not be any source code needed for the SensorNetwork and only the smaller SPLs contain program code. This also affects the instantiation process: there is no particular composition process needed (e.g., using code transformation and compilation of code) but only instantiation of used product lines.

The SensorNetworkNode SPL again might use other SPLs that provide lower level functionality, e.g., an SPL for database management systems (DBMS) to store data. Hence, there can be chains of SPLs using instances of smaller SPLs. This composition might lead to large systems and also systems of systems. Each SPL in such a chain of SPLs requires an own model to describe dependencies to lower-level SPLs that it uses. By providing a separate composition model for each of these SPLs we can reuse these models in other product lines.

Compositions of Product Line Instances. Compositions of multiple SPLs imply that we have to handle these SPLs and constraints between them on the model level. Domain constraints can describe dependencies between different SPLs but do not take concrete instances into account. These instances, however, have to be considered if one SPL uses multiple differently configured instances of another SPL or if different instances of the same SPL depend on each other.

As an example consider our MailClient that uses multiple differently configured list data structures as shown in Figure 1. One instance of the List SPL is a synchronized List, i.e., using feature SYNC, and one is a sorted List, i.e., using feature SORT (cf. Fig. 2). In such a composition, we describe the requires relationship between feature DEBUG of the MailClient and feature LOGGING of the List using a domain constraint. This is not possible for features SORT and SYNC because the MailClient requires two different instances, one using feature SORT and one using feature SYNC. That

OO-concept	SPL representation
class	SPL
object	SPL instance
class specialization	staged configuration
aggregation	uses-relationship of SPLs
type of member variable	type of SPL instance
name of member variable	name of SPL instance

Table 1. OO-concepts and the corresponding representation of concepts in product lines.

is, we cannot describe constraints that affect only a concrete instance of an SPL.

We can find another example in the sensor network scenario. In this case, differently configured instances of nodes (e.g., data storage nodes and sensor nodes) are communicating with each other and one instance (e.g., a sensor node) depends on the functionality of another instance (e.g., a data storage node). Again, we cannot describe the dependencies in the feature model, which is the same for all nodes, because we would refer to the same feature model and not a concrete instance of it. To solve this problem we propose to extend feature modeling with explicit modeling of SPL instances.

Instance Identification. Using multiple instances of one SPL requires assigning a unique name to each instance to identify the differently configured instances and define constraints between them. For example, we have to create a name for the synchronized and sorted list that are used by the MailClient. Furthermore, we can use these names on the implementation level of an SPL in order to create class instances (e.g., list nodes) that are part of the different instances of an SPL. For example, name spaces or packages can be used to identify the SPL instances on the source code level. The concrete technique used to identify instances (e.g., Java packages) depends on the SPL implementation and is outside of the scope of this paper.

4. An Extension of Product Line Modeling

We have seen that constraints between SPL instances are needed to ensure correct configuration for a number of dependent SPLs. To avoid manual implementation of these constraints at the source code level of an SPL we present an extension to current product line modeling that allows a domain engineer to describe SPLs and SPL instances and specify constraints between them.

Modeling SPL Instances. The term *instantiation* is used in product line engineering as well as in OOP. In product line engineering, creating an SPL instance means to derive a concrete product from an SPL. In OOP, classes are instantiated resulting in concrete objects. Czarnecki et al. compared SPLs to classes of OOP and SPL configurations to class instances [5]. We adopt this correspondence and model SPLs and SPL instances using classes and objects of OOP. This also means that class instantiation corresponds to SPL instantiation. Using the concept of aggregation furthermore allows us to have members within classes where the type of a member corresponds to an SPL and the object assigned to such a member corresponds to an SPL instance. *Staged configuration* of SPLs, i.e., specialization of the feature model [5], can be represented by specialization as known from OOP. This means, a specialized class C_B of class C_A corresponds to a specialized SPL S_B of SPL S_A. Also subtyping of SPLs and polymorphism can be applied: S_B is a subtype of S_A and variables of type S_A can refer to instances of S_A or S_B. We summarized all corresponding constructs in Table 1.

Based on the correspondence of OOP classes and SPLs we can use class diagrams to model SPL compositions. By using class

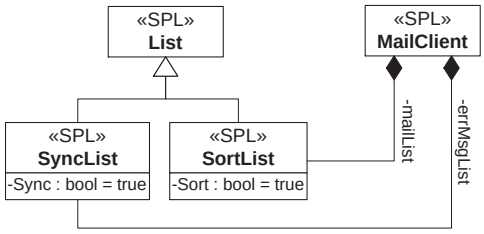


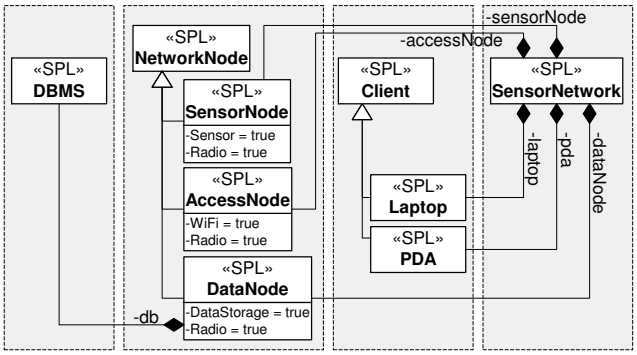
Figure 4. A MailClient SPL that uses different specializations of an SPL of list data structures (represented by aggregation). SPL specialization is represented by inheritance.

diagrams also complex compositions of SPLs can be created using existing tools and a familiar concept. Furthermore, existing support for generation of object-oriented code from class diagrams can be used to derive configuration generators from SPL composition models.

Using a class diagram, the MailClient example that uses a List SPL (cf. Fig 2) can now be modeled as shown in Figure 4. SPLs are represented by classes `MailClient` and `List`. Classes `SortList` and `SyncList` represent specialized variants of the `List` SPL that provide sorting and synchronization. The specialization, i.e., a pre-configured feature model, can be represented using special attributes of the classes (e.g., attribute `Sync` in class `SyncList`). Instances of SPLs used by other SPLs are described using aggregation, e.g., members `mailList` and `errMsgList` of class `MailClient` represent instances of different specialized List SPLs. By using specialized variants we can avoid constraints between `MailClient` and `List` that would be needed to define the different variants `SortList` and `SyncList`. Thus, we only have to refer to the specialized variants and can reuse the configuration of the specialized SPLs in other SPL compositions. Names of class members (e.g., `mailList` and `errMsgList`) are used to identify instances of an SPL.

Domain constraints are defined in the domain model and are still used to define constraints that apply for all instances of an SPL. For example, constraint `MailClient.Debug => List.Logging` (cf. Fig. 2) means that feature DEBUG of the MailClient SPL requires feature LOGGING of the List SPL. We can now provide additional constraints for specialized SPLs. For example, we can use `MailClient.Debug => SyncList.Logging` to enable feature LOGGING only in instances of synchronized lists because SyncList is a specialized variant of the List SPL. These constraints are part of the MailClient SPL and are separated from reusable specialized variants defined in the List SPL.

Instance Constraints. We use *instance constraints* to describe dependencies between SPLs and concrete instances. As an example, consider the model for a sensor network in Figure 5. The `SensorNetwork` SPL uses specialized instances of SPLs `Client` and `NetworkNode`. The specialized variant `DataNode` again uses an instance of SPL `DBMS` to store data. In the lower part, we depict constraints of the model. Domain constraint (1) is part of the domain model and shown for completeness. Additionally, we specify constraints between SPLs and specialized variants (2): feature PDA implies feature BLUETOOTH only in specialized variant `AccessNode`. We also used an instance constraint (3): if feature DATASTORAGE is used, we enable feature QUERIES in instance `pda` of the `SensorNetwork` SPL. Thus, only a concrete instance is affected and not the whole SPL.




```
domain constraints:
(1) SensorNetwork.Logging => Client.Logging
(2) SensorNetwork.PDA => AccessNode.Bluetooth

instance constraints:
(3) SensorNetwork.DataStorage => SensorNetwork.pda.Queries

conditional dependencies:
(4) SensorNetwork.Laptop => SensorNetwork.laptop
(5) SensorNetwork.PDA => SensorNetwork.pda
(6) SensorNetwork.DataStorage => SensorNetwork.dataNode
```

Figure 5. A sensor network SPL using network nodes and client applications. The specialized DataNode SPL itself uses a DBMS SPL.

In contrast to specialization, it might be simpler to define instance constraints; however, specialization provides better reuse because the specialized variants can also be used in other SPLs. The constraints presented here, are only *requires* or *implies* constraints but arbitrary propositional formulas might be supported [1]. Object Constraint Language (OCL) might be a good candidate for specifying the constraints [4].

Conditional Dependencies. A description of a composition of multiple SPLs using the model presented above includes all SPLs and SPL instances that might be needed in arbitrary configurations. Some of these SPLs and SPL instances, however, are only needed if features that use those SPLs are actually present in a concrete configuration. For example, if feature DATASTORAGE of the `SensorNetwork` SPL is not used, we also do not need special network nodes that store data. This means that such SPLs do not have to be considered at configuration time if we know that they are not used. We achieve this by defining constraints for attributes of SPLs. In the example shown in Figure 5, we used constraints 4–6 to define that attributes `laptop`, `pda`, and `dataNode` are only needed if a particular feature is selected. Hence, if feature DATASTORAGE is not selected, there will be no instance of SPL `DataNode`. Other SPLs that `DataNode` depends on are also not added to the configuration. In our example, this applies to SPL `DBMS` which, as a result, does not have to be configured. This is similar for SPLs that have specialized variants. These are only added to a configuration if there is an instance of the SPL itself or one of its specializations needed. For example, if there is neither an instance of `Laptop` nor of `PDA`, the whole `Client` SPL is not needed. Conditional dependencies are mappings of features to elements of the instance model. To implement such mappings we propose to use existing tools, e.g., FeatureMapper [9]. This allows us to visualize such dependencies and use views on a composition model to show only elements that belong to a particular feature or a number of features.

The Configuration Process. The configuration of a composition of SPLs is based on an instance model and the domain models of all involved SPLs. The configuration process should be supported by a configuration tool and the underlying models need not to be shown to the user. When creating a configuration, the user starts with an empty feature selection of the top-level SPL (e.g., SPL `SensorNetwork` in Fig. 5) and selects functional features. This selection is based on the domain model of an SPL but is additionally checked against constraints in the instance model. Dependent SPLs are only added when they are needed. This avoids conflicts of SPLs that are not used and simplifies the verification of a feature selection. After finishing the configuration of one SPL the configuration of dependent SPLs follows. Ideally, there is no further selection of functionality of dependent SPLs needed because it is implicitly provided via constraints; however, manual configuration of underlying SPLs might be needed or wanted.

In the following, we describe how a configuration for the SensorNetwork SPL can be derived. We start with an empty feature selection which includes instances of SPLs `SensorNode` `AccessNode` that are always needed in a network. The user selects feature DATASTORAGE which adds an instance of SPL `DataNode` (conditional dependency 6) and SPL `DBMS`. Selection of feature PDA adds an instance of SPL `PDA`. The configuration of functional features is finished with this selection and constraints have to be checked. Constraint (1) is ignored because feature LOGGING is not selected. Constraint (2) enables feature BLUETOOTH in all instances of the `DataNode` SPL and constraint (3) enables feature QUERIES in instance `pda` of SPL PDA. After that, all dependent SPLs have to be configured. This is done per SPL instance, e.g., if we would have selected clients `pda` and `laptop`, both have to be configured separately. In our case, there are no other features to be configured within dependent SPLs and the configuration is already finished. This should be the usual scenario, but there might also be a detailed configuration process of lower-level SPLs like the `DBMS` SPL.

5. Discussion and Further Work

The presented model can be used to describe SPLs that are composed from other SPLs. However, there are some issues that have to be further analyzed and can be subject to extensions of the presented model. In the following, we discuss some of these issues.

Integration with Domain Models. To provide an integrated software product line engineering process SPL instance modeling has to be integrated with domain modeling. Based on a domain model that describes constraints between features of one SPL and also constraints to other SPLs we can ease the transition to SPL instance modeling using a model transformation. The result is a basic instance model that consists of classes representing SPLs and constraints between these SPLs which represent domain constraints. Staged configuration of SPLs can also be automatically transformed into an inheritance hierarchy of classes that represent different stages of configuration. There are only a few steps to support such a transformation:

- create a class for each feature model of a composition of SPLs and
- create a hierarchy of subclasses to represent staged configuration.

Resulting classes that represent SPLs or specialized versions of an SPL can be linked to their feature model to support visualization, e.g., using a feature diagram. Since there are a number of special extensions to feature models, e.g., cardinality based feature models, a mapping of these models has to be further analyzed. In this paper, we have not clearly defined which types of constraints should be supported. There are a number of approaches to define constraints in domain models (e.g., [4, 7, 15, 1]) which should also be applicable to the presented instance model. Which approach is used, however, is not important for the presented solution in general.

Tool Support. Mapping domain models and instance models is the basis for tools that support development of such models and automates the configuration process. Further visualization support is possible by mapping features to elements in the instance model (conditional dependencies) using tools like FeatureMapper [9]. In further work, we aim at developing an integration of existing tools and an automated configuration process as part of FeatureIDE. [1] FeatureIDE is a plug-in for the Eclipse IDE, used to support the complete SPL development process. It is based on feature-oriented programming and supports domain models in the *guidsl* format [1].

Configuration Generators. As an extension to this basic tool support we want to use the presented model to derive configuration generators. These generators can be created by generating OO code from the instance model as supported by current UML tools. The model can be extended using an object-oriented language (e.g., Java) to include user-defined code. This code can include code specific to a composition technique and also code to interact with a user in the configuration process. By using an OO language for configuration we can directly access SPLs that are represented by classes and make use of polymorphism and method overriding to simplify SPL configuration. Execution of the resulting configuration generator results in an interactive configuration process for the composition of dependent SPLs.

Adaptation to the Environment. The presented approach can be applied to systems developed as SPLs where the developer has access to all subsystems (i.e., used SPLs) to configure them according to the needs of the top-level SPL. However, an SPL also interacts with its environment, i.e., the operating system, hardware, other software, etc., which usually cannot be changed. An SPL also has to be configured with respect to this external variability. Using the presented model we can also represent *external SPLs* (e.g., an operating system SPL [16]) and create constraints between the SPL of the problem domain and SPLs of the environment. These constraints have to ensure that the domain SPL configuration changes according to the environment. Providing a configuration for external SPLs as they appear in a concrete scenario (e.g., describing the actually used hardware) results in an SPL configuration that automatically adapts to this environment.

6. Related Work

There is a large amount of work addressing domain modeling and dependencies between multiple SPLs. Cardinality-based feature models with constraints were proposed by Czarnecki et al. [6, 12]. They allow a domain engineer to specify specializations and constraints in feature models where multiple selections of one feature are possible. The used *feature model references* [6] and *feature cloning* might be applicable for modeling product line instances; however, it mixes (1) domain modeling with domain implementation of a product line (handling instances of other product lines, etc.) and (2) does not provide means to create named instances of used product lines which is needed for implementation. Application product lines consuming different service-oriented product lines in a SOA environment where described by Trujillo et al. [17]. Their focus was on modeling the interfacing between SPLs in a service-oriented environment. This includes service registration and service consumption. Hence, their work is complementary to the presented approach and both might be combined in service-oriented environments. An approach that integrates feature models of different product lines was presented by Streitferdt et al. [16]. Their goal is to derive the configuration of a hardware product line based on the requirements of an SPL for embedded systems. The presented integration of multiple SPLs does not consider SPL instances or instance constraints which were not needed in their context.

In contrast to these modeling approaches, we found that feature models and dependencies between them are not sufficient to describe compositions of dependent SPLs where multiple instances of the same SPL are used. As a solution, we propose a model that describes SPL instantiation and dependencies between SPL instances. We see our approach as an extension of other product line modeling techniques and we think that their combination is needed to completely describe complex product lines that are composed with other product lines.

Product populations built from Koala components were described by van Ommering [18]. Koala components can be recursively built from smaller components leading to a set of complex products which is similar to dependent product lines described here. The focus of van Ommerings work was on interactions between components via interfaces using different connectors to support flexible component composition. Interfaces between components and their description, e.g., as defined in Koala, are also needed for safe composition when using our approach. Hence, in this respect the Koala approach is complementary to our work. Furthermore, the goal of our work is to describe compositions of SPLs independent of the implementation technique by focusing on features and dependencies between SPLs. Koala components are defined by composing smaller components at configuration time which is in contrast to our work. We aim at defining compositions of whole SPLs and not concrete components. That is, the composition of a concrete 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 or a concrete product.

Fries et al. presented an approach to model SPL compositions for embedded systems [8]. They use *feature configurations* which are a selection of configured features to describe a group of instances that share this feature selection. Hence, feature configurations are similar to specialized SPLs in staged configuration; however, they do not allow a user to describe multiple configuration steps or sub-typing between specialized variants. A composition model described by Fries et al. is defined for a complete composition of product line instances. Our approach uses an instance model that is part of a product line and defines a composition of related SPLs. Each referenced SPL itself has its own instance model defining other SPLs it is composed from. Hence, we define the composition for each SPL separately which eases reuse of instance models. Furthermore, we map features to referenced SPLs and SPL instances and combine instance and feature models of multiple SPLs only when this is needed, i.e., when a feature that references another SPL is selected. This avoids any evaluation of composition rules of product lines that are not used.

Tools like *pure::variants*[2] and *Gears*[3] allow a domain engineer to build feature models and also to describe dependencies among them. Both tools support modeling of dependencies between product lines and Gears explicitly supports nested product lines that can be reused between different product lines. *guidsl* is a tool to specify composition constraints for feature models using a grammar [1]. It provides means to check models and interactively derive a configuration for a feature model.

Batory et al. have shown that SPL development using layered designs scales to *product lines of program families*. The focus of their work was on generating families of programs from a single code base and reasoning about program families. The work does

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

[2] http://www.pure-systems.com

[3] http://www.biglever.com

not address relations between different product lines developed independently or between instances of such product lines.

7. Conclusion

Compositions of SPLs are used to structure and decompose large SPLs and also to reuse SPLs within other SPLs. Current feature models can be used to describe such compositions only if an SPL uses one instance of other SPLs. This is not sufficient if multiple instances of the same SPL are used in a larger SPL.

We presented an approach based on class diagrams and OOP that extends domain modeling. We provide means to model SPLs, SPL instances, their relationships, and constraints between them. In our model, *nested* or *hierarchical* SPLs, where only one instance of each involved SPL is used, are included as a special case. The presented model describes the high-level architecture of compositions of SPLs and their dependencies. We propose to use it to complement domain modeling and integrate it into the SPL development process if multiple SPLs are involved. We showed how conditional dependencies can be handled by using constraints that map features of an SPL to referenced instances of other SPLs. This serves a better understanding of compositions of dependent SPLs (e.g., supported by advanced visualization techniques) and can be used to automate the configuration process of a whole SPL composition scenario.

Acknowledgments

Marko Rosenmüller and Norbert Siegmund are funded by German Research Foundation (DFG), project number SA 465/32-1 and German Ministry of Education and Research (BMBF), project number 01IM08003C.

References

[1] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Verlag, 2005.

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

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

[5] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-level Configuration of Feature Models. In *Software Process Improvement and Practice 10*, pages 143–169, 2005.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Verlag, 2004.

[7] A. Deursen and P. Klint. Domain-specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[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*, pages 224–228. ACTA Press, 2007.

[9] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 943–944. ACM Press, 2008.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[12] 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)*, pages 331–348, 2005.

[13] C. W. Krueger. New Methods in Software Product Line Development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 95–102. IEEE Computer Society Press, 2006.

[14] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Verlag, 1997.

[15] D. Streitferdt, M. Riebisch, and I. Philippow. Details of Formalized Relations in Feature Models Using OCL. pages 297–304. IEEE Computer Society Press, 2003.

[16] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow. Configuring Embedded System Families Using Feature Models. In *Proceedings of Net.ObjectDays*, pages 339–350. Gesellschaft für Informatik, 2005.

[17] 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?*, 2007.

[18] R. van Ommering. Building Product Populations with Software Components. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 255–265. ACM Press, 2002.